IDENTIFYING TESTING REQUIREMENTS FOR MODIFIED SOFTWARE

A Dissertation
Presented to
The Academic Faculty

by

Taweesup Apiwattanapong

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

Georgia Institute of Technology
August 2007

IDENTIFYING TESTING REQUIREMENTS FOR MODIFIED SOFTWARE

Approved by:

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

Dr. Santosh Pande
College of Computing
Georgia Institute of Technology

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Dr. John Hatcliff
Departement of Computing and
Information Sciences
Kansas State University

Dr. Panagiotis Manolios
College of Computing
Georgia Institute of Technology

Date Approved: 21 June 2007

To my family,

for their unconditional love and support

ACKNOWLEDGEMENTS

The six years that I spent at Georgia Tech in Atlanta working on my PhD is the time I shall cherish. When I first came to Atlanta to pursue my degree, my aspiration was to learn how to do research. Now that I have fulfilled this aspiration, I can see how this special experience and my interactions with many exciting and generous people have shaped not only my research ideas but also my perspective on life in general. I owe a debt of gratitude to many people, and it would be impossible to mention them all here. I consider myself fortunate to have had the pleasure of pursing my PhD under the tutelage of such talented and inspiring mentors as Dr. Mary Jean Harrold and Dr. Alessandro Orso. I owe them a lot of gratitude not only for guiding me through the technical intricacies of program analysis but also for their constant encouragement and trust in my abilities. I also would like to thank the members of my committee: Dr. Panagiotis Manolios, Dr. John Hatcliff, and Dr. Santosh Pande, for their useful comments and stimulating discussions that have helped improve this dissertation.

Jim Bowring was my graduate school colleague, a roommate, and a landlord, and in each of these capacities he was always eager to share his wisdom and offer advice. I was fortunate to meet Eli Tilevich who has taught me how to push my limits, both intellectual and physical.

I would like to thank all the present and former members of the Aristotle Research Group, Saswat Anand, George Baah, Pavan Kumar Chittimalli, Jim Jones, Donglin Liang, Mikel Pennings, Raul Santelices, Hina Shah, Saurabh Sinha, Yi Zhang, and others for their friendship and support.

My friends from the Thai Student Organization helped make the rare free moments of my stay in Atlanta more enjoyable.

And last but not least, I would like to thank my family for their love and patience. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Throughout its lifetime, software must be changed for many reasons, such as bug fixing, performance tuning, and code restructuring. Testing modified software is the main activity performed to gain confidence that changes behave as they are intended and do not have adverse effects on the rest of the software. A fundamental problem of testing evolving software is determining whether test suites adequately exercise changes and, if not, providing suitable guidance for generating new test inputs that target the modified behavior. Existing techniques evaluate the adequacy of test suites based only on control- and data-flow testing criteria. They do not consider the effects of changes on program states and, thus, are not sufficiently strict to guarantee that the modified behavior is exercised. Also, because of the lack of this guarantee, these techniques can provide only limited guidance for generating new test inputs.

This research has developed techniques that will assist testers in testing evolving software and provide confidence in the quality of modified versions. In particular, this research has developed a technique to identify testing requirements that ensure that the test cases satisfying them will result in different program states at preselected parts of the software. This research has also developed supporting techniques for identifying testing requirements. Such techniques include (1) a differencing technique, which computes differences and correspondences between two software versions and (2) two dynamic-impact-analysis techniques, which identify parts of software that are likely affected by changes with respect to a set of executions.

CHAPTER 1

INTRODUCTION

Software is constantly subject to pressures for changes and perceived to be easily malleable [27]. Thus, software has always undergone continual modifications throughout its lifetime. Because changes occur frequently and continually, the problem of testing modified versions of software with respect to these changes in an efficient and effective way is important. *Regression testing* is the activity of testing modified versions of software to increase the confidence that the changes behave as intended and do not adversely affect the rest of the software. This testing activity has always been challenging because developers need to check not only the intended functionality of the changes themselves, but also the intended functionality of the rest of the software that interacts with the changes. Much research in regression testing has concentrated on three activities: regression-test selection (e.g., [13, 64, 74]), test-suite prioritization (e.g., [66, 70]), and test-suite reduction (e.g., [31, 65, 77]). These activities aim to improve the efficiency of regression testing by reducing the number of test cases needed to be rerun and maintained and reordering test cases based on some criteria, such as coverage. Little research, however, has concentrated on assessing and improving the quality of regression test suites. Most of such research may overestimate the adequacy of existing test suites in exercising the software with respect to changes. Furthermore, such research often does not provide suitable guidance for creating new test inputs that specifically target the changed behavior of the software when existing test suites are not adequate.

1.1 Goal and Scope

The goal of this research is to assess the quality of existing test suites with respect to changes. To address this goal, this research focuses on the problem of defining testing criteria and identifying testing requirements that can be used to determine the extent to which the changes are exercised and to guide the generation of new test inputs targeting

the changes.

1.2 Proposed Thesis

The thesis of this research is that static program analysis techniques can effectively compute differences and correspondences between entities in two versions of a program and, combined with dynamic analysis, can leverage the computed information to identify the requirements for testing evolving software effectively and efficiently.

1.3 Overview of the Dissertation

The process of generating requirements for testing changes, depicted in Figure 1, consists of three activities (differencing, dynamic impact analysis, and testing-requirements identification) that require as input two versions of a program: the original version, P, and a modified version, P'. To facilitate the discussion of these activities in subsequent chapters, Chapter 2 describes the necessary background material. This material includes fundamental concepts in the area of program analysis: control-flow graph, control and data dependences, and symbolic execution. The next three chapters present a new technique to identify testing requirements and its supporting techniques in the order of the process flow, shown in Figure 1. More precisely, Chapter 3 presents a differencing technique that reports change information, which includes information about differences and correspondences, for entities in P and P' at the statement level. Chapter 4 presents two dynamic-impact-analysis techniques that use the change information and a test suite (or a set of executions), T, to identify the parts of P' that are likely affected by changes (impact information) in those executions and discusses the tradeoffs between the two techniques. Chapter 5 presents a technique that uses the change and impact information to identify requirements for testing that can effectively assess the quality of test suites with respect to testing the changes and guide the generation of new test inputs targeting the changes. Each of these three chapters first discusses related work and its deficiencies, then describes each of the techniques and the implementation of the tool for each technique. Chapter 6 concludes the dissertation with merit and future directions of this research.

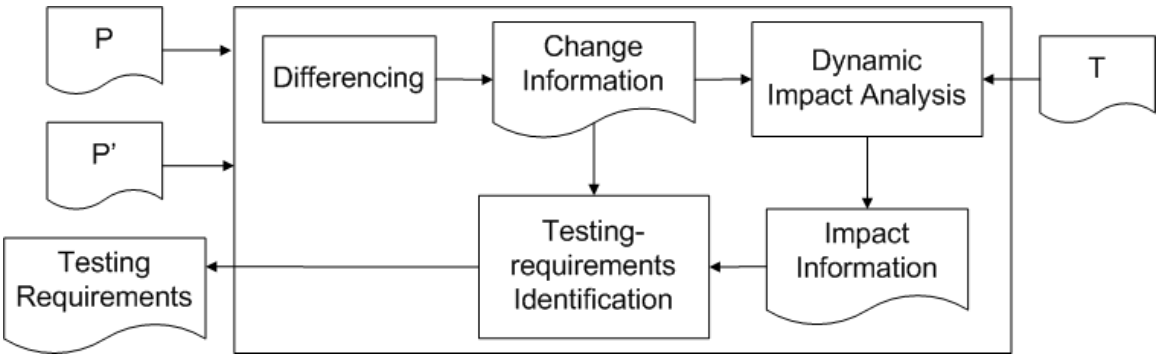


Figure 1: The testing-requirements-generation process.

1.4 Contributions

This research provides a number of contributions in the area of software testing.

1. An automated algorithm to compute the differences and correspondences between entities in two versions of a program;

2. Two techniques to identify parts of software that are likely to be affected by changes with respect to a set of executions;

3. A technique to identify testing requirements for modified versions of software that ensure that the test cases satisfying them will result in different control flows or different program states at selected points;

4. Implementations of the developed techniques; and

5. Empirical studies to evaluate the techniques on real programs.

CHAPTER 2

BACKGROUND

This chapter presents background material that facilitates the discussions of the new techniques in subsequent chapters. The next sections describe fundamental concepts in the area of program analysis: control-flow graph, control and data dependences, and symbolic execution.

2.1 Control-Flow Analysis and Representation

Most program-analysis techniques need the information about the flow of control within and between procedures in programs. Control-flow analysis [1] computes this information by analyzing statements in the programs. The control-flow information is usually represented using control-flow graphs. The following definition is based mainly on Aho et al.'s definition of control-flow graph [1].

Definition 1. A *control-flow graph* (CFG) $G = (N, E)$ for a procedure P is a directed graph in which N contains one node for each statement in P and E contains edges that represent possible flow of control between statements in P. N also contains unique entry and exit nodes, which represent the entry to and exit from P, respectively. An edge in E leaving a predicate node is labeled T (for true) or F (for false), which represents a control path taken when the predicate evaluates to that value.

As an example, Figure 2 shows method *getCheckedOutBooks* in class *Library* and its CFG.

2.2 Dependence Analysis

Another analysis used in this research is dependence analysis [16, 24], which identifies the dependence relationships between statements in programs. The material in this section is based on the work presented Ferrante et al. [24]. This research considers two classes of

```
  public class Library {
    Set<Book> getCheckedOutBooks() {
s1    Set<Book> chOut = new HashSet<Book>();
s2    for (Book book : books)
s3        if (book.getStatus() == Book.CHECKED_OUT)
s4            chOut.add( book);
s5    if (chOut.isEmpty())
s6        chOut = new EmptySet<Book>();
s7    return chOut;
    }
    ...
  }
```

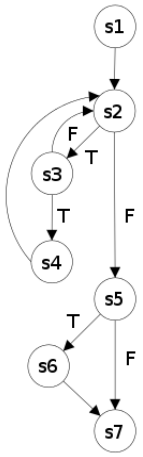


Figure 2: Partial class *Library* and control-flow graph of method *getCheckedOutBooks*.

dependences. A control dependence occurs between a statement and a predicate when the value of the predicate determines whether the statement is executed. A data dependence occurs between two statements when a variable used in one statement may contain incorrect value if the two statements are reversed. Next, control and data dependences are defined more formally.

Definition 2. Let $G = (N, E)$ be a CFG, and let $u, v \in N$. Node u is *control dependent* on node v with label 'L' if and only if v has successors v' along outgoing edge labeled 'L' and v'' such that every path in G from v' to the exit node contains u but at least one path in G from v'' to the exit node does not contain u.

For example, consider the control-flow graph of method *getCheckedOutBooks* in Figure 2: node $s3$ is control-dependent on node $s2$ with label 'T' because $s2$ has two successors, $s3$ and $s5$, and every path from $s3$ to the exit node contains $s3$, but no path from $s5$ to the exit node contains $s3$.

Definition 3. Let $G = (N, E)$ be a CFG, and let $u, v \in N$. Node v is *data dependent* on node u if and only if node u defines a variable x, v uses x, and there is a path in G from u to v such that no other node on that path defines x.

For example, in method *getCheckedOutBooks*, node $s7$ is data dependent on node $s1$ because node $s1$ defines variable *chOut*, $s7$ uses *chOut*, and no other node on path $(s1, s2, s5, s7)$ defines *chOut*.

2.3 Symbolic Execution

Symbolic execution [14, 36, 40] is a program-analysis technique that, unlike normal executions that exercise a program on a set of concrete inputs, exercises a program symbolically on a set of *classes* of inputs. Symbolic execution computes a symbolic representation of the computations of program variables along a path (from entry to exit) and the domain of that path in terms of input values. For any path, the symbolic representation of the path computation (the values of all output variables at the end of the path) and path domain (the domain of input variables for the path to be executed) can be developed incrementally as statements on the path are interpreted. To develop this representation, symbolic execution assigns each input a symbolic value and evaluates a path by interpreting the statements on that path in terms of these symbolic values. During execution, the values of program variables are maintained as algebraic expressions over the symbolic names. At each point along a path, values of variables are represented as a vector $(s(y_1), s(y_2), ...)$, where $s(y_i)$ denotes the current symbolic value of variable y_i. The path computation is a vector of values of all output variables at the end of the path. The path domain is also formed incrementally by interpreting branch predicates for the conditional statements on a path. At each branch predicate p, expressed in terms of symbolic names, the predicate is evaluated by substituting each variable with its symbolic expression to obtain a predicate with substitution p_s. If the path takes the true branch of that predicate, the domain is restricted by constraint p_s; otherwise, it is restricted by constraint $\neg p_s$. Thus, the path domain is a conjunction of constraints $p_{s,i}$ or $\neg p_{s,i}$, depending on the branches taken, for all branch predicates on the path. For non-executable paths, the conjunction of constraints is unsatisfiable, and thus no concrete values of inputs can lead to that path.

Three types of symbolic execution have been presented in the literature: path-dependent symbolic execution, dynamic symbolic execution, and global symbolic execution. These types differ primarily in their techniques to select paths to be executed. *Path-dependent symbolic execution* chooses the paths from user input or heuristics used in the symbolic-execution engine. *Dynamic symbolic execution* selects paths based on the paths that are executed by specific input data. *Global symbolic execution* does not select a single path to

be executed but attempts to create a symbolic representation of path computations and path domains that represents all paths.

2.4 Decision Procedures

When executing a program symbolically, a symbolic-execution engine needs to check whether a subpath being explored so far is feasible. To achieve this, such an engine requires a decision procedure (a method for solving a decision problem), to solve the feasibility of the subpath (i.e., the satisfiability of the path constraints of that subpath). The decision procedure must be able to handle the satisfiability problem in logics that are more expressive than propositional logic. In particular, symbolic execution concerns the satisfiability problem where formulas contain atoms that are interpreted with respect to background theories of real and integer arithmetic and theories of arrays, lists, and other data structures. This problem is known as the *Satisfiability Modulo Theories* (SMT) problem for a theory T: given a formula (e.g., path constraints) F, determine whether F is T-satisfiable.

There are two broad approaches to SMT: the eager approach and the lazy approach. The *eager* approach translates the input formula in a single satisfiability-preserving step into a propositional CNF formula and uses a SAT solver to check its satisfiability. The *lazy* approach, however, initially considers each atom occurring in a formula F simply as a propositional symbol and sends the transformed formula to a SAT solver. If the SAT solver reports that the formula is unsatisfiable, the original formula F is also T-unsatisfiable. If the SAT solver reports a propositional model of F (i.e., F is satisfiable), this model (a conjunction of literals) is checked by a T-solver. If the solver reports that the model is satisfiable, then F is T-satisfiable. Otherwise, the T-solver returns feedback that can guide the SAT solver in constructing a new model of F. This process is repeated until the SAT solver finds a T-satisfiable model or returns unsatisfiable.

In recent years, the DPLL(T) approach presented by Davis et al. has become the mainstream lazy approach for SAT Modulo Theories [28, 52]. This approach is based on the DPLL procedure for propositional logic [17, 18]. The DPLL(T) approach consists of a general DPLL engine, called DPLL(X), that is independent from any particular theory T and

a solver for a theory T of interest. A system DPLL(T) for deciding the satisfiability of conjunctive-normal-form (CNF) formulas in a theory T is produced by instantiating the parameter X with a module $Solver_T$ that can handle conjunctions of literals in T. There are several implementations of the DPLL(T) approach (e.g., Yices [22] and BarceLogic [51]). Yices is a SMT solver that decides the satisfiability of arbitrary formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, fixed-size bit vectors, quantifiers, and lambda expressions. BarceLogic supports difference logic over integers or reals, equality with uninterpreted function symbols, and the interpreted function symbols *predecessor* and *successor*.

CHAPTER 3

DIFFERENCING

The first step toward generating testing requirements for modified software is identifying changes that have been made between the original and modified versions (as shown in Figure 1). Precise information about changes enables the testing-requirements identifier to generate only requirements that actually reveal different behavior when running the modified version on test inputs satisfying them. Imprecise change information may cause the identifier to generate testing requirements that are misleading: test inputs satisfying such requirements do not reveal different behavior in the modified version. The identifier requires not only the change information but also the mappings between program entities in the original and modified versions. The identifier generates testing requirements by comparing program states at statements in the original version with program states at the corresponding statements in the modified version. (Section 5.2 provides further description of this step.) Thus, the identifier requires the mappings between statements in the two versions. Differencing techniques can provide both the change information and the mappings.

3.1 Related Work

Several techniques and tools for comparing source files textually (e.g., the UNIX `diff` utility [49]) have been proposed. However, these techniques have shortcomings. Textual differencing may report changes that have no effect on program semantics or syntax, such as the addition of a method that is never called and modifications in comments and white spaces, and do not consider changes in program semantics indirectly caused by textual modifications.

Consider, for example, the two versions of a partial Java program in Figure 3: the original version P and the modified version P'. Both versions use the class *Library*, shown in Figure 2. The output of `diff` running on P and P' would show that method *EmptySet.addAll* has been inserted and that the exception-type hierarchy has changed. However, detecting

Program P

```
public class UnavailBookFinder {
  private Library lib;
  ...
  public  static void main(String[] args) {
    Set<Book> res =
        lib.getCheckedOutBooks();
    if (includesOnOrder) {
      try {
        res.addAll( lib.getOnOrderBooks());
      }
      catch(UnsupportedOpException e){...}
      catch(RuntimeException e){...}
    }
  }
}

public class EmptySet<T>
    extends AbstractSet<T> {



  public int size() { return 0; }
  ...
}

public class UnsupportedOpException
    extends RuntimeException {...}
public class MyUnsupportedOpException
    extends UnsupportedOpException {...}
```

Program P'

```
public class UnavailBookFinder {
  private Library lib;
  ...
  public static void main(String[] args) {
    Set<Book> res =
        lib.getCheckedOutBooks();
    if (includesOnOrder) {
      try {
        res.addAll( lib.getOnOrderBooks());
      }
      catch(UnsupportedOpException e){...}
      catch(RuntimeException e){...}
    }
  }
}

public class EmptySet<T>
    extends AbstractSet<T> {
  public void addAll(Collection<T> col){
    throw new MyUnsupportedOpException();
  }
  public int size() { return 0; }
  ...
}

public class UnsupportedOpException
    extends RuntimeException {...}
public class MyUnsupportedOpException
    extends RuntimeException {...}
```

Figure 3: Partial code for an original version (P) and a modified version (P').

that the call to *res.addAll* in *UnavailBookFinder.main* of P and P' can be bound to different methods, and the exception can be thrown by the call to *res.addAll* in method *UnavailBookFinder.main*, would not be straightforward without additional analyses. For another example, consider the case when running `diff` with the intermediate version where only method *EmptySet.addAll* is inserted as P and the version with both changes as P'. The output of `diff` on this pair of P and P' would not show that the exception that may be thrown by the call to *res.addAll* in method *UnavailBookFinder.main* can be caught by different catch blocks.

Other existing differencing techniques are specialized to compute differences in programs. These techniques can be divided into two categories (which are discussed in Sections 3.1.1 and 3.1.2, respectively) : general-purpose differencing techniques and goal-specific differencing techniques.

3.1.1 General-purpose Program Differencing

These techniques can be further categorized into three groups: text-based program differencing, syntactic-based program differencing, and semantic-based program differencing.

3.1.1.1 Text-based Program Differencing

Maletic and Collard's approach [47] is a text-based program-differencing technique. Their technique transforms C/C++ source files into a format called srcML that makes program structures more explicit than raw source code and leverages `diff` to compare the srcML representations for the original and modified versions of the source code. (srcML is an XML-based format that represents the source code annotated with syntactic information.) The results of the comparison are then post-processed to create a new XML document, also in srcML format, with the additional XML tags that indicate the common, inserted, and deleted XML elements. Their approach utilizes available XML tools to ease the process of extracting change-related information. However, the technique is limited by the fact that it still relies on line-based differencing information obtained from `diff`.

3.1.1.2 Syntactic-based Program Differencing

The techniques in this group compare two versions of a program syntactically. They usually operate on abstract-syntax-tree representations or control-flow-graph representations of programs; thus, they can ignore textual differences that do not affect the programs, such as changes in program comments.

Several modern integrated development environments, such as Eclipse [25], incorporate a parser for the programming languages they support. Therefore, they can compare different versions of a program more effectively than tools based on simple textual comparison. However, the comparison capabilities of these tools are still limited, in that they recognize changes only at the purely-syntactic level. For example, they cannot identify indirect differences at the statement level due to changes involving object-oriented features.

Laski and Szermer present an algorithm that computes program differences by analyzing

the control-flow graphs of the original and modified versions of a program [42]. Their algorithm localizes program changes into clusters, which are single-entry, single-exit program fragments. Clusters are reduced to single nodes in the two graphs and are then recursively expanded and matched. Their control-flow graph representation does not model the object-oriented behaviors properly; thus, their algorithm may not compute accurate change information for object-oriented programs. For example, their algorithm cannot detect a difference at a call site that may invoke a method that has just been added in the modified version due to method overriding. Moreover, their algorithm for matching clusters has limited capability and may compute imprecise results. Their algorithm uses only the entry statements of two clusters to determine whether the two clusters are matched. If only the entry of one cluster in the modified version is changed, their algorithm may report that none of the statements in the cluster is matched. Their algorithm also does not allow matching of clusters at different nested levels .Thus, it may compute imprecise results.

BMAT (Binary MATching tool) [75] performs matching of both code and data blocks between two versions of a program in binary format. BMAT uses a number of heuristics to finding matches for as many blocks as possible. These heuristics, however, are specific to finding matches at the binary level and cannot be applied to the source or Java bytecode levels. Moreover, BMAT does not compute information about changes related to object-oriented constructs, such as method overriding or changes in class hierarchy.

3.1.1.3 Semantic-based Program Differencing

The techniques in this group compare two versions of a program semantically and identify the differences only when program semantics changes due to syntactic modification.

Semantic diff [38] compares two versions of a program procedure-by-procedure, computes a set of input-output dependences for each procedure, and identifies the differences between two sets computed for the same procedure in the original and modified versions. Semantic diff is performed only at the procedure level and may miss changes that, although not affecting input-output dependences, may drastically affect program behavior (e.g., constant value change).

Horwitz's approach [34] computes both syntactic and semantic differences between two programs using a partitioning algorithm. The approach models programs using a Program Representation Graph (PRG), a representation defined only for programs written in a language with scalar variables, assignment statements, conditional statements, while loops, and output statements. Because of this limitation in its underlying representation, Horwitz's approach cannot be applied to programs written in traditional languages, such as C, C++, or Java. Although, the PRG could be exteended to include features of such languages, the partitioning algorithm may not scale to large programs.

3.1.2 Goal-specific Program differencing

The approaches in this category compute differencing information for some specific goals and, thus, produce differencing information that is targeted to such goals. Binkley's approach [6] computes semantic differences between two program versions to reduce the cost of regression testing. The approach first identifies, on a System Dependence Graph (SDG), unmatched nodes and nodes with different incoming data- or control-flow edges in the two versions of the program considered. Then, it performs forward slicing starting from such nodes to identify all affected nodes in the graph.

Raghavan and colleagues present a differencing tool called DEX [59]. DEX compares two abstract semantic graphs (i.e., abstract syntax trees augmented with extra edges that encode type information) that represent two versions of a program. At the end of the comparison, DEX produces an edit script that contains the transformations necessary to transform the semantic graph of the original version into the semantic graph of the new version. The tool also collects change-related statistics to reveal some facts about the nature of bug fixes in software projects.

Ren and colleagues present CHIANTI, an impact analysis tool that uses a differencing engine to identify atomic changes between two versions of a Java program and dependences among these changes [61]. Rangarajan also uses the notion of atomic changes at the class and method levels and presents a tool called JEVOLVE [60], which analyzes Java programs and identifies modified classes that must be regression tested. These two differencing tools

operate at the method level and their goal is to provide differencing information to identify which test cases are affected by the changes between two versions. (CHIANTI also identifies which changes affect which test cases.) Therefore, the results of these techniques are imprecise for several program-analysis techniques including testing-requirements identification, which is the intended application of this research.

Xing and Stroulia presented UMLDIFF [78], an algorithm for automatically detecting structural changes between the designs of subsequent versions of an object-oriented program. UMLDIFF compares the class diagrams of two program versions and uses structural information to identify the differences between the two versions. Differences are identified in terms of addition, removal, moving, and renaming of packages, classes, interfaces, fields, and methods. UMLDIFF works at the class and method levels and does not compares statements in matched method pairs. Furthermore, UMLDIFF is mostly targeted at identifying design-level changes.

3.2 Differencing Algorithm

As discussed in the Introduction (Section 1.3), the precise information about differences and correspondences between entities in two versions of a program is necessary for generating effective testing requirements. However, existing differencing approaches have several shortcomings, as discussed in Section 3.1. Moreover, most existing approaches report declaration changes (e.g., changes in method access modes and superclass declaration) as is. Consequently, techniques that require change information at the statement level cannot readily use such differencing results. To overcome problems with these approaches, this research defines a new graph representation and a differencing algorithm that uses the representation to identify changes at the statement level between two versions of a program. This new representation augments a traditional control-flow graph (CFG) to model behaviors caused by object-oriented features in the program. Using this graph, the technique developed in this research identifies declaration changes and relates them to the points at the statement level where the different behavior may occur.

The new algorithm, which extends an existing differencing algorithm [42], consists of

five steps. First, it matches classes, interfaces, and methods in the two versions. Second, it builds enhanced CFGs for all matched methods in the original and modified versions of the program. Third, it reduces each graph to a series of nodes and hammocks [24] (single-entry, single-exit subgraphs). Fourth, it compares, for each method in the original version and the corresponding method in the modified version, the reduced graphs to identify corresponding hammocks. Finally, it recursively expands and compares the corresponding hammocks and nodes. Section 3.2.1 gives an overview of the algorithm. Sections 3.2.2, 3.2.3, and 3.2.4 detail the levels at which the algorithm compares the original and modified versions of the program. Section 3.2.5 discusses the algorithm's complexity.

3.2.1 Overview

The new algorithm, `CalcDiff`, given in Figure 4, takes as input an original version of a program (P) and a modified version of that program (P'). The algorithm also takes as inputs two parameters—LH and S—that are used in the node-level matching. Parameter LH is the maximum lookahead that `CalcDiff` uses when attempting to match nodes in methods. Parameter S is used when determining the similarity of two hammocks. At completion, the algorithm outputs a set of pairs (N) in which the first element is a pair of nodes and the second element is the status—either "modified" or "unchanged." The algorithm also returns sets of pairs of matching classes (C), interfaces (I), and methods (M) in P and P'.

`CalcDiff` performs its comparison first at the class and interface levels, then at the method level, and finally at the node level. The algorithm first compares each class in P with the like-named class in P', and each interface in P with the like-named interface in P', and produces sets of class pairs (C) and interface pairs (I), respectively. For each pair of classes and interfaces, `CalcDiff` then matches methods in the class or interface in P with methods having the same signature in the class or interface in P'; the result is a set of method pairs (M). Finally, for each pair of concrete (i.e., not abstract) methods in M, the algorithm constructs Enhanced CFGs (hereafter, ECFGs) for the two methods and matches nodes in the two ECFGs.

The next sections give details of `CalcDiff`, using the code in Figure 3 as an example.

3.2.2 Class and Interface Levels

`CalcDiff` begins its comparison at the class and interface levels (lines 1–2). The algorithm matches classes (resp., interfaces) that have the same fully-qualified name; the fully-qualified name consists of the package name followed by the class or interface name. Pairs of matching classes (resp., interfaces) in P and P' are added to C (resp., I). Classes in P that do not appear in C are deleted classes, whereas classes in P' that do not appear in C are added classes. Analogous considerations hold for interfaces. In the example programs in Figure 3, each class in P has a match in P', and thus there is a pair in C for each class in P.

To improve the differencing results, `CalcDiff` also accounts for the possibility of interacting with the user while matching classes and interfaces. After matching classes and interfaces in P with classes and interfaces in P' that have the same fully-qualified names, `CalcDiff` provides users the possibility of defining additional matches; users can provide the algorithm with matches between unmatched classes (interfaces) in P and unmatched classes (interfaces) in P'. This additional feature accounts for cases in which the user renamed or moved one or more classes and interfaces. The interaction with the user can be implemented efficiently because additional matches are required only for unmatched classes (rather than all classes) in P.

3.2.3 Method Level

After matching classes and interfaces, `CalcDiff` compares, for each pair of matched classes or interfaces, their methods (lines 3-4). The algorithm first matches each method in a class or interface with the method with the same signature in the corresponding class or interface. Then, if there are unmatched methods, the algorithm looks for a match based only on the name. This matching accounts for cases in which parameters are added to (or removed from) an existing method, which are found to occur in preliminary studies of this research, and increases the number of matches at the node level. Pairs of matching methods are added to M. Like the approach used for classes, methods in P that do not appear in set M are deleted methods, whereas methods in P' that do not appear in set M are added

Algorithm `CalcDiff`

Input: original program P
modified program P'
maximum lookahead LH
hammock similarity threshold S
Output: set of ⟨class,class⟩ C
set of ⟨interface,interface⟩ I
set of ⟨method,method⟩ M
set of ⟨⟨node,node⟩,status⟩ N

Begin: `CalcDiff`
1: compare classes in P and P'; add matched class pairs to C
2: compare interfaces in P and P'; add matched interface pairs to I
3: **for** each pair $\langle c, c' \rangle$ in C or I **do**
4: compare methods; add matched method pairs to M
5: **for** each pair $\langle m, m' \rangle$ in M **do**
6: create ECFGs G and G' for methods m and m'
7: identify, collapse all hammocks in G to obtain G_c
8: identify, collapse all hammocks in G' to obtain G'_c
9: $N = N \cup$ `HmMatch`(G_c, G'_c, LH, S)
10: **end for**
11: **end for**
12: **return** C, I, M, N
end `CalcDiff`

Figure 4: Algorithm `CalcDiff`.

methods. In the example (Figure 3), there would be a pair in M for each method in P, but not for method *EmptySet.addAll* in P' (which would therefore be considered as added).

Analogous to the matching at the class level, the matching at the method level can also leverage user-provided information. Given two matching classes (or interfaces), c and c', the user can provide matches between unmatched methods in c and c'. User-provided matchings can improve the differencing by accounting for cases in which methods are renamed.

3.2.4 Node Level

For each pair of matched methods $\langle m, m' \rangle$ in M, `CalcDiff` builds ECFGs (Enhanced CFG) G and G' for m and m', respectively (lines 5-6). Then, the algorithm identifies all hammocks in G and G', and collapses G and G' to graphs G_c and G'_c (lines 7-8), respectively. Next, `CalcDiff` calls procedure `HmMatch`, passing G_c,G'_c, LH, and S as parameters. `HmMatch` identifies differences and correspondences between nodes in G and G' (line 9), and creates and returns N, the set of matched nodes and corresponding labels

("modified" or "unchanged"). Finally, `CalcDiff` returns N, C, I, and M (line 12).

The next three sections discuss the ECFG (the representation used in node matching), hammocks and how they are processed, and the hammock-matching algorithm, `HmMatch`, respectively.

3.2.4.1 Enhanced Control-Flow Graphs (ECFG)

When comparing two methods m and m', the goal of the algorithm is to find, for each statement in m, a match (or corresponding statement) in m', based on the method structure. Thus, the algorithm requires a modeling of the two methods that (1) explicitly represents their structure, and (2) contains sufficient information to identify differences and similarities between them. Although CFGs can be used to represent the control structure of methods, traditional CFGs do not suitably model many object-oriented constructs. ECFGs extend traditional CFGs and are tailored to represent object-oriented programs.[1] The rest of this section illustrates how the ECFG represents various object-oriented features of the Java language.

Dynamic Binding.

Because of dynamic binding, an apparently harmless modification of a program may affect call statements in different parts of the program with respect to the change point. For example, class-hierarchy changes may affect calls to methods in any of the classes in the hierarchy, and adding a method to a class may affect calls to the methods with the same signature in its superclasses and subclasses.

The ECFG models a call site, in which a method m is called on an object o, to capture these modifications. A call site consists of call and *return* nodes and a callee node for each dynamic type T that can be associated with o. A *callee* node represents the method that is bound to the call when the type of o is T and is labeled with the signature of that method.

[1]The ECFG is similar to the Java Interclass Graph (JIG) [32] in terms of the handling of variables, object types, and exception constructs. However, the JIG is an inter-procedural representation of a program whereas the ECFG is an intra-procedural representation of a method with summarized information from inter-procedural analyses. The ECFG also models synchronization constructs.

A call site also contains (1) a *call edge* from the call node to each callee node, labeled with the type that causes such a binding, (2) a *return edge* from each callee node to the return node, and (3) an exception return edge, labelled "exception", from a callee node to a corresponding catch node, a finally node, or the method exit node if an execption can be thrown from inside the method call. Note that if the call is static (i.e., not virtual), there is only one callee node.

To illustrate, consider method $UnavailBookFinder.main$ in P (Figure 3). The ECFG for $UnavailBookFinder.main$ (Figure 5(a)), contains two callee nodes (12 and 13) for the call to $res.addAll$ because res's dynamic type can be either $HashSet$ or $EmptySet$. (The dynamic type of res is the same as the return type of method $Library.getCheckedOutBooks$ (see Figure 2).). The callee node for dynamic type $HashSet$ corresponds to method $HashSet.addAll$ whereas the callee node for dynamic type $EmptySet$ corresponds to method $AbstractSet.addAll$ because $EmptySet$ is a subclass of $AbstractSet$ and does not override method $addAll$.

Consider now one of the two differences between P and P' in Figure 3: the addition of method $addAll$ in $EmptySet$. Such a change causes a possibly different behavior in P and P' for the call to $res.addAll$ in method $UnavailBookFinder.main$: if the dynamic type of res is $EmptySet$, the call results in an invocation of method $AbstractSet.addAll$ in P and an invocation of method $EmptySet.addAll$ in P'.

Figure 5(b) shows how the different binding, and the possibly different behavior, is reflected in the ECFG for method $UnavailBookFinder.main$: the call edge labeled $EmptySet$ from the call node for $res.addAll$ (i.e., the call edge representing the binding when res's dynamic type is $EmptySet$) is now connected to a new callee node that represents method $EmptySet.addAll$. This difference between the ECFGs for $UnavailBookFinder.main$ in P and P' lets the differencing technique determine that this call to $res.addAll$ may behave differently in P and P'. Note that a simple textual comparison would identify the addition of the method, but it would require a manual inspection of the code (or some further analysis) to identify the points in the code where such change can affect the program's behavior.

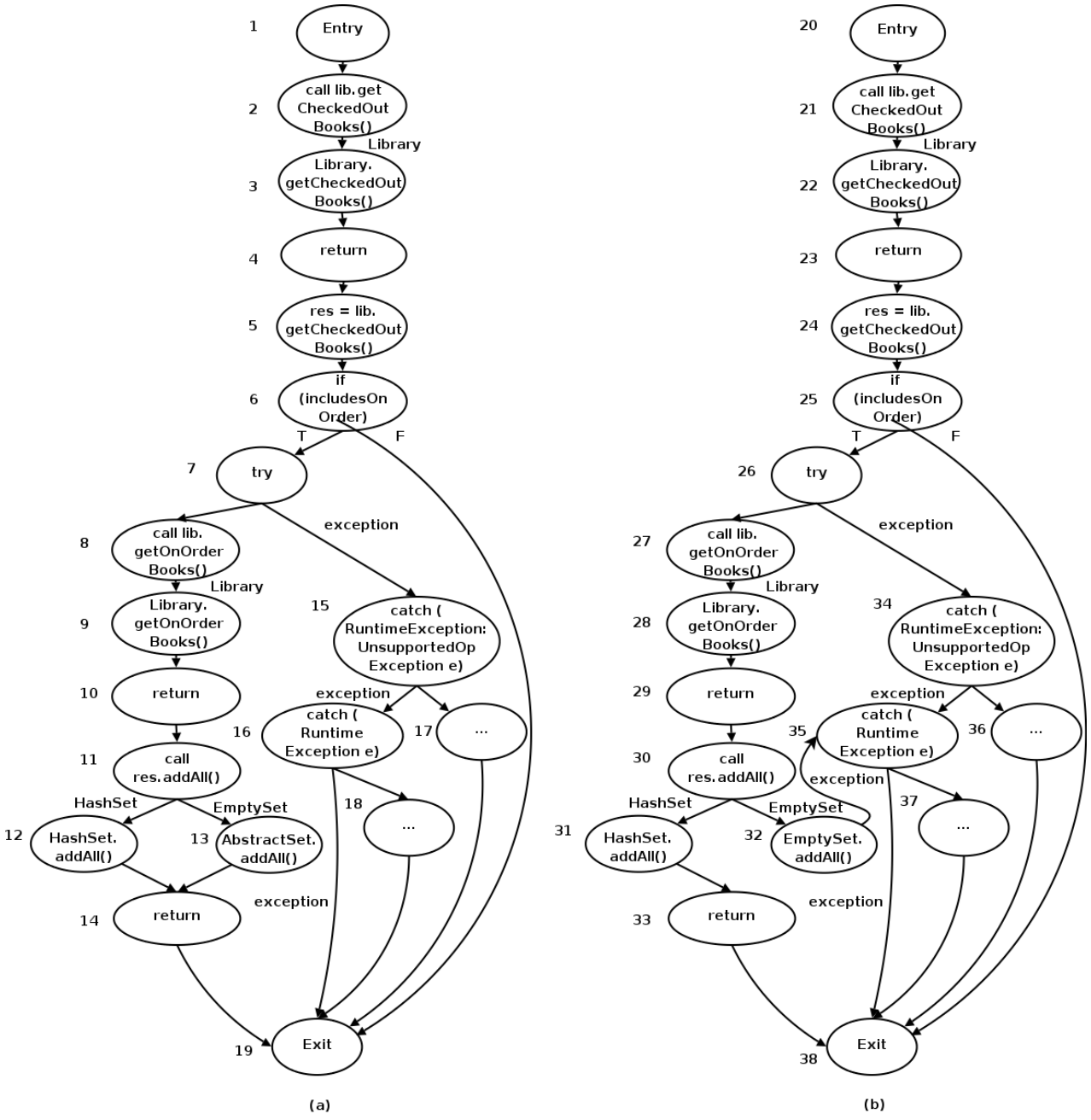


Figure 5: ECFGs for *UnavailBookFinder.main* in P and P' (Figure 3).

Variable and object types.

Changing the type of a variable may lead to changes in program behavior (e.g., changing a *long* to an *int*). To identify these kinds of changes, the ECFG augments the names of scalar variables with type information. For example, a variable a of type *double* is identified as *a_double*. This approach for representing scalar variables reflects any change in the type of a variable in the locations where that variable is referenced.

Another change that may lead to subtle changes in program behavior is the modification of class and interface hierarchies (e.g., moving a class from one hierarchy to another, changing the class that it extends). Hereafter, "class hierarchies" is used to represent both class and interface hierarchies, unless otherwise stated. Effects of these changes that result in different bindings in P and P' are captured by the new method-call representation. Other effects, however, must be specifically addressed. To this end, instead of explicitly representing class hierarchies, the hierarchy information is encoded by using globally-qualified names at points where a class is used as an argument to operator $instanceof$, as an argument to operator *cast*, as a type of a newly created exception, and as the declared type of a catch block. A *globally-qualified name* for a class contains the entire inheritance chain from the root of the inheritance tree (i.e., from class *java.lang.Object*) to its actual type.[2] A *globally-qualified name* for an interface contains all the super interfaces in alphabetical order. This method reflects changes in class hierarchies in the locations where the change may affect the program behavior. For example, the globally-qualified name for exception class $MyUnsupportedOpException$ in P is $java.lang.Throwable : java.lang.Exception : java.lang.RuntimeException : UnsupportedOpException : MyUnsupportedOpException$, whereas it is $java.lang.Throwable : java.lang.Exception : java.lang.RuntimeException : MyUnsupportedOpException$ in P'.

Exception Handling.

As for dynamic binding, program modifications involving exception-handling constructs can cause subtle side effects in parts of the code that have no obvious relation to the

[2]For efficiency, class *Object* is excluded from the name, except that for class *Object* itself.

modifications. For example, a modification of an exception type or a catch block can cause a previously caught exception to go uncaught in the modified program, thus changing the flow of control in unforeseen ways.

To identify these changes in the program, ECFG models exception-handling constructs in Java code explicitly using an approach similar to that used in [32]. The ECFG representation models each try statement with a *try* node and an edge between the try node and the node that represents the first statement in the try block.

The representation then models each catch block of the try statement with a *catch* node and a CFG to represent. Each catch node is labeled with the type of the exception that is caught by the corresponding catch block. An edge connects the catch node to the entry of the CFG enclosing the catch block.

An edge, labeled "exception," connects the try node to the catch node for the first catch block of the try statement. That edge represents all control paths from the entry node of the try block along which an exception can be propagated to the try statement. An edge labeled "exception" connects also the catch node for a catch block b_i to the catch node for catch block b_{i+1} that follows catch block b_i (if any). This edge represents all control paths from the entry node of the try block along which an exception is (1) raised, (2) propagated to the try statement, and (3) not handled by any of the catch blocks that precede catch block b_{i+1}.

This representation models finally blocks by creating a CFG for each finally block, delimited by *finally entry* and *finally exit* nodes. An edge connects the last node in the corresponding try block to the finally entry node. The representation also contains one edge from the last node of each catch block related to the finally to the finally entry node. If there are exceptions that cannot be caught by any catch block of the try statement and there is at least one catch block, an edge connects the catch node for the last catch block to the finally entry node.

Because the information used in building the exception-related part of the ECFG is computed through inter-procedural exception analysis [69], the ECFG can represent both intra- and inter-procedural exception flow. If an exception is thrown in a try block for a

method m, the node that represents the throw statement is connected to (1) the catch block in m that would catch the exception, if such a catch block exists, (2) the finally entry node, if no catch block can catch the exception and there is a finally block for the considered try block, or (3) the exit node of m's ECFG, otherwise. Conversely, if an exception is thrown in method m from outside a try block, the node that represents the throw statement is always connected to the exit node of m's ECFG.

For example, consider again method $UnavailBookFinder.main$ in P (Figure 3) and its ECFG (Figure 5(a)). The ECFG contains a try node for the try block (node 7) and catch nodes for the two catch blocks associated with the try block (nodes 15 and 16). The catch nodes are connected to the entry nodes of the CFGs that represent the corresponding catch blocks (nodes 17 and 18).

Consider now the second difference between P and P': the modification in the type hierarchy that involves class $MyUnsupportedOpException$. This class is a direct subclass of $UnsupportedOpException$ in P and a direct subclass of $RuntimeException$ in P'. Such a change causes a possibly different behavior in P and P' because, in P', an exception may be thrown in method $EmptySet.addAll$, propagate back to method $UnavailBookFinder.main$, and be caught by the catch block that catches exceptions of type $RuntimeException$.

Figure 5(b) shows how the possibly different behavior is reflected in the new representation: the node that represents the call to method $EmptySet.addAll$ (node32) is connected to the catch node that catches exceptions of type $RuntimeException$ (node 35) rather than to the return node (node 33). These differences between the two ECFGs let the differencing technique determine that, if the exception in method $EmptySet.addAll$ is thrown, method $UnavailBookFinder.main$ in P and P' may behave differently. A simple textual comparison would identify only the change in the type of $E3$, whereas identifying the side effects of such a change would require further analysis.

Synchronization.

Java provides explicit support for threading and concurrency through the `synchronized` construct. Using such a construct, Java programmers can enforce mutual exclusion semaphores (mutexes) or define critical sections (i.e., atomic blocks of code). Synchronized areas of code can be declared at the block, method, and class levels.

The ECFG models a synchronized area of code by creating two special nodes: synchronize start and synchronize end. A *synchronize-start* node is added before the node that represents the first statement of a synchronized area of code. Analogously, a *synchronize-end* node is added after the node that represents the last statement of a synchronized area of code.

In a program that uses synchronized constructs, changes in behavior can occur because (1) an area of code that was not synchronized becomes synchronized, (2) an area of code that was synchronized is no longer synchronized, or (3) a synchronized area is expanded or contracted. In the ECFG, these cases are suitably captured by addition, removal, or replacement of synchronize-start and synchronize-end nodes.

Reflection.

In Java, reflection provides runtime access to information about classes' fields and methods, and allows for using such fields and methods to operate on objects. In the presence of reflection, this representation can fail to capture some of the behaviors of the program. For example, using reflection, a method may be invoked on an object without performing a traditional method call on that object. For another example, a program may contain a predicate whose truth value depends on the number of fields in a given class; in such a case, the control flow in the program may be affected by the (apparently harmless) addition of unused fields to that class. Although some uses of reflection can be handled through analysis, others require additional, user-provided information. This research assumes that such information is available and can be leveraged for the analysis. In particular, for dynamic class loading, this research assumes that the classes that can be loaded (and instantiated) by name at a specific program point either can be inferred from the code (e.g.,

when a `cast` operator is used on the instance after the object creation) or are specified by the user. It is worth noting that, for the subjects used in the empirical studies (see Section 3.4), such external information is provided for all uses of reflection that could not be handled automatically by this technique.

Hammocks

`CalcDiff` uses hammocks and hammock graphs as a way to impose a hierarchical structure on the ECFGs, which facilitates the matching.

Definition 4. if G is a graph, a *hammock* H is an induced subgraph of G with a distinguished node V in H called the *entry node* and a distinguished node W not in H called the *exit node*, such that

1. All edges from (G - H) to H go to V.

2. All edges from H to (G - H) go to W.

Similar to Laski and Szermer's approach [42], once a hammock is identified, the algorithm reduces it to a *hammock node* in three steps. First, the nodes in the hammock are replaced by a new node. Second, all incoming edges to the hammock are redirected to the new node. Third, all edges leaving the hammock are replaced by an edge from the new node to the hammock exit. The resulting graph at each intermediate step is called a *hammock graph*.

Figure 6 illustrates how the ECFG for method $UnavailBookFinder.main$ in P is transformed into a single hammock node and the intermediate hammocks that are generated during the transformation. The regions inside the dotted lines in Figure 6(a) (i.e., nodes 11 to 13, and nodes 15 to 18) represent the two hammocks that the algorithm identifies and replaces with hammock nodes 11' and 15', respectively (Figure 6(b)). Then the region inside the dotted lines in Figure 6(b) (i.e., nodes 7 to 10, 11', 14, and 15') represent another hammock that the algorithm reduces to hammock node 7' (Figure 6(c)). This process continues until the graph is reduced to an intermediate graph with no hammocks (Figure 6(d). This research uses Ferrante et al.'s algorithm for identifying hammocks [24].

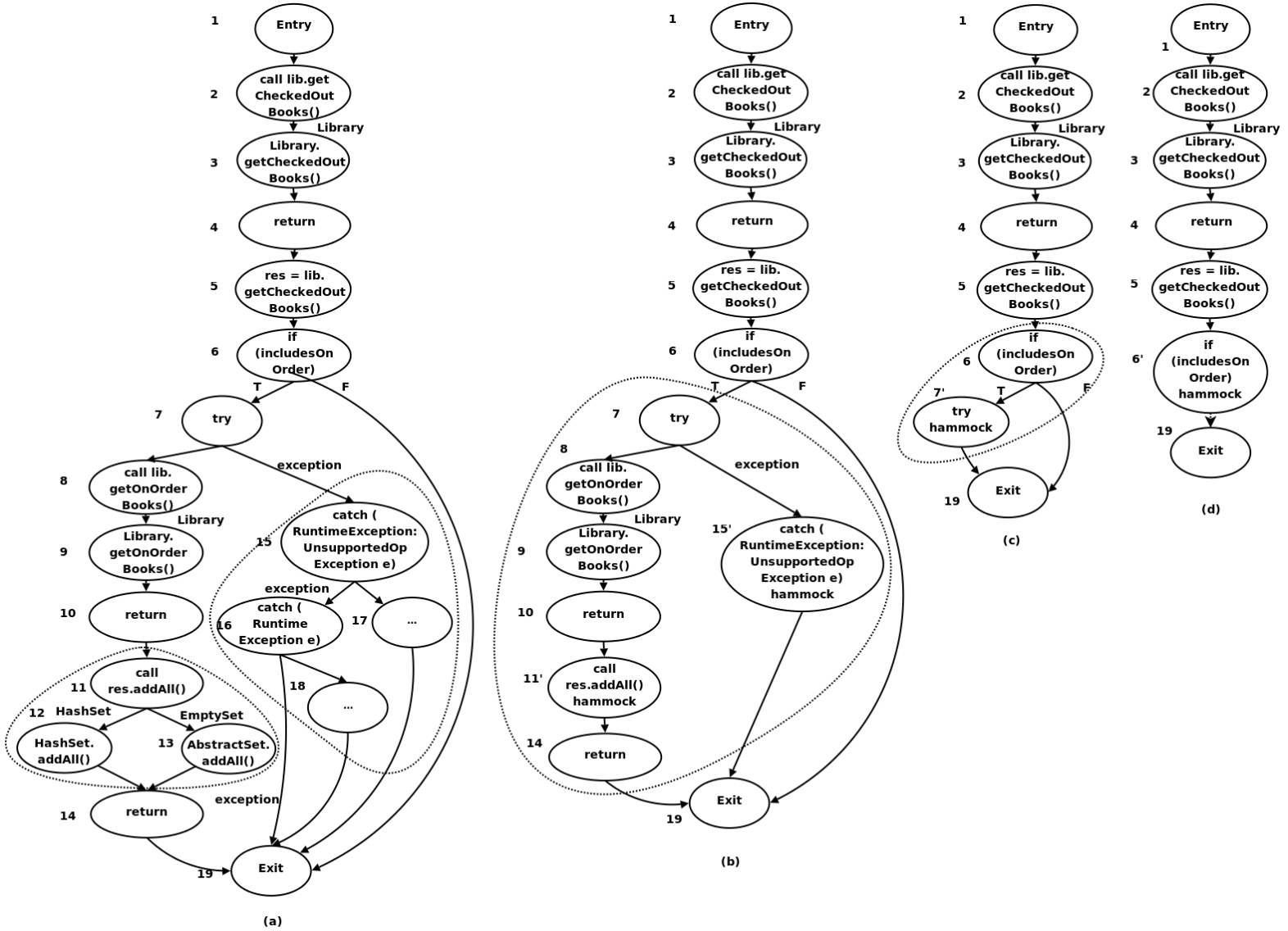


Figure 6: ECFG for *UnavailBookFinder.main* (a) and intermediate hammock graphs for *UnavailBookFinder.main* (b),(c), and(d).

A hammock H with start node s is minimal if there is no hammock H' that (1) has the same start node s, and (2) contains a smaller number of nodes. Hereafter, a hammock is used to refer to a minimal hammock, unless otherwise stated.

3.2.4.2 Hammock Matching Algorithm

The hammock matching algorithm, `HmMatch`, is given in Figure 7. The algorithm is based on Laski and Szermer's algorithm for transforming two graphs into their respective isomorphic graphs [42]. `HmMatch` takes as input G and G', two hammock graphs, LH, an integer indicating the maximum lookahead, and S, a threshold for deciding whether two hammocks are similar enough to be considered a match. The algorithm outputs N, a set of pairs whose first element is, in turn, a pair of matching nodes, and whose second element is a label that indicates whether the two nodes are "unchanged" or "modified."

To increase the number of matches, `HmMatch` extends Laski and Szermer's algorithm by allowing for the matching of hammocks at different nesting levels. This modification accounts for some common changes that were encountered in preliminary studies, such as the addition of a loop or a conditional statement at the beginning of a code segment. The rest of this section first describes algorithm `HmMatch` and then presents an example of use of the algorithm on the code in Figure 3.

`HmMatch` starts matching nodes in the two graphs by performing a depth-first pairwise traversal of G and G', starting from their start nodes. Thus, at line 1, the pair of start nodes is added to stack ST, which the algorithm uses as a worklist. Each iteration over the main *while* loop (lines 2–26) extracts one node pair from the stack and checks whether the two nodes match. The body of the loop first checks whether any node in the current pair is already matched (line 4). A matched node that has already been visited must not be considered again; in this case, the algorithm continues by considering the next pair in the worklist (line 5).

To compare two nodes, `HmMatch` invokes $comp(c, c', S, N)$ (line 9), where c and c' are the two nodes to compare, S is the similarity threshold for matching hammocks, and N is the set of matching nodes. Unless c and c' are hammock nodes, *comp* returns *true* if the two nodes'

procedure `HmMatch`
Input: hammock graph in original version G,
hammock graph in modified version G'
maximum lookahead LH
hammock similarity threshold S
Output: set of pair $\langle\langle$node,node$\rangle$,label$\rangle$ N
Use: $succs(A)$ returns set of successors of each node a in A
$comp(m, n, S, N)$ returns true if m and n are matched
$edgeMatching(n, n')$ returns matched outgoing edge pairs
Declare: stack of $\langle$node,node$\rangle$ ST
current depth d
current nodes c and c'
lookahead node sets L and L'
pair $\langle$node,node$\rangle$ $match$

Begin: `HmMatch`
push start node pair $\langle s, s' \rangle$ onto ST
while ST is not empty **do**
  pop $\langle c, c' \rangle$ from ST
  if c or c' is already matched **then**
    continue
  end if
  if $comp(c, c', S, N)$ **then**
    $match = \langle c, c' \rangle$
  else
    $match = null$; $L = \{c\}$; $L' = \{c'\}$
    for $(d = 0; d < LH; d++)$ **do**
      $L = succs(L)$; $L' = succs(L')$
      if $\bigvee_{p' \in L'} comp(c, p', S, N) \vee \bigvee_{p \in L} comp(c', p, S, N)$ **then**
        set $match$ to the first pair that matches
        break
      end if
    end for
  end if
  if $match$!= null **then**
    push $\langle match,$ "$unchanged$"$\rangle$ onto N
    set c and c' to the two nodes in $match$
  else
    push $\langle\langle c, c' \rangle,$ "$modified$"$\rangle$ onto N
  end if
  push a pair of sink nodes for each edge pair returned from $edgeMatching(c, c')$ onto ST
end while
end `HmMatch`

Figure 7: Hammock matching algorithm.

labels are the same. If c and c' are hammock nodes, *comp* (1) expands the hammock nodes into two graphs, (2) adds dummy exit nodes to both graphs, (3) recursively calls `HmMatch` to obtain the set of matched and modified node pairs, and (4) computes the ratio of unchanged-matched node pairs in the set to the number of nodes in the smaller hammock. If the ratio is greater than threshold S, *comp* returns $true$ (i.e., the two hammocks are matched) and pushes all pairs in the set returned by `HmMatch` onto N. Otherwise, *comp* returns $false$.

If two nodes c and c' are matched (i.e., *comp* returns $true$), they are stored in variable *match* as a pair (line 8) and later added to the set of matched nodes with label "unchanged" (line 20). Otherwise, `HmMatch` tries to find a match for c (resp., c') by examining c''s (resp., c's) descendants up to the specified maximum lookahead (lines 10–17). First, *match* is initialized to null, and the lookahead sets L and L' are initialized to contain only the current nodes (line 10). The algorithm then executes the for loop until a match is found or depth d reaches the maximum lookahead LH (lines 11–17). At each iteration, the algorithm updates L and L' to the sets of successors of their members, obtained by calling procedure *succs* (line 12). $succs(L)$ returns, for each node l in L and each outgoing edge from l, the sink of such edge. If node l is a hammock node, *succs* returns a set that consists of the start node and the exit node of the hammock. In this way, a match can occur between nodes in hammocks at different nesting levels. After computing the lookahead sets L and L', the algorithm compares each node in set L' with c and each node in set L with c' (line 13). If there is a match, the search stops, and the first matching pair found is stored in variable *match* (lines 14–15). The matching pair is then added to the set of matched nodes with label "unchanged" (line 20). After two nodes have been matched as unchanged, c and c' are set to be the two nodes in the matching pair (line 21). If no matching is found, even after the lookahead, c and c' are added to the set of matched nodes with label "modified."

After processing nodes c and c', the algorithm matches the outgoing edges from the two nodes by calling $edgeMatching(c, c')$. $edgeMatching$ matches outgoing edges from c and c' based on their labels. For each pair of matching edges, the corresponding sink nodes are pushed onto worklist ST (line 27). At this point, the algorithm continues iterating over the main $while$ loop until ST is empty.

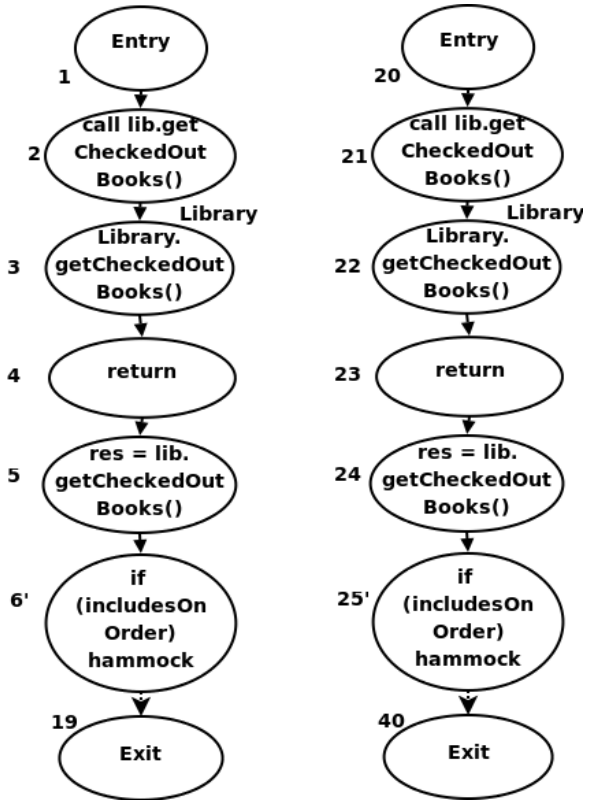


Figure 8: Hammock graphs for the original and modified versions of *UnavailBookFinder.main*.

When the algorithm terminates, all nodes in the old version that are not in any pair (i.e., that have not been matched to any other node) are considered deleted nodes. Similarly, all nodes in the new version that are not in any pair are considered added nodes.

To illustrate `HmMatch` better, consider a partial run of `CalcDiff` on the example code in Figure 3. In particular, consider the execution from the point at which the pair of methods *UnavailBookFinder.main* in P and P' is compared (line 5). At line 6 of `CalcDiff`, the ECFGs for the methods are created, and at lines 7 and 8 of the algorithm, hammocks in the ECFGs are identified and reduced to intermediate graphs with no hammocks. Then, at line 9, `CalcDiff` calls `HmMatch`, passing it the two graphs. For the example, assume that the lookahead threshold (LH) is 1, and that the hammock similarity threshold (S) is 0.5.

Figure 8 shows the graphs for the original and modified versions of the program. `HmMatch` pushes the pair of start nodes ⟨1,20⟩ onto stack ST (line 1).

In the first iteration over the main *while* loop, the algorithm extracts node pair ⟨1,20⟩

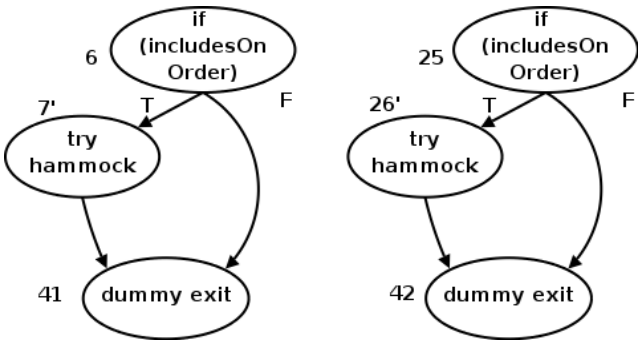


Figure 9: Hammock graphs for hammock nodes 6' and 25' in Figure 8.

from ST (line 3). Because neither node is already matched, the algorithm compares the two nodes by calling $comp(1, 20, 0.5, N)$ (line 7), which compares the nodes' labels and returns *true*. Therefore, the algorithm sets *match* to this pair (line 8), adds the pair of nodes to N with label "unchanged," and sets c and c' to be nodes 1 and 20 (lines 20-22), which in this case leaves c and c' unchanged. At this point, the outgoing edges from 1 and 20 are matched by calling $edgeMatching(1, 20)$. Each node in the entry pair has only one outgoing edge, and the two edges match, so the pair of sink nodes $\langle 2, 21\rangle$ is pushed onto the worklist.

The next four iterations over the main loop perform the same steps as the first iteration to match pairs $\langle 2, 21\rangle$, $\langle 3, 22\rangle$, $\langle 4, 23\rangle$, and $\langle 5, 24\rangle$ and add these pairs to N with labels "unchanged." In the fifth iteration, the algorithm extracts node pair $\langle 6', 25'\rangle$ from ST. Because nodes 6' and 25' are not already matched and are both hammock nodes, *comp* (line 7) expands nodes $6'$ and $25'$ to hammock graphs $G_{6'}$ and $G_{25'}$ (shown in Figure 9, respectively, and calls `HmMatch(`$G_{6'}$`,`$G_{25'}$`,1,0.5)`. `HmMatch` then pushes the pair of nodes $\langle 6, 25\rangle$ onto ST_1.[3] This pair is then extracted from the stack and compared (lines 3–7). Because both nodes have the same label, they are matched, and the pair is added to N_1 with label "unchanged" (lines 20–21). *edgeMatching* is then called on the two nodes in the pair, 6 and 25; *edgeMatching* matches like-labeled edges and the two pairs of sink nodes $\langle$41,42$\rangle$ and $\langle$7',26'$\rangle$ are pushed onto ST_1.

In the next iteration over the main loop, the nodes in pair $\langle$41,42$\rangle$ are compared. Because both of them are already matched, the algorithm continues to the next pair on the stack

[3]The subscript notation is used to distinguish variables in recursively called procedures.

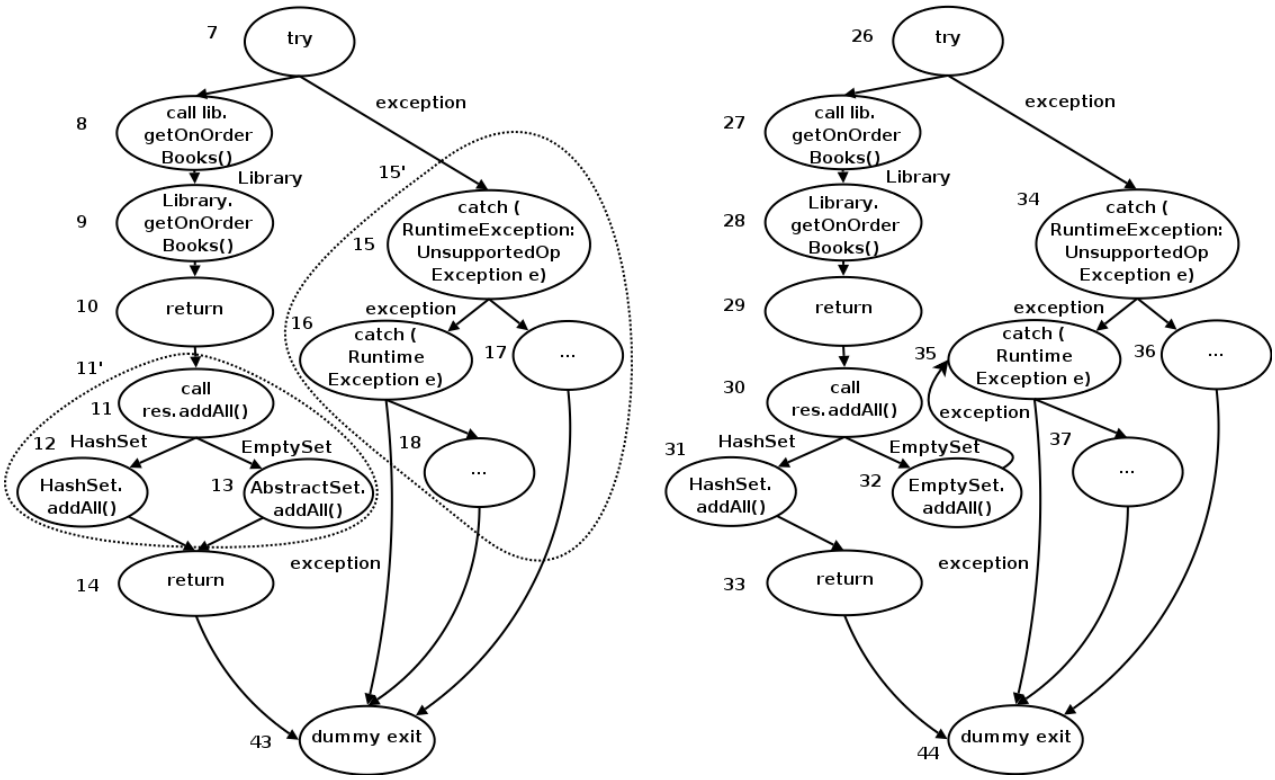


Figure 10: Hammock graphs for hammock nodes 7' and 26' in Figure 9.

(line 5). The next iteration extracts pair ⟨7',26'⟩ from ST_1. Because nodes 7' and 26' are not already matched and are both hammock nodes, *comp* expands nodes $7'$ and $26'$ to get two graphs $G_{7'}$ and $G_{26'}$ (shown in Figure 10), respectively, and calls `HmMatch(`$G_{7'}$`,`$G_{26'}$`,1.0.5)`. Figure 10 shows nodes in the two hammocks inside the hammock graph of the original version to facilitate the discussion of node matching at different hammock levels. `HmMatch` then matches the dummy exit nodes 43 and 44, and puts ⟨7,26⟩ onto ST_2. This pair is then extracted and compared. Because both nodes in the pair have the same label, they are matched and the pair is pushed onto N_2 with label "unchanged." Then the outgoing edges of both nodes are matched by $edgeMatching(7, 26)$, and the pairs ⟨15',34⟩ and ⟨8,27⟩ are pushed onto ST_2. In the next iteration of the main loop, the pair ⟨15',34⟩ is extracted and compared. Because both nodes are not already matched, the algorithm calls $comp(15', 34, 0.5, N_2)$, which returns $false$ because node 15' is a hammock node, but

node 34 is not. The algorithm performs a lookahead to find a match for either node 15' or node 34 (lines 10-17). Node 15' is compared with the successors of nodes 34, which are nodes 35 and 36, whereas node 34 is compared with the successors of nodes 15', which are nodes 15 and 43 (the entry and exit nodes of the hammock). Nodes 34 and 15 are matched, the algorithm, thus, sets *match* to the pair ⟨15,34⟩ (line 14), breaks out of the loop (line 15), and pushes the pair onto N_2. `HmMatch` continues comparing and matching the rest of the graph in an analogous way. When all nodes in the two hammocks in Figure 10 are compared, `HmMatch` returns to the calling procedure *comp*. Because 24 out of 28 nodes are unchanged-matched (nodes 11' and 15' have no match, and nodes 13 and 32 are modified-matched), and the similarity threshold is 0.5, *comp* classifies the two hammocks as matched. Therefore, the pairs in N_2 are added to N_1, *comp* returns *true*, pair ⟨7',26'⟩ is added to N_1 with label "unchanged," and pair ⟨41,42⟩ is pushed onto ST_1. The rest of the graphs are compared and matched analogously. The part of the example shown so far illustrates the main parts of the algorithm including the matching of hammock nodes, the lookahead, and the matching of nodes at different hammock levels.

3.2.5 Worst-Case Time Complexity

The dominating cost of `CalcDiff` is the matching at the node level. Let m and n be the number of nodes in all matched methods in the original and modified versions of a program, respectively. Let p be the maximum number of nodes in a method, and let the maximum lookahead be greater than p. In the worst case, if no matching of hammocks at different nesting levels occurs, the algorithm compares each node in a method with all nodes in the matching method (at most p), leading to a worst-case complexity of $O(p \cdot min(m, n))$. If matching of hammocks at different nesting levels occurs, the algorithm may compare a pair of nodes more than once. To decide whether two hammocks are matched, $HmMatch$ compares each node in one hammock with nodes in the other hammock and counts the number of matched nodes. If lookahead is performed, the same pairs of nodes are compared again in the context of the new pair of hammocks. The number of times the same pairs of nodes are compared depends on the maximum nesting depth of hammocks and the

maximum lookahead. In the worst case, the maximum nesting depth is $O(p)$. Let the maximum lookahead be greater than p. The worst-case complexity of this algorithm is then $O(p^2 \cdot min(m, n))$.

3.3 JDiff: A differencing tool

To evaluate this differencing algorithm, we developed JDIFF, a prototype tool for Java programs. The tool consists of two main components: a differencing tool and a graph-building tool. The differencing tool inputs the original and modified versions of a program, compares them, and outputs sets of pairs of matching classes, methods, and nodes. The differencing tool calls the graph-building tool to build an ECFG for each method. To build ECFGs, the tool leverages the capabilities of JABA (Java Architecture for Bytecode Analysis)[4], a Java-analysis front-end.

3.4 Empirical Studies on Differencing Algorithm

To evaluate the differencing algorithm `CalcDiff`, we performed three studies on two real, medium-sized programs to investigate the following research questions;

RQ1: How often do changes involving object-oriented features occur in practice?

RQ2: How long does technique `CalcDiff` take to compute change information with varying values of lookahead and hammock-similarity threshold?

RQ3: How much does algorithm `CalcDiff` gain, in terms of numbers of matched nodes, compared to Laski and Szermer's algorithm?

This section describes experimental setup, presents the studies, and discusses the results.

3.4.1 Experimental Setup

These studies used two subjects: DAIKON and JABA. DAIKON is a tool for discovering likely program invariants developed by Ernst and colleagues [23], whereas JABA is the analysis tool described above. To evaluate the effectiveness of algorithm `CalcDiff`, we ran JDIFF on 39 pairs of consecutive versions of DAIKON ($\langle v1, v2\rangle$ to $\langle v39, v40\rangle$) and on seven pairs of

[4]http://www.cc.gatech.edu/aristotle/Tools/jaba.html

consecutive versions of JABA ($\langle v1, v2\rangle$ to $\langle v7, v8\rangle$). All versions are real versions extracted from the CVS repositories for the two subjects.

The time interval between two versions of DAIKON is one week (except for weeks in which there were no changes in the software; in these cases, the interval is extended one week at a time until the new version included modifications). The versions of JABA can be divided into two groups: v1–v4 and v5–v8. Versions v1 to v4 correspond to a period in which JABA was undergoing a major restructuring, and therefore contain a greater number of changes than versions v5 to v8. Also, whereas the time interval between consecutive versions for v1–v4 and v5–v8 is about two weeks, the time interval between v4 and v5 is about six months.

Table 1: Subject programs used in differencing studies

Program	Versions	Classes	Methods	KLOC
DAIKON	40	357-755	2878-7112	48-123
JABA	8	550	2800	60

Table 1 summarizes the characteristics of the two subject programs. The table shows, for each subject, the number of versions (*Versions*), the number of classes (*Classes*), the number of methods (*Methods*), and the number of non-comment lines of code in thousands (*KLOC*). The size of DAIKON varies widely between versions and, thus, is reported as ranges between the numbers of the first version and those of the last version. The size of JABA varies little between versions. Thus the table reports the average size of all versions.

We ran all studies on a Sun Fire v480 server equipped with four 900 MHz UltraSparc-III processors and 16 GB of memory. Each run used only one processor and a maximum heap size of 3.5 GB.

There are several threats to the validity of these studies. An external threat exists because the studies used only two subject programs. Thus, the results may not be generalizable. However, the subject programs are real programs that are used on a regular basis by several research groups, and the changes considered are real changes that include bug fixes and feature enhancements. Another external threat to validity exists for the fourth study: this study used only one test suite for each subject. Different test suites may generate

different results.

Threats to internal validity mostly concern possible errors in the algorithm implementations and measurement tools that could affect outcomes. To control for these threats, we validated the implementations on known examples and performed several sanity checks.

3.4.2 Study 1: Object-Oriented Changes

One of the main advantages of using technique `CalcDiff`, instead of traditional text-based differencing approaches, is that `CalcDiff` can account for the effects of changes due to object-oriented features. (For simplicity, in the following, such changes are referred to as *object-oriented changes*.) The goal of Study 1 is to assess the usefulness of `CalcDiff` by investigating how often object-oriented changes occur in practice. This study used the 39 pairs of consecutive versions of DAIKON and the seven pairs of consecutive versions of JABA. We first ran `JDiff` on each pair of versions $\langle P_i, P_{i+1} \rangle$, where $1 \leq i \leq 39$ for DAIKON and $1 \leq i \leq 7$ for JABA, then analyzed `JDiff`'s output, and determined whether each change is an object-oriented change.

Table 2 presents the results of this study. The upper part shows the number of object-oriented changes in DAIKON, and the lower part shows the number of object-oriented changes for JABA. The table presents, in separate columns and for each pair of versions, the number of occurrences of two types of object-oriented changes: changes in dynamic binding (Binding) and changes in types of local variables and fields (Type). (The other two types of object-oriented changes discussed in Section 3.2.4.1 occur rarely in the subject programs and versions: there are only three changes related to synchronized blocks and no changes in the exception class hierarchy.) A pair of columns under each type show the number of actual (i.e., syntactic) changes (AC) and the number of statements indirectly affected by such changes (IA).

For changes in dynamic binding, the actual changes are additions or deletions of methods, changes in methods from non-static to static or vice versa, and changes of method access modifiers. The statements indirectly affected by such changes are the method calls that may be bound to different methods as a consequence of the change. For changes in

Table 2: Number of actual changes (AC) and number of statements indirectly affected (IA) for each kind of object-oriented changes in each pair of versions of DAIKON and JABA

DAIKON

Pair	Binding		Type	
	AC	IA	AC	IA
$v1$,$v2$	26	151	14	31
$v2$,$v3$				
$v3$,$v4$			17	54
$v4$,$v5$			1	2
$v5$,$v6$			2	8
$v6$,$v7$	4	4	19	34
$v7$,$v8$	3	48		
$v8$,$v9$	5	14		
$v9$,$v10$	2	2		
$v10$,$v11$	8	14		
$v11$,$v12$	4	5	6	53
$v12$,$v13$	2	3	4	10
$v13$,$v14$				
$v14$,$v15$	3	4		
$v15$,$v16$			1	1
$v16$,$v17$				
$v17$,$v18$	6	8		
$v18$,$v19$	21	90		
$v19$,$v20$	5	6		
$v20$,$v21$	17	19		

Pair	Binding		Type	
	AC	IA	AC	IA
$v21$,$v22$	11	15		
$v22$,$v23$	29	90	3	4
$v23$,$v24$	7	45		
$v24$,$v25$	13	187	1	1
$v25$,$v26$	38	230	2	6
$v26$,$v27$				
$v27$,$v28$				
$v28$,$v29$				
$v29$,$v30$				
$v30$,$v31$				
$v31$,$v32$				
$v32$,$v33$				
$v33$,$v34$	19	130	10	16
$v34$,$v35$	8	14		
$v35$,$v36$				
$v36$,$v37$	6	14		
$v37$,$v38$	7	54		
$v38$,$v39$				
$v39$,$v40$	14	22	10	30

JABA

Pair	Binding		Type	
	AC	IA	AC	IA
$v1$,$v2$	2	2		
$v2$,$v3$	1	1		
$v3$,$v4$	7	12	1	1
$v4$,$v5$	12	313	73	289

Pair	Binding		Type	
	AC	IA	AC	IA
$v5$,$v6$	14	21		
$v6$,$v7$				
$v7$,$v8$			1	4

types, the actual changes consist of changes in declarations of local variables and fields, and the affected statements are the statements in which such variables and fields are referenced. For example, the changes for pair $\langle v1, v2\rangle$ of DAIKON consist of modifications to 26 methods, which may cause 151 method calls to behave differently, and modifications to the type of 14 local variables and fields, which may affect the behavior of 31 statements.

The indirectly-affected statements represent the additional information provided by `CalcDiff` that other differencing approaches, such as `diff`, could not identify (without further analysis). The results of this study show that object-oriented changes occur in more than half of the cases considered. In a few cases, object-oriented changes may result in more than 100 indirectly-affected statements (i.e., statements in the code that may behave

differently but would not be identified by traditional differencing approaches).

3.4.3 Study 2: Efficiency

The goal of Study 2 is to measure the efficiency of JDiff for various values of lookahead, LH, and hammock similarity threshold, S. This study used as subjects of the study all versions of Jaba and Daikon used in the previous study. For each pair of versions, we ran JDiff with different values for LH and S, and measured the running times. Because the goal of the study is to assess the efficiency of technique `CalcDiff`, the running times reported do not include the time required by Jaba to perform the underlying control- and data-flow analysis. (This time ranges from 150 to 586 seconds for Daikon and from 279 to 495 for Jaba.)

Figures 11 and 12 show the running time (in seconds) of JDiff. For Daikon, because there are 39 pairs of versions, the running times of JDiff are reported using box-and-whisker plots. Figures 11(a), 11(b), and 11(c) show box-and-whisker plots of the running times obtained for thresholds of 0.0, 0.6, and 1.0, respectively. In the diagrams, the horizontal axis represents the value of LH, and the vertical axis represents the running time in second. In a box-and-whisker plot, the upper and lower ends of the box are the upper and lower quartiles, respectively; the median is marked by the horizontal line inside the box; the whiskers are the two lines outside the box that extends to the highest and lowest observations that are not outliers; and the outliers are marked by star symbols above and below the whiskers. To illustrate, consider the runs of JDiff with $LH = 10$ and $S = 1.0$ (see the second box-and-whisker plot in Figure 11(c)), the plot divides the 39 runs (one per pair of versions) into four groups of about 10 runs each. The first group, represented by the whisker below the box, shows that the 10 runs in this group took between 90 and 120 seconds to complete. The second group, represented by the lower half of the box, shows that the runs in this group took between 120 and 135 seconds. The line inside the box shows the median, which is about 135 seconds. The third group, represented by the upper half of the box, shows that the running time for the 10 runs in this group ranges between 135 and 190 seconds. Finally, the last group, represented by the whisker above the box, shows that the running time for

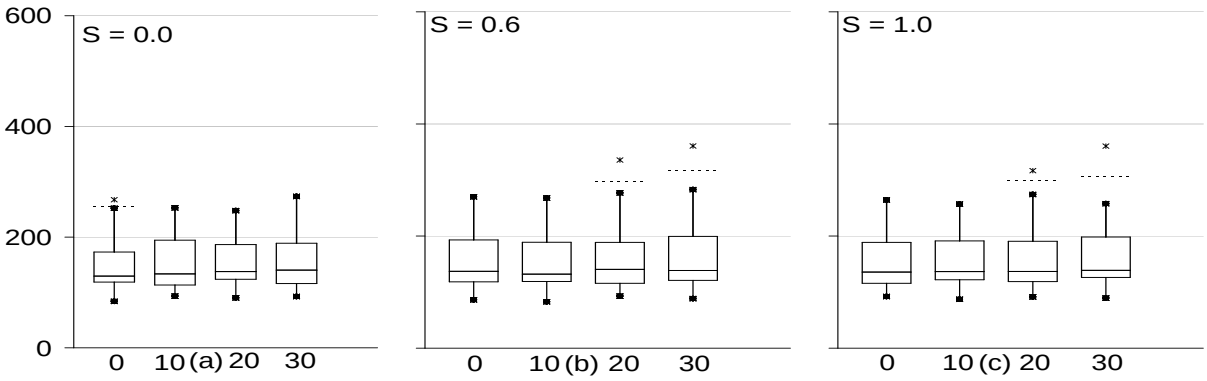


Figure 11: Average time (sec) for various pairs of versions of DAIKON , lookaheads, and similarity thresholds.

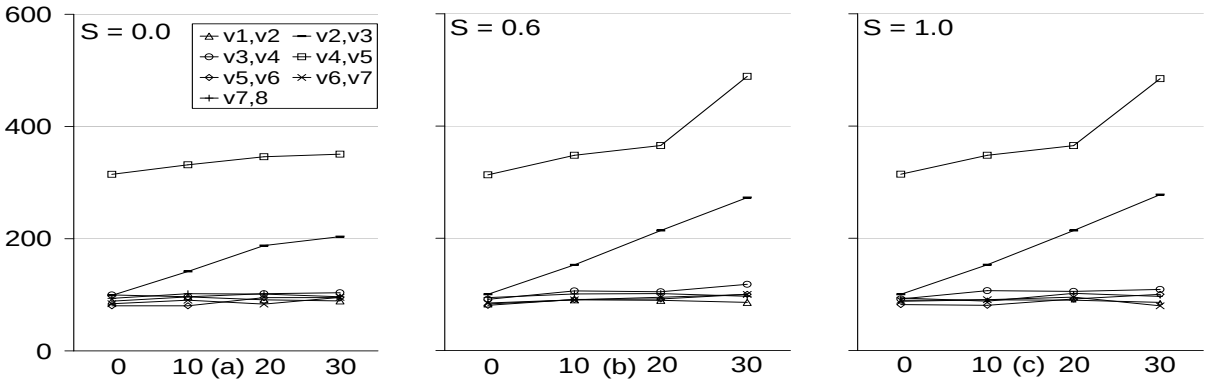


Figure 12: Average time (sec) for various pairs of versions of JABA, lookaheads, and similarity thresholds.

the 10 runs in this group ranges between 190 and 250 seconds. For JABA, there are only seven pairs of versions; therefore, Figures 12(a), 12(b), and 12(c) show the running time of JDIFF on each individual pair for similarity thresholds of 0.0, 0.6, and 1.0, respectively. For example, JDIFF took about 200 seconds to compute the differences between versions v2 and v3 of JABA with $S = 0$ and $LH = 30$ (see Figure 12(a)).

The results show that, when LH is kept constant, the value of S affects only slightly the performance of JDIFF. Intuitively, with $S = 0$, the algorithm matches a hammock in the original program's ECFG with the first hammock found in the modified version's ECFG. Thus, each hammock is compared at most once, and the running time is almost the same regardless of the value of LH. In addition, because each hammock is compared at

most once, the running time for these cases is less than for $S = 0.6$ and $S = 1.0$, where a hammock may be compared more than once. For $S = 0.6$ and $S = 1.0$, the number of times a hammock is compared depends on the lookahead and on the actual changes. As shown in the results, only in this case does the time increase when the lookahead increases. In all cases considered, JDiff took less than six minutes to compute the differences between a pair of versions (less than 15 minutes when Jaba's analysis time is included). Note that JDiff is still a prototype, so we expect even better performance on an optimized version of the tool.

3.4.4 Study 3: Effectiveness

The goal of Study 3 is to evaluate the effectiveness of algorithm `CalcDiff` compared to Laski and Szermer's algorithm [42]. To this end, we compared the number of nodes that each algorithm matches. For the study, we implemented Laski and Szermer's algorithm (`LS`) by modifying our tool. Note that in their paper [42], the handling of some specific cases is not discussed. For example, when two hammocks have the same label, they are expanded and compared, but the algorithm behavior is undefined in the case in which the expanded graphs cannot be made isomorphic by applying node renaming, node removing, and node collapsing. Those cases are handled in the same way for both algorithms. There are three differences between the two algorithms: (1) `LS` does not use the lookahead but searches the graphs until the hammock exit node is found; (2) `LS` does not allow the matching of hammocks at different nesting levels; and (3) `LS` does not use the hammock similarity threshold but decides whether two hammocks are matched by comparing the hammocks' entry nodes only.

We ran both algorithms on all versions of Daikon and Jaba used in Studies 1 and 2 and counted the number of nodes in the sets of added, deleted, modified, and unchanged nodes. We ran `JDiff` several times, using different values of lookahead LH and similarity threshold S. Note that this study considers only nodes in modified methods because added, deleted, and unchanged methods do not show differences in matching capability between the two algorithms.

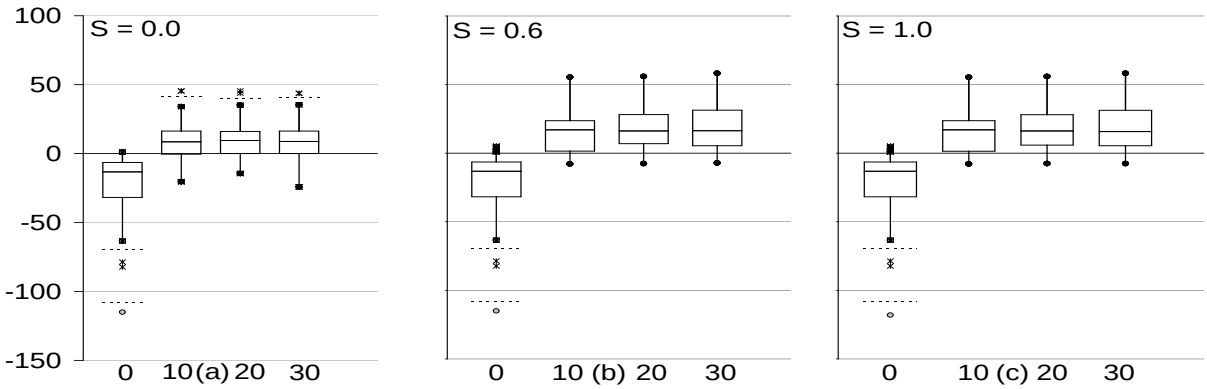


Figure 13: Percentage of the number of nodes identified as matched by JDiff but as unmatched by `LS` in Daikon.

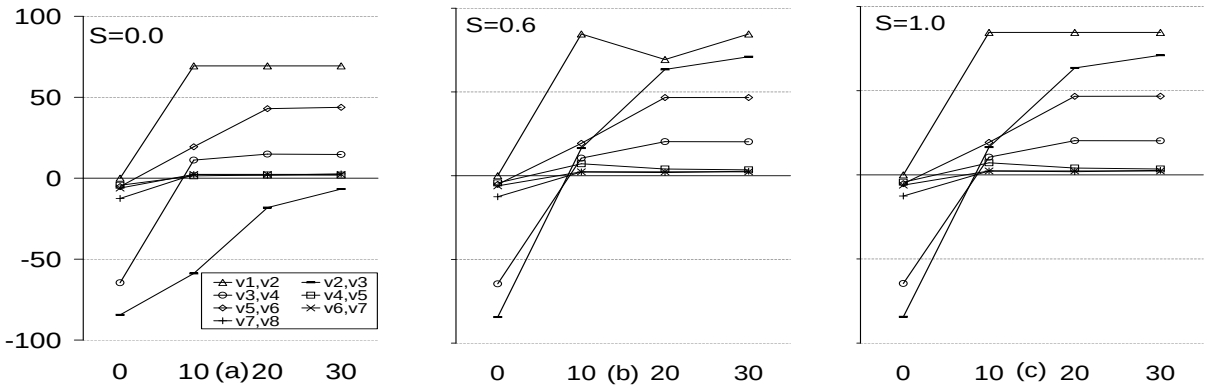


Figure 14: Percentage of the number of nodes identified as matched by JDiff but as unmatched by `LS` in Jaba.

To compare the effectiveness of the two algorithms, we compute the number of nodes that are matched by JDiff but that are not matched by `LS`. This number is reported in terms of a percentage over the total number of nodes identified as unmatched by `LS`. Intuitively, this percentage represents the relative improvement, in terms of matching, achieved by `CalcDiff` over `LS`. The number of nodes identified as unmatched by `LS` varies between 745 and 46,745 for Daikon and between 116 and 40,024 for Jaba.

Figures 13 and 14 present the results of this study. Figures 13(a), 13(b), and 13(c) present the results for Daikon, summarized across all 39 pairs of versions, for similarity thresholds of 0.0, 0.6, and 1.0, respectively. Figures 14(a), 14(b), and 14(c) present the results for each considered pair of versions of Jaba for similarity thresholds of 0.0, 0.6, and

1.0, respectively. The horizontal axis of each graph represents the value of LH, and the vertical axis of each graph shows the percentage described earlier. For example, for $S = 1.0$ and $LH = 20$ (see Figure 13(c)), `CalcDiff` matches on average approximately 10% more nodes than `LS`, when both algorithms run on the same pair of versions of DAIKON. For another example, for $S = 0$ and $LH = 20$ (see Figure 13(a)), `CalcDiff` matches about 45% more nodes than `LS` for JABA's pair of versions $\langle v5, v6 \rangle$. In this case, `CalcDiff` matches 1,319 additional nodes out of 3,233 nodes that `LS` identifies as unmatched nodes.

The results of this study show that `CalcDiff` performs better than `LS` in almost all cases in which LH is not 0. For $LH = 0$, the fact that `LS` performs better than `CalcDiff` confirms our intuition. Without lookahead, algorithm `CalcDIff` does not search further to find a match when two hammocks do not match, whereas `LS` matches hammocks and nodes by searching the graphs until the hammock exit node is found. It is worth noting that the `LS` approach has a higher cost than `CalcDiff`. In fact, as we verified in this study, `LS` always takes longer than `CalcDiff` to compare two versions.

We further analyzed the data and found that the few cases in which algorithm `CalcDiff`, with $LH \geq 10$, performs worse than `LS` are special cases in which: (1) there are very few changes between the considered versions; and (2) the headers of the hammocks that contain changes are the same, which favors `LS`'s hammock-matching strategy.

In all other cases considered (i.e., $LH \neq 0$ and other configurations except JABA $\langle v2, v3 \rangle$ with $S = 0.0$), `CalcDiff` matches more nodes than `LS` and improves the matching results by 10% on average for DAIKON and up to 90% for JABA. The results also show that the number of matched nodes increases when LH increases, which confirms our intuition. Finally, the results show that, for $LH > 10$, the number of matched nodes is slightly greater in the case of $S = 0.6$ and $S = 1.0$ than in the case of $S = 0$.

Note that added nodes identified by `LS` or `CalcDiff` can be classified as (1) code that is actually added or (2) code that cannot be matched because of the limitations of the algorithms. Therefore, to measure the relative effectiveness of the two algorithms, we should compute the percentage using the number of nodes in the second category only. In this sense, the percent improvement reported in this study underestimates the actual improvement.

3.4.5 Discussions

The empirical results show that, for both subjects the increase in the running time of JDiff when $S = 1.0$ is not noticeable in most cases. In only few cases does the running time increase, but by at most a factor of two. Where the effectiveness of the technique is concerned, JDiff performs noticeably better when $S = 1.0$ than when $S = 0.0$ or 0.6 for any value of LH. Therefore, based on the empirical results, we suggest using 1.0 as the value of S.

With the suggested value of the similarity threshold ($S = 1.0$), in most cases the running time of JDiff when $LH = 30$ remains almost the same compared to the time when $LH = 0$. In the other cases, the running time increases at most by a factor of two. The effectiveness, however, increases considerably when $LH = 30$ compared to when $LH = 0$. Therefore, we suggest using 30 as the value of LH.

CHAPTER 4

DYNAMIC IMPACT ANALYSIS

As shown in Figure 1, after identifying changes, the testing-requirements identifier can begin computing requirements. However, when testing time and resources are limited, developers may want to focus on the parts of software that are more likely to be exercised in the field first. The identifier, thus, requires a technique to identify the parts of software that, based on a set of executions, are more likely to be affected by the changes. The identifier can use this impact information to prune the affected parts of software that are less likely to be exercised and, thus, restrict the cost of symbolic execution by reducing the number of statements to be analyzed. Software change impact analysis, which estimates the potential effects of changes, can provide this impact information.

4.1 Related Work

Many impact-analysis techniques are presented in the literature. This research focuses on dependency-based impact analysis [8]. Existing dependency-based impact-analysis techniques (e.g., [8, 46, 58, 67, 71]) rely on static analyses such as static forward slicing or transitive closure on call graphs. Although, these techniques can compute the safe estimated impact of changes, their conservative assumptions often result in the *impact set*, the subset of affected program components, that includes most of the software. The problem of sound static-analysis-based techniques is that they consider all possible behaviors of the software, some of which may not be exercised by the users. Moreover, they often include some impossible behaviors due to the imprecision of the analysis. Therefore, recently, researchers have investigated and defined impact-analysis techniques based on dynamic information about program behavior [10, 44, 45]. The dynamic information consists of execution data for a specific set of program executions such as executions in the field, execution based on an operational profile, or executions of test suites. Dynamic impact analysis estimates the subset of program entities that are affected by the changes during at least one of the

considered program executions.

All existing dynamic-impact-analysis techniques are based on the same technique called `PathImpact`. To illustrate this main technique, consider the example, original version P used in Chapter 3 (Figure 3). Figure 15 shows the partial code of P that is relevant to the discussion in this chapter. The partial call graph of P used to illustrate the technique is shown in Figure 16. To shorten the traces used in the discussion, each method is referred to by a unique identifier. The edge connecting node D to node C is a back-edge in the call graph and, thus, indicates recursion.

As part of the example, dynamic execution data corresponding to possible executions of program P, shown in Figure 17, are provided. Each line in this figure corresponds to a trace of an execution of P and consists of an identifier for the execution followed by a list of method call and return events during that execution. Each method call is represented by the method name. Each `r` represents the return from the most recently called method, and `x` represents the exit from the program. For example, the trace for Exec2 corresponds to an execution in which M is called, M calls B, B calls C, C returns to B, B calls G, G returns to B, B returns to M, and M exits.

`PathImpact` [44] relies on instrumentation to collect dynamic information from a running software system. As the instrumented program executes, it records multiple execution traces, of the form shown in Figure 17. `PathImpact` first processes these traces sequentially using the SEQUITUR compression algorithm[1] [50] to obtain a compact representation called a whole-path DAG (directed acyclic graph) [41]. The execution traces can be processed as they are generated and need not be stored, and the whole-path DAG is used instead of the traces to calculate the change impact set. Given the execution traces in Figure 17, and assuming that the sequence CErFrDrr is repeated twice in Exec4, the resulting DAG is shown in Figure 18. The compression algorithm creates rules, shown as numbered interior nodes in the DAG, to remove repetition within and across traces. Note that Holzmann and Puri's algorithm for representing model states as a minimized automaton [33]

[1]SEQUITUR is an algorithm for recursive construction of a hierarchical structure from a sequence of discrete symbols by replacing repeated phrases with a new symbol whose grammatical rule generates the phrase.

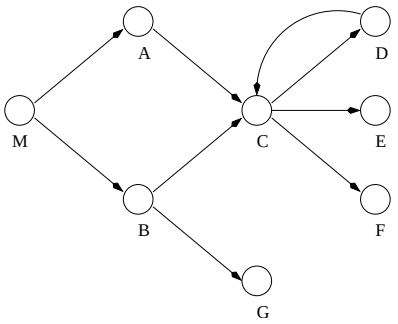
```
    Program P

public class UnavailBookFinder { ...
  public static void main(String[] args) { ...
        Set<Book> res = lib.getCheckedOutBooks();
        ...
            res.addAll( lib.getOnOrderBooks());
        ...
  }
  ...
}

public class Library { ...
  public Set<Book> getCheckedOutBooks() {
      ...
      if ( book.getStatus() == Book.CHECKED_OUT) { ... }
      ...
  }
  public Set<Book> getOnOrderBooks() {
      ...
      if ( book.getStatus() == Book.ON_ORDER) {
        ...
        Date d = book.getDeliveryDate();
        ...
      }
       ...
  }
}

public class Book {
  ...
  public int getStatus() {
    ...
    boolean b1 = isCheckedOut();
    boolean b2 = isOnOrder();
    int c1 = getNumAvailableCopies();
    ...
  }
  public boolean isCheckedOut() { ... }
  public boolean isOnOrder() { ... }
  public int getNumAvailableCopies() {
    ...
    boolean b1 = anotherBook.getStatus();
    ...
  }
  public Date getDeliveryDate() { ... }
}
```

Figure 15: Partial code for the original version P.



Id	Method name
M	UnavailBookFiner.main
A	Library.getCheckedOutBooks
B	Library.getOnOrderBooks
C	Book.getStatus
D	Book.getNumAvailableCopies
E	Book.isCheckedOut
F	Book.isOnOrder
G	Book.getDeliveryDate

Figure 16: Call graph for program P.

```
Exec1: M B G r G r r A C E r r r x
Exec2: M B C r G r r x
Exec3: M A C E r D r r r x
Exec4: M B C E r F r D r r ... C E r F r D r r r x
```

Figure 17: Traces for P. The dots in the last trace indicate that the sequence CErFrDrr is repeated several times.

can be used in place of the SEQUITUR compression algorithm to provide lower runtime overhead and better compression ratio.

`PathImpact` traverses the DAG to determine an impact set, given a set of changes. Law and Rothermel present the complete algorithm for performing this traversal [45]. Intuitively, one way to visualize its operation is to consider beginning at a changed method's node in the DAG, traversing through the DAG by performing recursive forward and backward in-order traversals, and stopping when any trace termination symbol, `x`, is found. By traversing forward in the DAG, the algorithm finds all methods that execute after the change and therefore could be affected by the change. By traversing backward in the DAG, the algorithm determines all methods into which execution can return. More precisely, `PathImpact` performs a forward traversal by visiting each symbol on the right-hand side of the rule in an interior node from left to right and by visiting the method symbol in a leaf node. When visiting each non-terminal symbol a in rule r, `PathImpact` traverses the DAG until it reaches the interior node corresponding to symbol a, continues its traversal on symbols on the right-hand side of the rule associated with that node, and, after visiting the last node in that rule, returns to the next symbol after symbol a in rule r. `PathImpact`

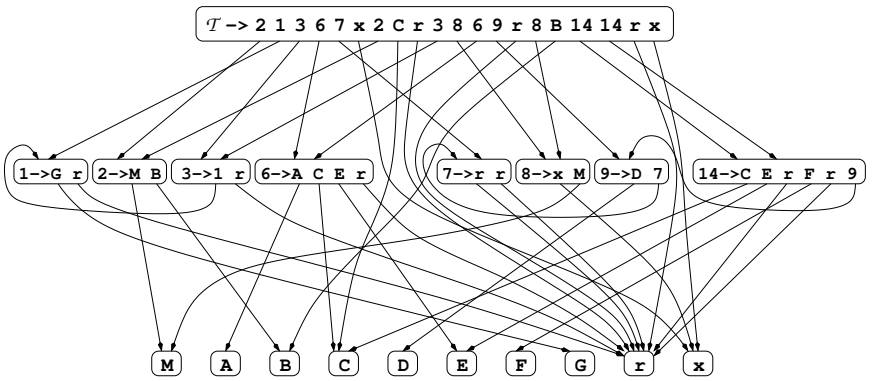


Figure 18: Whole-path DAG for the execution traces in Figure 17.

performs a backward traversal similarly except that it traverses the symbols on the right-hand side of a rule from right to left.

To illustrate, consider the impact set computed by `PathImpact` for the program and the executions in this example (Figures 16 and 17) for change $Z = \{\text{A}\}$ (i.e., only method A is modified). `PathImpact` starts at leaf node A, traverses forward to interior node $6 \rightarrow ACEr$, and visits symbols (and leaf nodes) C, E, and r. After visiting the last symbol in this node, `PathImpact` traverses forward to node $\tau \rightarrow 213...$ and continues its traversal after each symbol 6 in that node. For the first occurrence of 6, `PathImpact` visits symbol 7. Because symbol 7 is a non-terminal symbol, `PathImpact` traverses node $7 \rightarrow rr$ and visits symbol r twice before returning to node $\tau \rightarrow 213...$ again and stops when it finds x. After forward traversal, `PathImpact` finds that $CErrrx$ is the sequence that follows method A in one of the executions that exercise A. This sequence means that A calls C, C calls E, E returns into C, C returns into A, and A returns into a caller before the program terminates. Thus, `PathImpact` includes C and E in the impact set and performs backward traversal to identify the caller of A. The backward traversal is performed similarly as described above. The resulting impact set computed by `PathImpact` is {M, A, C, D, E}.

`PathImpact` incurs significant runtime overhead because it requires time linear in the size of the DAG computed so far to add a new event to the DAG (compressed traces) on the fly. In terms of space, the traces, even when compressed, can be very large.

Breech and colleagues present an algorithm for computing the same impact sets as `PathImpact` does, but on the fly [10]. Their algorithm collects, for each execution, an

impact set for each method. At the entry of a method X, the algorithm adds X to the impact set of each method currently on the call stack. Then, it adds all methods on the call stack to X's impact set. The worst-case space complexity is quadratic in the number of methods. The worst-case time complexity per method call is $O(n)$, where n is the number of methods. Moreover, their technique assumes that all executions terminate with an empty call stack.

Sections 4.2 and 4.3 present two new approaches, `CoverageImpact` and `ExecuteAfter`, for dynamic impact analysis with significantly lower runtime overhead than `PathImpact`. `CoverageImpact` computes impact sets by combining coverage information with static slicing whereas `ExecuteAfter` computes impact sets by analyzing the execute-after relations obtained from executions. The two techniques have precision-efficiency trade-off (i.e., `ExecuteAfter` technique computes more precise impact sets than `CoverageImpact` while incurring more overhead).

4.2 Coverage-Based Dynamic Impact Analysis

The `CoverageImpact` technique relies on lightweight instrumentation to collect dynamic information from a running software system. As the instrumented program executes, it records coverage information in the form of bit vectors, and `CoverageImpact` uses these bit vectors to compute the impact set. The bit vectors contain one bit per method. A value of 1 in the bit for method m in the vector for execution e indicates that m was covered in e, and a value of 0 indicates that m was not covered in e. For example, for the executions shown in Figure 17, the coverage information consists of the set of bit vectors shown in Table 3.

Table 3: Coverage bit vectors for the execution traces in Figure 17.

Exec ID	M	A	B	C	D	E	F	G
Exec1	1	1	1	1	0	1	0	1
Exec2	1	0	1	1	0	0	0	1
Exec3	1	1	0	1	1	1	0	0
Exec4	1	0	1	1	1	1	1	0

Given a set of changed methods *CHANGES*, `CoverageImpact` uses the bit vectors collected during execution to determine an impact set. This section provides an overview of

Orso et al.'s algorithm for this analysis [55]. To identify the impact set, `CoverageImpact` computes, for each method X in *CHANGES*, a dynamic forward slice based on the coverage data for executions that traverse X. The impact set is the union of the slices thus computed.

`CoverageImpact` computes the impact sets in two steps. First, using the coverage information, it identifies the executions that traverse at least one method in the change set *CHANGES* and marks the methods covered by such executions. Second, it computes a static forward slice from each change in *CHANGES* considering only marked methods. The impact set is the set of methods in the computed slices.

To illustrate, consider the impact set computed by `CoverageImpact` for the program and executions in this example (Figures 16 and 17, Table 3) for change *CHANGES* = {A} (i.e., only method A is modified). The executions that traverse A are Exec1 and Exec3, and the covered methods include M, A, B, C, D, E, and G. Assume that the traditional static slice for method A consists of methods M, A, C, D, E, and F. The resulting impact set —the slice computed considering only marked methods—would then be {M, A, C, D, E}.

4.3 Execute-After-Relation-Based Dynamic Impact Analysis

`CoverageImpact` incurs low runtime overhead to collect dynamic information because it needs only coverage information. It, however, may compute imprecise impact sets. For example, in an execution where a method, X, is executed only before the only changed method, Y, if the static slice of method Y includes X, `CoverageImpact` will compute an imprecise impact set for method Y (i.e., it includes method X, which should not be affected in this execution) because method X is in the static slice and is marked as executed. Therefore, another dynamic-impact-analysis approach that computes impact sets with the same precision as `PathImpact` and the cost comparable to `CoverageImpact` is developed.

This section discusses the findings on what information is essential for computing dynamic impact sets, introduces a new algorithm for collecting this information efficiently during program executions, and presents a proof of correctness for the algorithm.

4.3.1 The Execute-After Relation

Dynamic impact analysis computes, for one or more program changes, the corresponding dynamic impact set: the set of program entities that *may* be affected by the change(s) for a specific set of program executions.[2] Intuitively, all entities that are executed after a changed entity are potentially affected by that change. As such, the dynamic impact set for a changed entity e must include all program entities that are executed after e in the considered program executions.

Therefore, to compute dynamic impact sets for a program P and a set of executions E, the only information required is whether, for each pair of entities $e1$ and $e2$ in P, $e2$ was executed after $e1$ in any of the executions in E. This binary relation, named *Execute After* (*EA* hereafter), can be defined for entities at different levels of granularity. For ease of comparison with existing dynamic impact-analysis techniques, the EA relation is defined formally for the case in which the entities considered are methods and executions are single-threaded. The generalization of the EA relation with regard to multi-threaded executions is discussed in Section 4.3.3.

Definition 5. Given a program P, a set of executions E, and two methods X and Y in P, (X, Y) $\in$ EA for E if and only if, in at least one execution in E,

1. Y calls X (directly or transitively),
2. Y returns into X (directly or transitively), or
3. Y returns into a method Z (directly or transitively), and method Z later calls X (directly or transitively).

The problem with existing dynamic-impact-analysis techniques [10, 44, 45, 55] is that they do not explicitly compute the EA relation. Instead, they infer the relation from information that is either too expensive to collect or too imprecise to provide accurate results. For example, technique `PathImpact` uses complete program traces to identify which

[2]The uncertainty occurs because the analysis uses the execution information obtained from executing the original version to estimate the affected program entities in the modified version.

methods are executed after a change (see Section 4.1). For another example, technique `CoverageImpact` uses coverage information combined with static slicing to approximate the information contained in complete program traces. Preliminary investigation shows that trace information is excessive for both deriving the EA relation and performing dynamic impact analysis because it contains much unnecessary information. The following discussion demonstrates that execution traces contain mostly redundant information and presents the considerably smaller amount of information that the new technique collects at runtime.

The first finding, when analyzing the information contained in program traces such as the ones in Figure 17, is that using only the information provided by method-return events unnecessarily complicates the analysis of the traces. Method-return events can be used to identify the methods into which the execution returns, but provide this information only indirectly—some form of the stack-based walk of the traces is typically required to identify such methods. To simplify the dynamic impact analysis, the new technique collects, instead of method-return events, method-returned-into events. A *method-returned-into event* for a method X, denoted as X_i, is generated when an execution returns (from any method) into X. For now, assume that method-returned-into events can be easily collected. Section 4.4.2 discusses how to collect such events for Java programs efficiently. By considering only method-entry and method-returned-into events, trace Exec 1 in Figure 17 can be rewritten as follows:

$$\boxed{\mathtt{M}_e\ \mathtt{B}_e\ \mathtt{G}_e\ \mathtt{B}_i\ \mathtt{G}_e\ \mathtt{B}_i\ \mathtt{M}_i\ \mathtt{A}_e\ \mathtt{C}_e\ \mathtt{E}_e\ \mathtt{C}_i\ \mathtt{A}_i\ \mathtt{M}_i}$$

The rest of the section uses this example trace to illustrate how to capture the EA relation between any pair of methods in an execution.

Obviously, the EA relation can be derived from this complete trace. By the definitions of method-entry and method-returned-into events, an observation is as follows:

> Method X executes after method Y if, in the trace, there is a method-entry or method-returned-into event for X that follows a method-entry or method-returned-into event for Y.

However, if the only goal is to derive the EA relation, the complete trace contains much unnecessary information. In fact, the above observation can be restated as follows:

(X, Y) ∈ EA iff at least one event for X occurs after at least one event for Y.

To assess whether at least one event for X occurs after at least one event for Y, an analysis does not need a complete trace—the first method event for Y (referred to as $\mathtt{Y}_f$), the last method event for X (referred to as $\mathtt{X}_l$), and the ordering of the two events in the trace are adequate. If an event $\mathtt{Y}_*$[3] for method Y occurs before an event $\mathtt{X}_*$ for method X, then necessarily $\mathtt{Y}_f$ occurs before $\mathtt{X}_l$: by definition, $\mathtt{Y}_f \leq \mathtt{Y}_* < \mathtt{X}_* \leq \mathtt{X}_l$. Conversely, if $\mathtt{Y}_f$ occurs after $\mathtt{X}_l$, then there cannot be any $\mathtt{X}_*$ and $\mathtt{Y}_*$ such that $\mathtt{Y}_*$ occurs before $\mathtt{X}_*$: $\mathtt{Y}_* < \mathtt{X}_*$ contradicts $\mathtt{X}_* \leq \mathtt{X}_l < \mathtt{Y}_f \leq \mathtt{Y}_*$.

One conclusion is that, in general, the essential information for deriving the EA relation for an execution is, for each method, the first and the last events that the method generates in the execution. The first event for a method X always corresponds to the first method-entry event for X. The last event for a method X corresponds to the last method-entry event for X or the last method-returned-into event for X, whichever comes last. Intuitively, the first and last events for a method represent the first and last executions of the method, where an *execution* of a method means an execution of one or more statements in the method's body.

By considering only the first and the last events for each method, the previous example trace can be reduced to the following sequence:

$$\boxed{\mathtt{M}_e\ \mathtt{B}_e\ \mathtt{G}_e\ \mathtt{G}_e\ \mathtt{B}_i\ \mathtt{A}_e\ \mathtt{C}_e\ \mathtt{E}_e\ \mathtt{C}_i\ \mathtt{A}_i\ \mathtt{M}_i}$$

To simplify the discussion, in the rest of this section, the notation for method events introduced above is used: $\mathtt{X}_f$ indicates the first method event for a method X, and $\mathtt{X}_l$ indicates the last method event for X. Using this notation, the above trace can be rewritten as follows:

$$\boxed{\mathtt{M}_f\ \mathtt{B}_f\ \mathtt{G}_f\ \mathtt{G}_l\ \mathtt{B}_l\ \mathtt{A}_f\ \mathtt{C}_f\ \mathtt{E}_f\ \mathtt{E}_l\ \mathtt{C}_l\ \mathtt{A}_l\ \mathtt{M}_l}$$

Note that, because there is only one event for method E, the event appears as both the first and the last. This sequence contains at most two entries for each method in

[3]The notation $\mathtt{Y}_*$ and $\mathtt{X}_*$ indicates any event for method Y and X, respectively.

the program. Because this sequence contains the EA relation, it is referred to as *EA sequence*. As discussed previously, the EA sequence contains the essential information needed to perform dynamic impact analysis.

Dynamic impact analysis computed by using EA sequences is as precise as an analysis performed on complete traces, while achieving significant space savings. These savings are obvious when dynamic information is collected for real executions, in which methods can be executed thousands (or millions) of times. However, achieving space savings by collecting EA sequences would not be useful if, to collect them, complete execution traces need to be gathered first. Therefore, the new algorithm, presented in the next section, for collecting EA sequences on the fly at a cost comparable to the cost of collecting simple method coverage information is developed.

4.3.2 Algorithms

One straightforward way to collect EA sequences is to use a list of events and update it (1) at each method entry and (2) every time the flow of control returns into a method after a call. The update must operate such that only the first and the last events for each method are kept in the list. Therefore, every time an event for a method X is generated, this approach checks whether the list already contains entries for X. If not, it adds both an $\texttt{X}_f$ entry and an $\texttt{X}_l$ entry at the end of the list. Otherwise, if there is already a pair of entries, the approach removes the existing $\texttt{X}_l$ entry and adds a new $\texttt{X}_l$ entry at the end of the list. (Intuitively, this approach only records the first method event and keep updating the last method event.) This straightforward approach is space efficient—the space required never exceeds $2n$, where n is the number of methods in the program. However, it is not time efficient because, for every method event generated, the event list must be searched and updated. The searching time could be eliminated by keeping a pointer to the last event for each method and by suitably updating such pointers every time a method event is generated. However, this approach still needs to update up to five pointers at each event and is, thus, penalized by the memory-management overhead.

Algorithm `CollectEA`
Declare: F array of first method events
L array of last method events
C counter
n number of methods in the program
Output: F, L
{On program start}
Begin:
1: initialize F[i] to $\perp$, for $0 \leq$ i $< n$
2: initialize L[i] to $\perp$, for $0 \leq$ i $< n$
3: initialize c to 1
end
{On entry of method M}
Begin:
4: **if** (F[M] = $\perp$) **then**
5: F[M] = c
6: **endif**
7: L[M] = c
8: increment c by 1
end
{On control returning into method M}
Begin:
9: L[M] = c
10: increment c by 1
end
{On program termination}
Begin:
11: output F, L
end

Figure 19: Algorithm `CollectEA`.

To minimize the overhead imposed by the analysis, an algorithm is developed for collecting EA sequences at runtime that is more efficient (by a constant factor) than the list approach, in terms of both time and space, and does not incur memory-management overhead. The new algorithm is based on the use of two arrays of event timestamps, F and L. Arrays F and L are used to store the timestamp of the first and last events, respectively, generated by each method. The notation F[X] (resp., L[X]) denotes the element of array F (resp., L) for a method X. The timestamp is a global counter that is incremented by one at each event. Figure 19 shows the algorithm, `CollectEA`.

`CollectEA` is an on-line algorithm, whose different parts are triggered by the events that occur during a program execution. When the program starts, all elements of arrays F and L are initialized to $\perp$, and counter c is initialized to 1 (lines 1–3). $\perp$ denotes a non-numeric

Table 4: Values of F, L, and c during the example execution.

event	F								L								c
	M	A	B	C	D	E	F	G	M	A	B	C	D	E	F	G	
start	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	1
M_e	1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	2
B_e	1	⊥	2	⊥	⊥	⊥	⊥	⊥	1	⊥	2	⊥	⊥	⊥	⊥	⊥	3
G_e	1	⊥	2	⊥	⊥	⊥	⊥	3	1	⊥	2	⊥	⊥	⊥	⊥	3	4
B_i	1	⊥	2	⊥	⊥	⊥	⊥	3	1	⊥	4	⊥	⊥	⊥	⊥	3	5
G_e	1	⊥	2	⊥	⊥	⊥	⊥	3	1	⊥	4	⊥	⊥	⊥	⊥	5	6
B_i	1	⊥	2	⊥	⊥	⊥	⊥	3	1	⊥	6	⊥	⊥	⊥	⊥	5	7
M_i	1	⊥	2	⊥	⊥	⊥	⊥	3	7	⊥	6	⊥	⊥	⊥	⊥	5	8
A_e	1	8	2	⊥	⊥	⊥	⊥	3	7	8	6	⊥	⊥	⊥	⊥	5	9
C_e	1	8	2	9	⊥	⊥	⊥	3	7	8	6	9	⊥	⊥	⊥	5	10
E_e	1	8	2	9	⊥	10	⊥	3	7	8	6	9	⊥	10	⊥	5	11
C_i	1	8	2	9	⊥	10	⊥	3	7	8	6	11	⊥	10	⊥	5	12
A_i	1	8	2	9	⊥	10	⊥	3	7	12	6	11	⊥	10	⊥	5	13
M_i	1	8	2	9	⊥	10	⊥	3	13	12	6	11	⊥	10	⊥	5	14

special value used to identify methods that have not yet been executed. (If a method has value ⊥ at the end of the execution, then that method was not executed at all in that execution.) Every time a method M is entered, the algorithm checks the value of F[M]. If F[M] is ⊥ (i.e., M has not yet been executed), then the algorithm sets F[M] to the current value of the counter (lines 4–6). Because, as discussed in the previous section, the last event generated by M may be a method-entry event, L[M] is also set to the current value of the counter (line 7). Finally, counter c is incremented by one (line 8).

Every time the control flow returns into method M, L[M] is updated to the current value of the counter (line 9), and counter c is incremented by one (line 10). In this way, each element in L contains the timestamp of the last time the corresponding method was (partially) executed.

To illustrate the algorithm, consider the execution producing the example trace Exec1 used in Section 4.1 (see Figure 17). Table 4 shows the values of F, L, and c after each method (and program) event. The leftmost column (event) shows the program-start, method-entry, and method-returned-into events. Columns labeled F and L show, for each method, the values of the corresponding elements in the F and L arrays, respectively. Finally, the rightmost column (c) shows the value of the counter.

On program start, F, L, and c are initialized. When M is called, F[M] and L[M] are set to 1, the current value of c, and c is incremented to 2. Likewise, when B is called, F[B] and L[B] are set to 2, and c is incremented to 3. When G is called, F[G] and L[G] are set to 3, and c is incremented to 4. Then, when the control flow returns into B, which generates a $\mathtt{B}_i$ event, L[B] and c are updated accordingly. When method G is called again, F[G] is not updated (because its value is not $\perp$), L[G] is updated, and c is incremented. Additional updates of F, L, and c occur in an analogous way until the program terminates.

The equivalency of the information in a pair of F and L arrays and the information in an EA sequence is illustrated by presenting the steps to derive one from the other and vice-versa. (Note that maintainers need not perform these steps to obtain impact sets from a pair of F and L arrays.) Converting a pair of F and L arrays to an EA sequence requires two steps: (1) order the elements of F and L (considered together and without including elements with value $\perp$) based on their value (if F[X] equals L[X], F[X] precedes L[X]), and (2) for each method X, replace F[X] with $\mathtt{X}_f$ and L[X] with $\mathtt{X}_l$. Converting an EA sequence to a pair of F and L arrays requires the two previous steps to be reversed: (1) for each method X, $\mathtt{X}_f$ is replaced with F[X], and $\mathtt{X}_l$ is replaced with L[X], and (2) increasing values, starting from 1, is assigned to the elements in arrays F and L, based on their position.

For example, the pair of F and L arrays for the previous example (shown as the last row of Table 4) would first be ordered,

F[M]	F[B]	F[G]	L[G]	L[B]	F[A]	F[C]	F[E]	L[E]	L[C]	L[A]	L[M]

and then be converted as follows:

$\mathtt{M}_f$	$\mathtt{B}_f$	$\mathtt{G}_f$	$\mathtt{G}_l$	$\mathtt{B}_l$	$\mathtt{A}_f$	$\mathtt{C}_f$	$\mathtt{E}_f$	$\mathtt{E}_l$	$\mathtt{C}_l$	$\mathtt{A}_l$	$\mathtt{M}_l$

Because arrays F and L provide the same information as an EA sequence, the EA relation can be derived from such arrays, as stated in the following lemma:

Lemma 1. (X, Y) $\in$ EA $\iff$ F[Y] $<$ L[X]

Proof. To prove Lemma 1, consider three characteristics of the algorithm:

1. Counter c increases monotonically each time a method event occurs.

2. For each method X, F[X] is set only once, to the then-current value of counter c, at the first method-entry event for X.

3. For each method X, L[X] is set to the then-current value of counter c every time a method event for X occurs.

The proof proceeds in two parts, by first showing that

$$\texttt{(X, Y)} \in \texttt{EA} \Rightarrow \texttt{F[Y]} < \texttt{L[X]} \quad \texttt{(1)}$$

and then showing that

$$\texttt{F[Y]} < \texttt{L[X]} \Rightarrow \texttt{(X, Y)} \in \texttt{EA} \quad \texttt{(2)}$$

Part (1). According to the definition of EA relation (Definition 5), there are three cases in which (X, Y) $\in$ EA

In the first case (Y calls X), a $\texttt{Y}_e$ event is generated at timestamp t_1 and an $\texttt{X}_e$ event is generated at timestamp $t_2 > t_1$. At the end of the execution, because of Characteristics 1, 2, and 3, F[Y] is either t_1 (if $\texttt{Y}_e$ is the first entry event for Y) or a value less than t_1 (otherwise), and L[X] is either t_2 (if $\texttt{X}_e$ is the last method event for X) or a value greater than t_2 (otherwise). Thus, F[Y] < L[X] in this case.

In the second case (Y returns into X), a $\texttt{Y}_e$ event is generated at timestamp t_1 (when X calls Y directly or transitively) and an $\texttt{X}_i$ event is generated at timestamp $t_2 > t_1$ (when Y returns). As for the previous case, F[Y] $\leq t_1$, and L[X] $\geq t_2$. Thus, F[Y] < L[X] also in this case.

In the third case (Y returns into a method Z that later calls X), a $\texttt{Y}_e$ event is generated at timestamp t_1 (when Z calls Y) and an $\texttt{X}_e$ event is generated at timestamp $t_2 > t_1$ (when Z calls X). As for the previous cases, F[Y] $\leq t_1$, and L[X] $\geq t_2$, and thus F[Y] < L[X] also in this case.

Because (X, Y) $\in$ EA implies F[Y] < L[X] in all three cases, part (1) of the Lemma holds.

Part (2). This part follows directly from the meaning of arrays F and L: if F[Y] < L[X], then the first (partial) execution of method Y precedes the last (partial) execution of

method X—X executes after Y. □

Now that the `CollectEA` algorithm has been shown to capture correctly the EA relation among methods for a given execution, the rest of this section discusses how the dynamic-impact-analysis technique uses such information to compute dynamic impact sets. To compute the dynamic impact set for a changed method, the technique includes every method whose timestamp in L is greater than or equal to the timestamp in F for the changed method. In the case of more than one changed method, this technique just needs to compute the impact set for the changed method, U, with the least timestamp in the F array. By definition, the impact set for U is a superset of the impact set computed for any of the other changed methods: any other changed method X has a greater timestamp than U and, thus, the set of methods executed after X is a subset of the set of methods executed after U. More formally, given a set of changed methods *CHANGES*, the technique identifies U and computes the dynamic impact set for *CHANGES* as follows:

$$\text{U} = \text{X} \mid \text{F[X]} \leq \text{F[Y]}, \text{X,Y} \in \textit{CHANGES}$$

$$\text{impact set for } \textit{CHANGES} = \{\ \text{X} \mid \text{L[X]} \geq \text{F[U]}\ \}$$

To illustrate this, consider the previous example execution and a *CHANGES* set that consists of A and C. In this case, U is method A, and the dynamic impact set for *CHANGES* is {M, A, C, E}. Note that changed methods that were not executed (i.e., methods whose timestamps are $\perp$ at the end of the execution) are not considered. In the case of multiple executions (i.e., multiple EA sequences), the impact set is computed by taking a union of the impact sets for the individual executions.

Lemma 2. The dynamic impact sets computed as described include (1) the modified methods and (2) all and only methods that are (partially) executed after any of the modified methods.

Proof. By definition, this technique computes dynamic impact sets with the following property:

$$\text{impact set} = \{\ \text{X} \mid \text{L[X]} \geq \text{F[Y]} \text{ for any modified method Y}\}$$

Lemma 2 follows immediately from Lemma 1 and from the above property. □

The space complexity of `CollectEA` is $O(n)$, where n is the number of methods in the program, because the algorithm needs two arrays, each of size n, and a counter. Compared to approaches that use traces, this algorithm achieves dramatic savings in terms of space because program traces, even if compressed, can be very large. For example, in previous work, traces even for relatively small programs [56] can be on the order of 2 gigabytes. The time overhead of `CollectEA` is a small constant per method call. At each method entry, the algorithm performs one check, one increment, and at most two array updates. Every time the control returns into a method, the algorithm performs one array update and one increment.

4.3.3 Multi-Threaded Executions

In multi-threaded executions, one method can be executed not only before or after another method, but also concurrently. According to the definition of dynamic impact analysis, any method (or part thereof) that is executed after a changed method is potentially affected by the change. Therefore, any method that is executed concurrently with a changed method is also potentially affected by the change because of possible interleaving of threads. Unfortunately, method-entry, and method-returned-into events are not enough to identify affected methods in these cases.

To illustrate, consider a multi-threaded program in which method A is entered at time t_1 and exited at time t_2, method B is entered at time t_3 and exited at time t_4, and $t_1 < t_3 < t_2$. In such a case, A and B are executed in parallel, and a possible sequence of events is (assuming that methods A and B are invoked by two methods X and Y, respectively):

$$\boxed{\texttt{...} \quad \texttt{A}_f \ \texttt{A}_l \ \texttt{B}_f \ \texttt{B}_l \ \texttt{X}_l \ \texttt{Y}_l}$$

If method B is a changed method, the above sequence does not provide enough information to identify A as possibly affected by B because it only appears before B. To address this problem and account for multi-threaded executions, algorithm `CollectEA` must be modified as follows. One pair of arrays F and L with a global counter is still adequate;

however, method-return events need to be collected. Method-return events provide the information about whether one method in a thread exits before or after the entry of another method in another thread. The algorithm treats method-return events in the same way as method-return-into events. For the example above, the trace would therefore change as follows:

$$\boxed{\ldots \quad \mathtt{A}_f \ \mathtt{B}_f \ \mathtt{A}_l \ \mathtt{X}_l \ \mathtt{B}_l \ \mathtt{Y}_l}$$

The impact sets can be computed from arrays F and L in the same way previously described.

4.4 Dynamic Impact Analysis Tools

4.4.1 `CoverageImpact` tool

Because `CoverageImpact` requires only method coverage information, it is implemented on top of InsECT [12] and uses available InsECT functionality to collect coverage information. The analysis part of the implementation takes as input the collected information and computes a conservative approximation of dynamic forward slicing by using reachability on static call graphs, which are built by JABA, that includes only executed methods.

4.4.2 EAT: An execute-after-based dynamic-impact-analysis tool

EAT is a tool written in Java that consists of three main components: (1) an instrumentation module, (2) a set of runtime monitors, and (3) an analysis module.

4.4.2.1 Instrumentation Module

The instrumentation module uses InsECT to instrument the program under analysis by adding probes that produce method events. In Section 4.3, method events are assumed to be easily produced. The way these events are produced in practice depends on the programming language that is targeted by the analysis. Because the subjects of the studies are Java programs, this section discusses how to collect the events for the Java language.

Collecting method-entry events is straightforward: simply instrument each method immediately before the first statement with a probe that generates an event with an attribute. The attribute is the numeric identifier for the method in which the event is generated.

Collecting method-returned-into events is more complicated because, in Java, there are three ways in which a method X can return into another method Y:

1. Normal return: X returns into Y because of a `return` statement or simply because X terminates. In this case, the execution continues at the instruction in Y that immediately follows the call to X.

2. Exceptional return into a catch block: while X executes, an exception is thrown that is not caught in X but is caught in Y. In this case, the execution continues at the first instruction in the catch block in Y that caught the exception.

3. Exceptional return into a finally block: while X executes, an exception is thrown that is not caught in X and not caught in Y, but Y has a finally block associated with the code segment that contains the (possibly indirect) call to X. In this case, the execution continues at the first instruction in the finally block in Y.

The instrumentation for a Java program for collecting method-returned-into events must handle these three cases. To this end, the instrumentation module instruments each call site by adding (1) a probe immediately before the instruction that follows the call site, (2) a probe before the first instruction of each catch block (if any) associated with the code segment that contains the call site, and (3) a probe before the first instruction of the finally block (if any) associated with the code segment that contains the call site. Each of these probes generates an event and attaches to the event, in the form of an attribute, the numeric identifier for the method in which the event is generated.

Note that the program instrumented in this way may generate some redundant method-returned-into events, but the correctness of the algorithm is preserved. For example, if method Y returns normally into method X, but there is a finally block in X associated with the code segment that contains the call to Y, then two probes will be triggered, which generate two $\texttt{X}_i$ events: one after the call and one in the finally block (which would be executed anyway). Every time an $\texttt{X}_i$ event is duplicated, the first event produced is simply discarded when the second event occurs (i.e., the value of element L[X] is set to the new

value of the counter), which is correct because the goal is to record the last time the method is executed. Such events are produced only in a few cases and, moreover, only require the update of one array element and the increment of a counter (duplication only occurs for method-returned-into events). Therefore, their impact on the efficiency of the approach is unnoticeable. Obviously, a more sophisticated instrumentation could avoid the production of these duplicated events, but the additional overhead would hinder the practicality of the approach.

The other two events required by this approach, program start and program termination, are already provided by InsECT [12] and only need to be enabled when instrumenting.

4.4.2.2 Monitors

The monitors are static methods that implement the four parts of the algorithm shown in Figure 19. The monitors initialize, update, and output the F and L arrays during program executions.

Leveraging InsECT functionality, the tool links the events generated by the probes with the appropriate monitors. Therefore, when a method event is generated, InsECT calls the appropriate static method and passes the event attribute (i.e., the identifier of the method in which the event was generated) as a parameter. When a program event is generated, which happens only at program start and program termination, InsECT simply calls the appropriate method with no parameter.

4.4.2.3 Analysis Module

The analysis module inputs the arrays produced by the monitors and the change information and outputs dynamic impact sets. To compute the impact sets, the analysis module uses the approach described in Section 4.3.2.

4.5 Empirical Studies on Dynamic-Impact-Analysis Algorithms

The studies in this section investigate the following research questions:

RQ1: How much overhead does the instrumentation required by the two techniques (`CoverageImpact` and `CollectEA`) impose, in practice, on the programs under analysis

compared with each other?

RQ2: How much does technique `CollectEA` gain, in terms of precision, with respect to technique `CoverageImpact`?

RQ3: How much does the analysis results obtained from using the field data (execution information from deployed instances of software), differ from the results obtained from using synthetic data (e.g., execution information from in-house test suites)?

This section presents experimental setup, describes the empirical studies and discuss their results.

4.5.1 Experimental Setup

4.5.1.1 Subject programs used in dynamic-impact-analysis studies

Table 5: Subject programs

Program	Versions	Classes	Methods	LOC	Test Cases
SIENA	8	24	219	3674	564
JABA	11	355	2695	33183	215

This study used as subjects several versions of two programs—JABA and SIENA—summarized in Table 5. The table shows, for each subject, the number of versions (*Versions*), the number of classes (*Classes*), the number of methods (*Methods*), the number of non-comment lines of code (*LOC*), and the number of test cases in the subject's test suite (*Test Cases*). The number of classes, methods, and lines of code is averaged across versions.

SIENA [11] is an Internet-scale event notification middleware for distributed event-based applications; and JABA is a framework for analyzing Java programs. For both subjects, we extracted from their CVS repositories consecutive versions from one to a few days apart.

4.5.1.2 Method and Measures

Because dynamic impact analysis requires dynamic information, these studies used the test suites for the subjects as input sets. The test suite for SIENA was created for earlier experiments [54]. The test suite for JABA, which was also available through CVS, was created and used internally by the program developers. To examine the differences in

runtime overhead for short and long executions, we divided this test suite two parts: short tests (125 test cases) and long tests (90 test cases). The short and long tests take on average approximately 430 ms and 5250 ms, respectively, to run. The results for the short and the long tests are labeled differently, as *Jaba* and *Jaba-long*, respectively.

As change sets for use in assessing impacts, these studies used the actual sets of changes from each version of the subject programs to each subsequent version. To compute such changes, these studies used JDiff, a differencing tool discussed in Section 3.3.

4.5.2 Study 1: Costs

This study investigates RQ1. To evaluate relative execution costs for `CollectEA` and `CoverageImpact`, we measured the time required to execute an instrumented program on a set of test cases, gather the dynamic data (F and L arrays for `CollectEA`, and method coverage for `CoverageImpact`), and output that information to disk. This study compares the execution costs of the two techniques to each other and to the cost of executing a non-instrumented program on the same set of test cases. Because timing data are collected for each individual test case, the results are computed by considering each test case in a test suite independently and then averaging the results across all test cases in the test suite.

The results of this study are shown in Table 6. For each program and version, the table reports the average execution time of each individual test case on the uninstrumented program, on the program instrumented by `CoverageImpact`, and on the program instrumented by `CollectEA`. The table also reports the minimum, average, and maximum percentage overhead imposed by `CoverageImpact` (*%CoverageImpact Overhead*) and by `CollectEA` (*%CollectEA Overhead*).

As the table shows, the overhead imposed by `CollectEA` varies widely depending on the subject (on average about 110% for Siena and 13% for Jaba) and also for different executions of a given program version (e.g., it varies from 3% to 20% for Jaba-Long-v9). The overhead for `CoverageImpact` shows a similar trend.

Careful investigation of the results has shown that the observed variation is caused by a fixed cost associated with the instrumentation. Such fixed cost is due to the time

Table 6: Execution time (ms)

Program	*Uninstru-mented*	*Coverage-Impact*	*CollectEA*	*%CoverageImpact Overhead*			*%CollectEA Overhead*		
				min	*avg*	*max*	*min*	*avg*	*max*
Siena-v0	52	107	109	92	106	196	98	111	163
Siena-v1	52	107	109	93	106	204	98	111	166
Siena-v2	52	107	110	93	106	170	98	112	165
Siena-v3	53	108	110	91	104	189	98	108	164
Siena-v4	53	108	110	91	104	183	96	108	158
Siena-v5	53	108	110	80	104	194	88	108	166
Siena-v6	53	108	110	84	104	200	90	108	166
Jaba-v0	421	451	475	5	7	10	10	13	17
Jaba-v1	423	453	476	5	7	10	10	13	15
Jaba-v2	423	453	476	4	7	10	9	13	15
Jaba-v3	424	454	477	5	7	10	10	13	15
Jaba-v4	428	459	483	5	7	11	11	13	14
Jaba-v5	429	459	483	5	7	10	10	13	15
Jaba-v6	429	459	483	5	7	10	11	13	15
Jaba-v7	429	459	483	5	7	10	11	13	15
Jaba-v8	452	489	511	5	7	8	6	12	14
Jaba-v9	461	496	514	5	8	14	5	12	14
Jaba-long-v0	5170	5591	5728	3	9	15	3	12	27
Jaba-long-v1	5125	5497	5749	3	9	14	3	13	26
Jaba-long-v2	5128	5496	5737	1	8	11	3	13	26
Jaba-long-v3	5170	5508	5763	2	8	13	3	13	28
Jaba-long-v4	5300	5668	5903	2	8	13	3	13	26
Jaba-long-v5	5304	5641	5928	2	8	15	3	13	26
Jaba-long-v6	5363	5715	5960	1	8	15	3	13	30
Jaba-long-v7	5313	5693	5932	1	8	13	3	13	29
Jaba-long-v8	5338	5674	5943	1	7	12	5	12	20
Jaba-long-v9	5360	5689	5969	1	6	12	3	13	20

required to (1) load and initialize the instrumentation-related classes and data structures, and (2) store the dynamic information on disk on program termination. For short running executions, such as the executions of SIENA, the fixed cost is considerable, whereas for longer executions it is less relevant. For example, for JABA, all the executions that require more than a few seconds (about four seconds, for the executions considered) have an overhead consistently below 15% and as low as 3% in many cases. Although there are not enough data points to generalize these results, they are encouraging. The results are especially encouraging because most real programs execute for more than a few seconds (e.g., most interactive programs). Moreover, the tools used in the studies are only prototypes without any optimizations, so reducing considerably both fixed and variable costs associated with the instrumentation may be possible.

When compared to `CoverageImpact`, `CollectEA` is, as expected, more expensive than `CoverageImpact`. However, the practical difference between the two techniques is small, ranging, on average, from 7% in the worst case (for *Jaba-long-v9*), to 3% in the best case (for

Jaba-long-v9). Therefore, we can conclude that `CollectEA` is practical for most programs, especially programs that run for more than a very short period of time. We can also conclude that `CollectEA` is applicable in all cases in which `CoverageImpact` is applicable.

4.5.3 Study 2: Precision

This study investigates RQ2. To this end, the study compares `CollectEA` with `CoverageImpact` in terms of precision. As a sanity check for the implementation of both tools and to reduce the threats to internal validity, we also compare `CollectEA` with `PathImpact` to ensure that they produce the same results. To evaluate the precision of the techniques, this study measures the relative sizes of the impact sets computed by the techniques on a given program, change set, and set of program executions. This study reports and compares such sizes in relative terms, as a percentage over the total number of methods. This study is an extension of the studies presented in the previous work [56], in which only a subset of the executions considered here are reported (due to the cost of the most expensive technique considered, `PathImpact`). For this study, we implemented a version of technique `PathImpact` that does not compress the traces and, thus, has an acceptable time overhead (at the cost of a huge space overhead).

The graph in Figure 20 shows the results of the study. In the graph, each version of the two programs[4] occupies a position along the horizontal axis, and the relative impact-set size for that program and version is represented by a vertical bar—dark grey for technique `CoverageImpact`, light grey for `PathImpact`, and black for `CollectEA`. The height of the bars represents the impact set size, averaged across all test cases, expressed as a percentage of the total number of methods in the program. As expected, the graph shows that, in all cases, the impact sets computed by `CollectEA` and `PathImpact` are identical (but computed at very different costs). The graph also shows that the impact sets computed by `CollectEA` are always more precise than those computed by `CoverageImpact`. In some cases, such differences in precision are considerable (e.g., for *Siena-v6*, *Jaba-v4*, and *Jaba-v5*). Therefore, the limited additional overhead imposed by `CollectEA` over `CoverageImpact` justifies its

[4]Note that there are two entries for each version of JABA because of the separation between short and long tests.

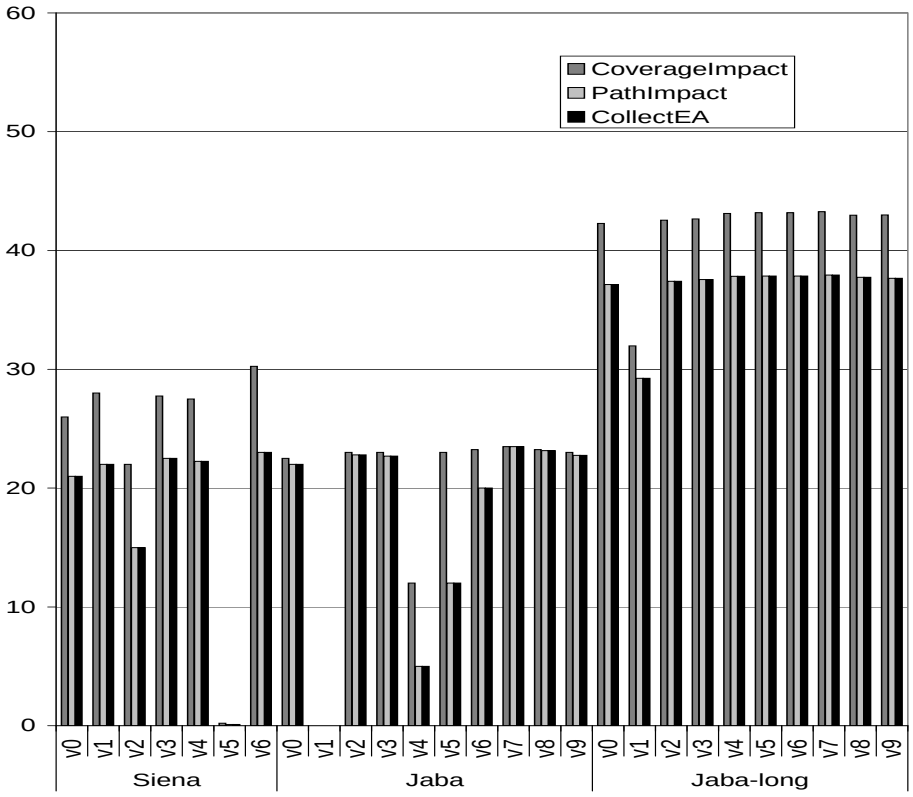


Figure 20: Precision results, expressed as percentage of methods in the impact sets.

use.

4.5.4 Study 3: Field Data version In-house Data

The goal of this study is to assess whether using field data, instead of synthetic data, can yield different analysis results. To achieve this goal, we performed a study that compared the results of performing dynamic impact analysis using two data source: field data, referred to as FIELD, and in-house data, referred to as IN-HOUSE.

The study used Java Architecture for Bytecode Analysis (JABA), a framework for analyzing Java programs. To collect field data, we instrumented JABA for method coverage information and released it to eleven users who agreed to have information collected during execution. Five of the eleven users had already used JABA for their work whereas the other six users had just started projects that involved the use of JABA. Seven of eleven users involved in the studies are working in the Aristotle research lab: four are part of the Aristotle research group and use JABA for their research; another two are students working

Table 7: Number of methods changed in the sets of real versions considered

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
15	3	15	6	3	2	3	178	95	12
C_{11}	C_{12}	C_{13}	C_{14}	C_{15}	C_{16}	C_{17}	C_{18}	C_{19}	C_{20}
87	28	6	61	22	2	61	5	6	89

in College of Computing who use JABA for two graduate-level projects; the last one is a Ph.D. student who is using a regression testing tool built on top of JABA. The remaining four users are three researchers and a student working in three different universities, one of which is abroad.

To instrument and collect the data, we used the GAMMATELLA tool [57]. When instrumenting, the tool also includes in the program the network-communication code that is used to send data back to a central server. On the server side, the tool performs both the data-collection and data-storage tasks. Using GAMMATELLA, we gathered data for ten weeks, during which approximately 1,100 executions are collected.

The in-house data are collected from the executions of the regression test suite that have been developed for JABA over the years.

This study used a set of real changes made to JABA by extracting 21 versions of JABA during the period of seven months from its CVS repository. For each (version, subsequent-version) pair (v_i,v_{i+1}) of JABA, we identified the changes between the two versions and, for each change, (1) mapped it to the method m containing the change, and (2) added m to the set of changes C_i. The resulting sets of 20 changes, C_1 to C_{20}, are the sets used for this study.

Table 7 shows the number of methods changed for each of the 20 sets. As the table shows, the number of methods changed ranges from a minimum of 2, for change sets C_6 and C_{16}, to a maximum of 178, for change set C_8.

The results of computing the impact sets using the two data source are shown in Table 8. The table reports a number of measures. FL and IH are the sizes of the impact sets computed using FIELD and IN-HOUSE data sources, respectively. FL/IH is the ratio of the size of the impact set computed using FIELD data source to the size of the impact set

Table 8: Results for the comparison of FIELD (FL) and IN-HOUSE (IH) data source on real changes

C	FL	IH	FL/IH	$FL\text{-}IH$	$IH\text{-}FL$
C_1	776	784	0.99	96	104
C_2	778	771	1.01	116	109
C_3	778	784	0.99	110	116
C_4	780	778	1.00	112	110
C_5	617	617	1.00	0	0
C_6	750	765	0.98	86	101
C_7	791	796	0.99	97	102
C_8	806	794	1.02	126	114
C_9	822	785	1.05	139	102
C_{10}	789	800	0.99	111	122
C_{11}	737	766	0.96	68	97
C_{12}	802	797	1.01	113	108
C_{13}	805	788	1.02	120	103
C_{14}	797	784	1.02	122	109
C_{15}	773	751	1.03	127	105
C_{16}	0	0	1.00	0	0
C_{17}	790	767	1.03	130	107
C_{18}	753	759	0.99	98	104
C_{19}	763	761	1.00	99	97
C_{20}	819	793	1.03	131	105
AVG	736.30	732.00	1.01	100.05	95.75
STD	174.11	172.14	0.02	37.22	27.37
MAX	822	800	1.05	139	122

computed using IN-HOUSE data source; $FL\text{-}IH$ is the set difference between the impact set computed using FIELD data source and the impact set computed using IN-HOUSE data source (i.e., the number of methods that are considered affected when using FIELD data source, but are not considered affected when using IN-HOUSE data source). $IH\text{-}FL$ is defined analogously.

The data in Table 8 clearly show that the results of the analysis are affected significantly by the data sources considered. For example, 18 of the 20 changes (all but C_5 and C_{16}) result in a significant number of methods (68-139) included in the impact set computed using FIELD data source but not in the impact set computed using IN-HOUSE data source and vice versa (97-122). Also in this case, both FIELD and IN-HOUSE data source yield on average fairly dissimilar impact sets.

Considering Table 8, note that, for all change sets considered, the sizes of the impact sets computed using FIELD and IN-HOUSE data source are almost identical—what differs is the composition of those sets. Note also that the above results are not due to the fact that the sets of entities covered by the in-house test suite and by field executions are mostly disjoint. In fact, the internal test suite and the field executions both cover approximately 65% of the code and their coverage sets have an 85% overlap.

CHAPTER 5

TESTING-REQUIREMENTS IDENTIFICATION

With the availability of change information provided by the differencing activity and impact information provided by the dynamic-impact-analysis activity (see Figure 1), the testing-requirements-identification activity can proceed. This activity takes as input the original program version, P, and a modified version, P', and the change and impact information. The identification activity computes testing-requirements that can be used to assess the quality of test suites with respect to testing the changes between P and P' and guide the generation of new test inputs targeting the changes.

5.1 Related Work

Several existing techniques that are related to assessing the quality of test suites with respect to changes and guiding the generation of new test cases are presented in the literature. A first class of techniques computes testing requirements for whole programs in terms of program entities: control-flow entities or data-flow entities (e.g.,[26, 43, 53]). Among these techniques, the most closely related to the new technique is Ntafos's required-elements (k-tuples of definition-use associations) [53]. The technique identifies the chain of length k of def-use associations in a program. The techniques in this class do not focus on changes and treat all program entities equally. Therefore, using these techniques, software testers need to test the parts of the software that are not affected by changes. In addition, these technique require only that program entities are executed, which may not be adequate to reveal different behaviors caused by changes.

A second class of related techniques incorporates into their testing requirements conditions under which faults can propagate. In their RELAY framework, Richardson and Thompson [62] describe a precise set of conditions for the propagation of faults to the output. Morell [48] builds a theory of fault-based testing by using symbolic execution to determine fault-propagation equations. These techniques do not target changed software

and, moreover, rely on symbolic execution of an entire program, which is impractical for large software. The technique developed in this research, in contrast, constrains complexity by limiting the generation of testing requirements to preselected distances from changes and incorporating conditions that guarantee propagation up to those distances.

A third class of related techniques augments existing test suites to strengthen their fault-revealing capability. Harder and colleagues [30] introduce operational coverage, a technique based on a model of the behavior of methods. Whenever a candidate test case refines an operational abstraction (i.e., invariant) of a method, the candidate is added to the test suite. Bowring and colleagues [9] present another behavior-based technique that builds a classifier for test cases using stochastic models describing normal executions. These techniques do not provide a criterion for quality assessment of test suites, but rather a means to classify and group test cases. Moreover, they do not perform any kind of change-impact propagation. Overall, these techniques are mostly complementary to techniques for assessing the quality of test suites with respect to changes.

A fourth class of related techniques shares the goal of creating testing requirements based on program changes. Binkley [7] and Rothermel and Harrold [63] use system dependence graph (SDG) based slicing [35] to select testing requirements based on data- and control-flow relations involving changes. SDG-based techniques typically do not scale due to the memory and processing costs of computing summary edges [5]. Gupta and colleagues [29] propose a technique that is also based on slicing, but uses an on-demand version of Weiser's slicing algorithm [76] and avoids the costs associated with building SDGs. Their technique computes chains of control and data dependences from the change to output statements, which may include a considerable part of the program and are likely to be difficult to satisfy. Moreover, our preliminary studies show that considering the effects of changes on the control- and data-flow alone, as these techniques propose, is often not sufficient for exercising the effects of the changes on the software. Even when test cases exercise the *output-influencing* data-flow relationships from the point of change(s) [21], the modified behavior of the software may not be exercised.

```
      Program P

  public class BookRecommender { ...
    public static void main(String[] args) { ...
s1    Book recBook = bookfinder.getRecommendedBook();
s2    int numpages = (recBook == null)? 0:recBook.getNumPages();
      ...
    }
    ...
  }

  public class BookFinder { ...
    public Book getRecommendedBook() { ...
s3    for(Book book : library.getBooks()) {
s4      if(!(book instanceof     // s4'  if (book.category
          ==AudioBook)         //            ==Book.NONFICTION)
s5        return book;
      }
s6    return null;
    }

  public class Book {
    public static final int NONFICTION = 1;
    public static final int CHILDREN = 2;
    ...
  }

  public class AudioBook extends Book { ... }
```

Figure 21: Partial code for the original version P with a change at $s4$.

5.1.1 Motivating Example

To illustrate the inadequacy of the criteria based solely on control and data flow, consider the example in Figure 21, which shows the original program version, P, and an alternative version of statement $s4$ in P, $s4'$. This alternative version is used to construct a modified version of P, called P'. The *Bookfinder.getRecommendedBook* in P returns the first instance of class *Book* that is not of type *AudioBook* or *null* if all books are audio books. The change at $s4'$ in P' causes method *Bookfinder.getRecommendBook* to return a book whose catagory is $NONFICTION$. If the non-fiction book returned is of type *AudioBook*, the call to *recBook.getNumPages* will throw an unsupported operation exception (because an audio book does not have pages). A technique that tests the changed program by rerunning all test cases that traverse the change would generally not reveal a

regression error introduced by the change (unsupported operation exception at statement $s2$). Even a technique that exercises data-flow relationships from the point of change to an output would be unlikely to reveal the problem. The only way to exercise the change at $s4$ suitably is to require that a non-fiction book of type *AudioBook* is returned from method $BookFinder.getRecommendedBook()$.

The problem is that changes in the software affect, after their execution, the state and the control-flow of the software, but these effects often manifest themselves only under specific conditions. Therefore, criteria that simply require the coverage of program entities (e.g., statements and data-flow relationships) are usually inadequate for assessing the quality of test suites, in that they may overestimate the adequacy of test suites with respect to program changes. These criteria are satisfied when all required program entities are executed even though the test suites do not reveal different behaviors and, thus, are of limited use in guiding test-case generation. To account for this limitation, techniques must incorporate a means to model program states and compare them to obtain the necessary conditions for different behavior between the two versions. However, state-modeling techniques usually have high complexity, incur high costs in both space and time, and do not scale when applied to whole programs.

5.1.2 Overview of the Approach

The new technique developed in this research addresses the shortcomings of existing techniques by identifying testing requirements, which form testing criteria, that guarantee that the test suites satisfying them, when executed on the modified version, will result in different control flow or different program states at selected program points in the original and modified versions. The new technique provides this guarantee by leveraging symbolic execution, which precisely models program states. To control the overall cost of this approach, the new technique applies symbolic execution to only parts of the program that were modified or affected by the modifications. More precisely, the technique performs symbolic execution from change points to selected program points based on the distance from changes in terms of control and data dependences and, thus, limits the number of statements to be analyzed.

The empirical studies of this research show that considering program points that are only a short distance away from changes can be sufficient for building testing requirements that are effective in revealing different behavior while making the overall approach practical.

This technique involves several steps. First, it uses the change information to identify affected statements in the new version (P') of a program at the preselected distances using forward-dependence analysis. Second, it uses the mappings between statements in the old version (P) and the new version (P') to identify statements in P corresponding to affected statements computed in the first step. Third, it uses symbolic execution to compute, for statements that are executed after the changed statements in P and P', a path condition and a symbolic state. Fourth, it compares path conditions and symbolic states of corresponding statements in P and P' and defines testing requirements based on the results of the comparison. Fifth, it instruments P' to assess, during regression testing, the extent to which a test suite satisfies these testing requirements. Finally, based on the set of unsatisfied testing requirements, the technique provides guidance to the tester for the development of new test cases.

5.2 Testing Requirements Computation and Checking

This section provides details of the approach developed in this research for assessing the quality of test suites with respect to changes and guiding the generation of new test cases targeting the changes. Section 5.2.1 describes the change-based criteria for testing modified programs. Section 5.2.2 presents the algorithm to compute testing requirements for a single change that form this change-based criteria, Section 5.2.3 discusses how the requirements can be checked, and Section 5.2.4 discusses an extension of the algorithm to handle multiple changes.

5.2.1 Change-based Criteria

Ideally, a criterion for testing a modified program P' should guarantee that the effects of the changes that can propagate to the output actually propagate, so that such effects will be revealed. Such an approach can be seen as an application of the PIE model [73] to the case of changed software: the criterion should ensure that the change is executed (E), that it

infects the state (I), and that the infected state is propagated to the output (P). However, generating testing requirements for such a criterion entails analyzing the execution of P and P' (e.g., using symbolic execution) from the point of the change until the end of the program, which is impractical for any non-trivial program.

A more practical approach than the one described above that can still be effective is to define a set of criteria, each of which ensures that the executions of P and P' results in different states after executing statements at a specific distance from the change (i.e., it ensures that the effects of the change have propagated at least to these statements). The distance is expressed in terms of data- and control-dependence chains, rather than control flow because the effects of changes propagate along the dependence chains. The definition of distance is based on the two means the effects of changes may propagate. First, affected variables (i.e., variables whose values are affected by the changes) may be used in defining other variables and, thus, propagate the effects to those variables (data dependence). Second, affected variables may be used in predicates causing the control flow to continue on different paths (control dependence). The diverging control flow, in turn, may indirectly cause values of variables to differ at statements after the two paths meet because variables may be assigned different values in the true or false branch. Based on these observations, statements at distance 1 from a statement include the closest statement along any data- and control-dependence chain from the originating statement where the effects of changes may manifest themselves. More specifically, given an assignment statement $s1$, statements at distance 1 from $s1$ include all statements that are data-dependent on $s1$. Given a branching statement $s2$, its affected statements include all statements that (1) are data dependent on any statement that is control dependent on $s2$ and (2) are not control-dependent on $s2$. For example, consider the code fragment in Figure 22 with an alternative version of statement $s2$ ($s2'$), which can be used to generate a modified version. The only statement that may expose different behavior in the modified version is statement $s5$ because the change may cause the control flow to take the true branch of statement $s2$ in one version and the false branch in the other version. Thus, the value of x at statement $s5$ could be 1 in one version and 0 in the other version. Without the constraint that the affected statements

```
s1    x = 0;
s2    if ( a > 0 ) { // s2'   if ( a > 1 ) {
s3       x = 1;
s4       y = x;
      }
s5       z = x;
```

Figure 22: A code fragment for illustrating the identification of statements at distance 1 from a branching statement

must not control dependent on the branching statement, the set of affected statements at distance 1 would include statement $s4$. However, $s4$ could not expose different behavior because, if the executions in both versions reach $s4$, the value of x in both executions will be the same (i.e., x equals 1).

Statements at distance d are computed recursively as statements that are at distance 1 from a statement at distance $d-1$. Note that a statement can be at distance m and distance n at the same time. Such a statement is considered to be at distance $min(m, n)$. Note also that the change itself is considered to have *distance* 0 (i.e., requirements defined for distance 0 refer to the state immediately after the changed statement is executed). For example, consider the change at statement $s4'$ in P' (Figure 21). Statements at distance 1 from $s4'$ include the assignment statement $s1$, which contains a call to $BookFinder.getRecommendedBook()$ (because the control flow of both versions merge at method return). Other statements that contain calls to this method are also included in the set of statements at distance 1 from $s4'$.

This distance provides the tester a way to balance effectiveness and efficiency of the criterion. On the one hand, criteria involving greater distances are more expensive to compute and satisfy than those involving shorter distances. On the other hand, criteria involving greater distances ensure that states farther away from the change differ and, thus, that the effects of the change have propagated *at least* to those points in the program. Intuitively, requiring this propagation increases the likelihood that different (possibly erroneous) behaviors due to the changes will be exercised and revealed. Note that, in some cases, it may not be possible to propagate the effects of a change beyond a given distance d. For example,

imagine a change in the way an array is sorted; after the sorting is done, the states should be exactly the same in P and P'. In these cases, there would be no testing requirements involving a distance d or higher because no different states in P and P' could be generated. We discuss this aspect in more detail in Section 5.2.2, where we present our algorithm for computing testing requirements.

For each change and distance value, the new algorithm generates a set of testing requirements that must be met to satisfy the change-based criterion at that particular distance. The testing requirements are represented as boolean predicates, expressed in terms of constant values and the values of the variables at the point immediately before the change in P'.

5.2.2 Algorithm

The algorithm for computing testing requirements for a change in a program, `ComputeReqs` (shown in Figure 23), takes three groups of inputs: (1) P and P', the original and modified versions of the program, respectively; (2) *change*, a pair (c, c') of statements where c' is modified in, added to, or deleted from P'; and (3) *requested_distance*, the dependence distance for which the testing requirements are generated. This algorithm handles a new (deleted) statement by matching it to a dummy statement in P (P'). If the new (deleted) statement contains a definition of a variable v, then the algorithm adds the dummy statement $v = v$ in P (P').[1] (This is needed to ensure the correct behavior of our algorithm.) Otherwise, the dummy statement is a simple *no-op* (no operation). Analogously, a new (deleted) branching statement is matched to a dummy branching statement in P (P') with the same control dependent regions and predicate *true*. After the dummy statements have been introduced, new and deleted statements are simply treated as modified statements.

`ComputeReqs` outputs *reqs*, a set of testing requirements that must be met to satisfy the criterion at the requested distance. `ComputeReqs` uses five external functions: $match(n')$ returns the statement in P that corresponds to n' in P' (This information is provided by

[1] Without loss of generality, we assume that a single statement can define only one variable. Any statement that defines more than one variable can be transformed into a number of statements with one definition each.

procedure `ComputeReqs`

Input: original and modified versions of the program, P and P',respectively
pair (c in P, c' in P') of changed statements, *change*
dependence distance *requested_distance*

Output: set of testing requirements (initially empty), *reqs*

Use: $match(n')$ returns a statement in P that corresponds to n' in P'
$def(n)$ returns the variable defined at statement n if any, or *null*
$FDD(n, d, P)$ returns set of statements dependent on n
$PSE(c, n, P)$ returns program state at n
$TRI(S, S')$ returns set of testing requirements

Declare: sets of pairs of affected statements, *affected* and *next_affected*
statements in P', s' and n'
program states in P and P', S and S', respectively

Begin:

{Step 1: Identify affected parts of P'}

1: *affected*= $\{c'\}$
2: **while** *requested_distance*- - > 0 **do**
3: $next\_affected = \emptyset$
4: **for each** $s' \in$ *affected* **do**
5: $next\_affected = next\_affected \cup \{n' | n' \in FDD(s', def(s'), P')\}$
6: **end for**
7: $affected = next\_affected$
8: **end while**

{Step 2: Compute testing requirements}

9: **for each** $s' \in$ *affected* **do**
10: $S = PSE(c, match(s'), P); S' = PSE(c', s', P')$
11: $reqs = reqs \cup TRI(S, S')$
12: **end for**
13: return $reqs$

end

Figure 23: Algorithm to compute testing requirements.

the differencing algorithm); $def(n)$ returns the variable defined at statement n or *null* if n does not define a variable; $FDD(n, d, P)$, $PSE(c, n, P)$, and $TRI(S, S')$ are explained below.

`ComputeReqs` consists of two main steps: identification of the affected statements in P and P' at the requested distance and computation of the testing requirements, which correspond to the conditions under which the change induces different states, in P and P', after the execution of the affected statements.

In the first step, `ComputeReqs` initializes *affected*, a set of affected statements in P', to the changed statement c' (line 1). For each iteration of the while loop, `ComputeReqs` computes the affected statements one more dependence distance away from the change by computing forward direct dependences for each affected statement at the current distance and adding these dependents to *next_affected* (line 5). *Forward Direct Dependence (FDD)* identifies the statements control-dependent on the input statement s' or data-dependent on the definition of variable $def(s')$ at s'. For example, to compute affected statements at distance 1 from $s4'$ in P' (see Figure 21), `ComputeReqs` calls $FDD(s4', null, P')$, which returns the statement in P' that corresponds to $s1$ in P. After `ComputeReqs` processes each pair of affected statements, it assigns *next_affected* to *affected* and repeats the process until the requested distance is reached.

In the second step, `ComputeReqs` computes the testing requirements for the affected statements identified in the first step. To do this, at each statement s' in P' (resp., $match(s')$ in P), `ComputeReqs` uses partial symbolic execution to identify the path conditions and symbolic states of s' (resp., $match(s')$). *Partial Symbolic Execution (PSE)* is similar to global symbolic execution [14], except that the changed statement c is the starting point, all live variables at the changed statement are input variables, and s (resp., s') is the ending point. PSE differs from global symbolic execution in two respects. First, rather than considering all paths from program inputs to program outputs, PSE considers only finite subpaths from the change to s (resp., s') along dependence chains up to the desired distance. Second, instead of representing path conditions and symbolic states in terms of input variables, PSE expresses them in terms of constant values and program variables

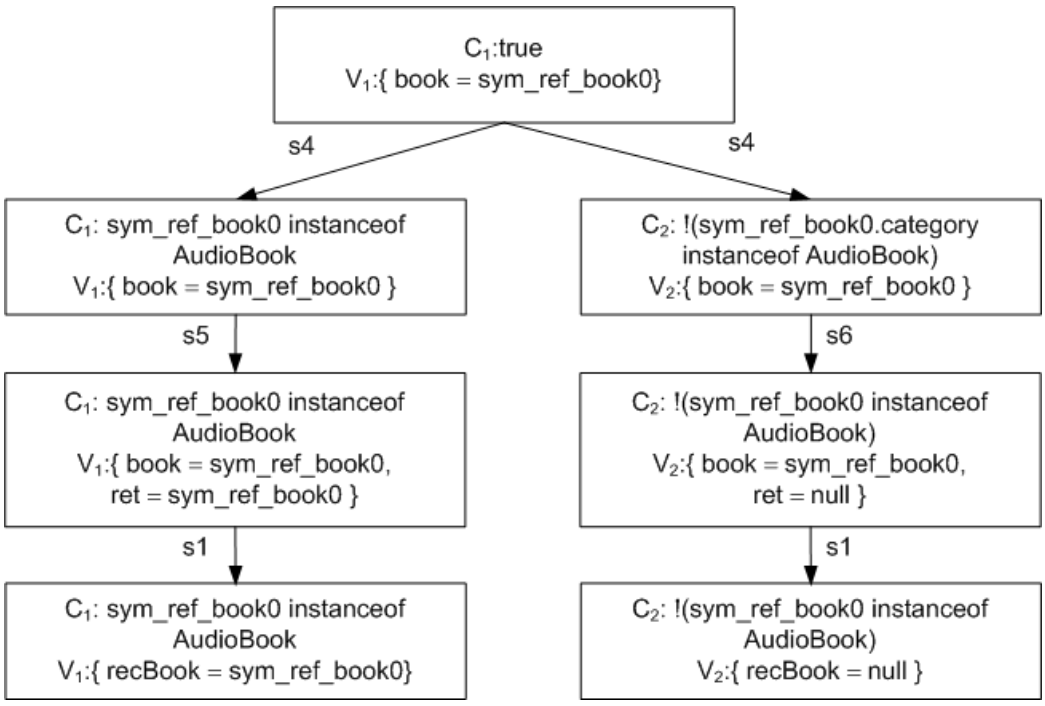


Figure 24: Symbolic execution tree for $s4$-$s1$ in P.

representing the state of the program at the point immediately before the change. Analogous to global symbolic execution, PSE represents program states with case expressions. More formally, the program state at statement s, is defined as the set $\{(C_{s,i} : V_{s,i}) | i \geq 1\}$, where $C_{s,i}$ is the path condition to reach statement s from the change through path i, and $V_{s,i}$ is the symbolic state when $C_{s,i}$ holds. A symbolic state is represented as a set of variable assignments, $V_{s,i} = \{v_1 \leftarrow e_1, v_2 \leftarrow e_2, ...\}$, where e_i is the value of v_i expressed symbolically. ($v_i \leftarrow e_1$ means that the value of variable v_i is expression e_i.)

For the previous example, $PSE(s4, s1, P)$ evaluates $s1$ in terms of sym_ref_book0, the value of variable *book* at the point immediately before $s4$, on two paths: $(s4, s5, s1)$ and $(s4, s6, s1)$. Figure 24 illustrates the symbolic execution tree from $s4$ to $s1$. Each rectangle represents a state in P, and each edge, labeled with a statement, represents the transformation from one state to another when that statement is executed. From the tree, PSE returns $\{(sym\_ref\_book0 instanceof AudioBook : \{recBook = sym\_ref\_book0\}), (!(sym\_ref\_book0 instanceof AudioBook) : \{recBook = null\})\}$

After each pair of affected statements at the requested distance is symbolically executed,

`ComputeReqs` calls TRI, which compares each statement and its counterpart in terms of their path conditions and symbolic states to identify testing requirements. For each pair of corresponding statements (s, s') in P and P', TRI produces a number of testing requirements. These requirements guarantee that test inputs satisfying them would result in different states after executing s'. For an assignment statement, program states are different when the variable defined there has different values. Using the representation of a program state, this condition can be formulated as follows. Let $S_s = \{(C_{s,i} : V_{s,i}) | 1 \leq i \leq m\}$ and $S_{s'} = \{(C_{s',j} : V_{s',j}) | 1 \leq j \leq n\}$ be the program states after executing statements s and s' in P and P', respectively, where $V_{s,i} = \{v_1 \leftarrow e_1, v_2 \leftarrow e_2, ...\}$, $V_{s',j} = \{v'_1 \leftarrow e'_1, v'_2 \leftarrow e'_2, ...\}$ and each variable v_i in P corresponds to variable v'_i in P'. In the following, for simplicity, we abbreviate X_s with X and $X_{s'}$ with X' for any entity X. The condition above can be expressed as a set of requirements: $\forall i, j, k\{(C_i \wedge C'_j) \wedge (e_k \neq e'_k) | v_k \leftarrow e_k, v'_k \leftarrow e'_k\}$. For a branching statement, program states are different when the control flow diverges. More formally, the condition can be expressed as $\forall i, j\{(C_i \wedge C'_j) \wedge (p' \wedge \neg p)\}$, for the true branch, and $\forall i, j\{(C_i \wedge C'_j) \wedge (\neg p' \wedge p)\}$, for the false branch, where p' and p are the predicates at s' and s, respectively, where each variable v_i (resp., v'_i) uses in p (resp., p') is substituted with e_i (resp., e'_i), given $(v_i \leftarrow e_i)$ is in V_i and $(v'_i \leftarrow e'_i)$ is in V'_i

Software testers may choose to satisfy only one of the requirements generated for each affected statement or to satisfy all of these requirements. If one of the requirements is met, the statement corresponding to the requirements will expose one different behavior. However, that behavior may not lead to different output. If all requirements are met, there are more chances that some different behaviors will lead to different output.

Note that, to measure coverage of the generated requirements, the requirements need not be simplified or solved. Checking whether a test case satisfies any of these requirements can be performed by substituting each variable in a requirement with its concrete value (obtained during the execution of the test case at the point immediately before the change) and evaluating the truth value of the requirement. Simplification and constraint solving are necessary only if we want to use the requirements to guide test-case generation or determine the requirements' feasibility.

Table 9: Path conditions and symbolic states from s4-s1 of P and P'

stmt	C	V
s3	C_1: True	$V_1 : \{book = sym\_ref\_book0\}$
s3'	C'_1: True	$V'_1 : \{book = sym\_ref\_book0\}$
s4	$C_1 : sym\_ref\_book0 instanceof AudioBook$	$V_1 : \{book = sym\_ref\_book0\}$
	$C_2 : !(sym\_ref\_book0.category instanceof AudioBook)$	$V_2 : \{book = sym\_ref\_book0\}$
s4'	$C'_1 : sym\_ref\_book0.category = NONFICTION$	$V'_1 : \{book = sym\_ref\_book0\}$
	$C'_2 : sym\_ref\_book0.category \neq NONFICTION$	$V'_2 : \{book = sym\_ref\_book0\}$
s5	$C_1 : sym\_ref\_book0 instanceof AudioBook$	$V_1 : \{book = sym\_ref\_book0,$ $ref = sym\_ref\_book0\}$
s5'	$C'_1 : sym\_ref\_book0.category = NONFICTION$	$V'_1 : \{book = sym\_ref\_book0,$ $ref = sym\_ref\_book0\}$
s6	$C_1 : !(sym\_ref\_book0.category instanceof AudioBook$	$V_1 : \{book = sym\_ref\_book0,$ $ref = null\}$
s6'	$C'_1 : sym\_ref\_book0.category \neq NONFICTION$	$V'_1 : \{book = sym\_ref\_book0,$ $ref = null\}$
s1	$C_1 : sym\_ref\_book0 instanceof AudioBook$	$V_1 : \{recBook = sym\_ref\_book0\}$
	$C_2 : !(sym\_ref\_book0 instanceof AudioBook)$	$V_2 : \{recBook = null\}$
s1'	$C'_1 : sym\_ref\_book0.category = NONFICTION$	$V'_1 : \{recBook = sym\_ref\_book0\}$
	$C'_2 : sym\_ref\_book0.category \neq NONFICTION$	$V'_2 : \{recBook = null\}$

As discussed above, there are changes whose effects do not propagate beyond a certain distance (see the array-sorting example provided in Section 5.2.1). In these cases, if the constraints corresponding to conditions (1) and (2) can be solved, they evaluate to *false*, which means that the corresponding requirements are unsatisfiable and testers do not need to further test the effects of that change.

For the example in Figure 21, the path conditions and symbolic states of $s4$-$s1$ in P and P' are shown in Table 9. A statement in P' that corresponds to an unchanged statement, s_n in P is referred to as s'_n. When identifying testing requirements at distance 1, `ComputeReqs` computes the requirements necessary for revealing different states at $s1$ and $s1'$ by calling $TRI(S_{s1}, S'_{s1'})$. The testing requirement is

$((C_1 \wedge C'_1 \wedge \{recBook_1 \neq recBook'_1\}) \vee (C_1 \wedge C'_2 \wedge \{recBook_1 \neq recBook'_2\}) \vee$

$(C_2 \wedge C'_1 \wedge \{recBook_2 \neq recBook'_1\}) \vee (C_2 \wedge C'_2 \wedge \{recBook_2 \neq recBook'_2\}))$, where $recBook_n$ and $recBook'_n$ refers to the value of variable $recBook$ along path n in P and P', respectively.

This testing requirement can be simplified to

$$(!(sym\_ref\_book0.category instanceof AudioBook) \wedge sym\_ref\_book0.category = NONFICTION) \wedge (sym\_ref\_book0 \neq null)$$

Now that the algorithm for computing the testing requirements is described. The next section (Section 5.2.3) explains an approach for checking these requirements when the modified version of software is executed on concrete test inputs.

5.2.3 Checking Testing-requirements

When testing the modified version, testers may compare the outputs of a pair of executions on the original and modified versions. The results of this comparison are useful only if the inputs of those executions are the same. Analogously, as described in Section 5.2.2, `ComputeReqs` generates testing requirements by comparing program states at corresponding points in the original and modified versions. The testing requirements are represented as boolean predicates in terms of variables' values at the point immediately before a change. The testing requirements can thus be checked only when all variables present in the requirements have the same values as their counterparts in the other version. For a pair of executions on the original and modified versions of a deterministic program, the program states at the points immediately before the change are obviously the same the first time the change is executed. This condition may not be true the other times the change is executed because the execution of the change may affect the program states or paths in the modified version.

To ensure that the program states be the same at the time of requirements checking, the checker uses two complimentary approaches. The checker first computes static, forward dependence analysis from the change until the end of the program a priori to determine whether the change depends on itself. (This static analysis need not be extremely precise and, thus, can scale to large programs.) If the analysis indicates that the change does not depend on itself (i.e., an execution of the change cannot affect the program states the next times the change is executed), the testing requirements can be checked every time the change is executed; otherwise, the checker uses dynamic, forward dependence analysis to approximate whether the program states are affected by an execution of the change at

runtime. Every time the testing requirements are checked during an execution, the checker examines whether any requirement is satisfied. If none of the requirements are met, the effects of the change certainly do not propagate. The program states will thus be the same the next time the change is executed, and the requirements can be checked again. However, if at least one of the requirements is satisfied, the effects of the change will propagate. The checker approximates the effects using dynamic, forward dependence analysis (e.g., dynamic tainting [15]). During an execution, dynamic, forward dependence analysis identifies the variables whose values may depend on a given statement. This dependence analysis thus can provide a safe approximation of variables that may be affected by a change. By using the results of this analysis, the checker can safely check the testing requirements that do not contain any of the affected variables when the change is executed the next time.

To handle multiple changes, this research extends both the techniques for computing and checking testing requirements . The next section (Section 5.2.4) discusses the extensions to both techniques.

5.2.4 Multiple Changes

When multiple changes have been made to a program, one change may affect the others. Furthermore, some statements in the rest of the program may be affected by two or more changes even though those changes do not affect one another. The testing requirements generated by `ComputeReqs` for each change can guarantee different behavior for test inputs satisfying them in the presence of multiple changes because PSE computes program states by analyzing all other changes (and their effects) along the paths from the change considered to each affected point. Therefore, `ComputeReqs` can generate the testing requirements for multiple changes by taking a union of sets of requirements generated for each individual change in the modified version. For example, consider a code fragment in Figure 25. This example includes an alternative version for each of statements $s2$ and $s3$ (i.e., $s2'$ and $s3'$, resp.), which is used to construct a modified version. Given the specified distance be two, the set of affected statements at this distance from $s2$ contains only statement $s5$ because $s5$ is at distance 1 from $s3$, which, in turn, is at distance 1 from $s2$. To generate testing

```
s1    if ( b > 0 )
s2      a = 0;          // s2'    a = 2;
s3    if ( a > 0 )      // s3'    if ( a > 1 )
s4      x = 1;
s5    y = x + 1;
s6    z = y + y;
```

Figure 25: A code fragment for illustrating the identification of statements at distance 2 in the presence of multiple changes

Table 10: Path conditions and symbolic states from $s2$-$s5$ in the original and modified versions of the code fragment in Figure 25

stmt	C	V
s2	C_1: True	$V_1 : \{a = 0; x = x_0; y = y_0\}$
s2'	C'_1: True	$V'_1 : \{a = 2; x = x_0; y = y_0\}$
s3	$C_1 : 0 > 0$	$V_1 : \{a = 0; x = x_0; y = y_0\}$
	$C_2 : 0 \leq 0$	$V_1 : \{a = 2; x = x_0; y = y_0\}$
s3'	$C'_1 : 2 > 0$	$V'_1 : \{a = 2; x = x_0; y = y_0\}$
	$C'_2 : 2 \leq 0$	$V'_2 : \{a = 2; x = x_0; y = y_0\}$
s4	N/A	N/A
s4'	C'_1: True	$V'_1 : \{a = 2; x = 1; y = y_0\}$
s5	C_2: True	$V_1 : \{a = 0; x = x_0; y = x_0 + 1\}$
s5'	C'_1: True	$V'_1 : \{a = 2; x = 1; y = 2\}$

requirements for the change at $s2'$, the requirements identifier calls PSE to compute program states at $s5$ in the the modified version (which includes both changes) and the original one.

Table 10 shows the results of running PSE on the original and modified versions from statement $s2$ to $s5$. As in the previous example, a statement in the modified version that corresponds to an unchanged statement s_n in the original version is referred to as s'_n. The path conditions and symbolic states of unreachable paths are removed from the table. According to this table, the requirements identifier generates three requirements (one for each variable): $(True \wedge True \wedge (2 \neq 0))$, $(True \wedge True \wedge (x_0 \neq 1))$, and $(True \wedge True \wedge (x_0 + 1 \neq 2))$. The first requirement is trivially true, and the second and third requirements can be simplified to the same requirement: $(x_0 \neq 1)$. These requirements reflect the necessary condition for the propagation of the impact from statement $s2'$ to $s5'$ in the presence of both changes.

However, the effects of several changes that are analyzed together in PSE may not be sufficiently tested because the distance is computed from the first change encountered. For the previous example, statement $s5$ is at distance 2 from statement $s2$ and at distance 1 from $s3$. If the two changes are considered as a whole, statement $s5$ should be included in the set of affected statements at distance 1 for this combined change. `ComputeReqs` thus redefines the computation of the distance for an affected statement by using the distance from the last change encountered along the dependence chain. Therefore, for this example, the set of statements at distance 2 from the combined change ($s2'$ and $s3'$) includes statement $s6'$ because $s6'$ is at distance 2 from $s3'$, which is the last change along the dependence chain.

Each change that is analyzed in the contexts of other changes has to be analyzed in its own context because there may be a feasible path from the beginning of the program to the change without passing through any other changes. In the previous example, $s3'$ has to be analyzed independently from $s2'$ because $s3'$ can be reached through the false branch of $s1'$.

The testing-requirements checker also needs to be extended. The static, forward dependence analysis must determine whether the variables present in a testing requirement at a change depend on any other changes. If the analysis indicates that a change at statement s_a does not depend on another change at statement s_b, the requirements generated for the change at s_a can be checked regardless of the number of times the change at s_b is executed. In the example shown in Figure 25, the change at $s3'$ is dependent on the change at $s2'$. Thus, the requirements at $s3'$ can be checked only if statement $s2'$ has not been executed or the dynamic dependence analysis indicates that the requirements can safely be checked. During an execution, the testing requirements can be checked without any restrictions until at least one of the requirements at any change is satisfied. After that, the checker needs to determine whether any variables present in the testing requirements are affected by the changes associated with the satisfied requirements. The requirements are checked only if they contain no affected variables. In the previous example, after statement $s2'$ is executed, the testing requirements at statement $s3'$ can be checked only if the requirements do not contain the affected variable a. (Because one of the requirements at $s2'$ is "True," the

impact of the change always propagate.)

The extension to handle multiple changes makes the techniques for computing and checking the testing requirements applicable to a wide range of changes occurred during software maintenance. However, the number of statements that need to be analyzed in the computation of testing requirements at each distance grows with the number of changes. Another approach one can applied to limit the number of statements to be analyzed without greatly reducing the effectiveness of the generated requirements is to focus on the parts of software that are exercised and likely affected by the changes in a set of executions of interest (e.g., executions by end users). Therefore, Section 5.2.5 describes the integration of testing-requirements identification and dynamic impact analysis, which identifies such affected parts.

5.2.5 Integration with Dynamic Impact Analysis

`ComputeReqs` can use the results of dynamic impact analysis when identifying the affected points. Each time after calling FDD, `ComputeReqs` checks each statement in the results of FDD whether its containing method is in the resulting set of dynamic impact analysis (i.e., whether the method is identified as likely affected by the changes in the set of executions of interest). Only the statements whose methods are in the resulting set of dynamic impact analysis are used in the next iteration or by the partial symbolic execution.

This integration approach computes a more precise set of affected points than a simple approach that computes the affected points by first computing the set of affected points at the specified distance before checking whether their containing methods are in the resulting set of dynamic impact analysis. The precision loss comes from the use of statements at short distances whose methods are not in the resulting set of dynamic impact analysis to compute the affected points at greater distances. Furthermore, Our integration approach may not have much higher cost (and can even be lower in some cases) than the simple approach because the number of statements that need to be considered in the computation of the affected points at greater distances may be reduced.

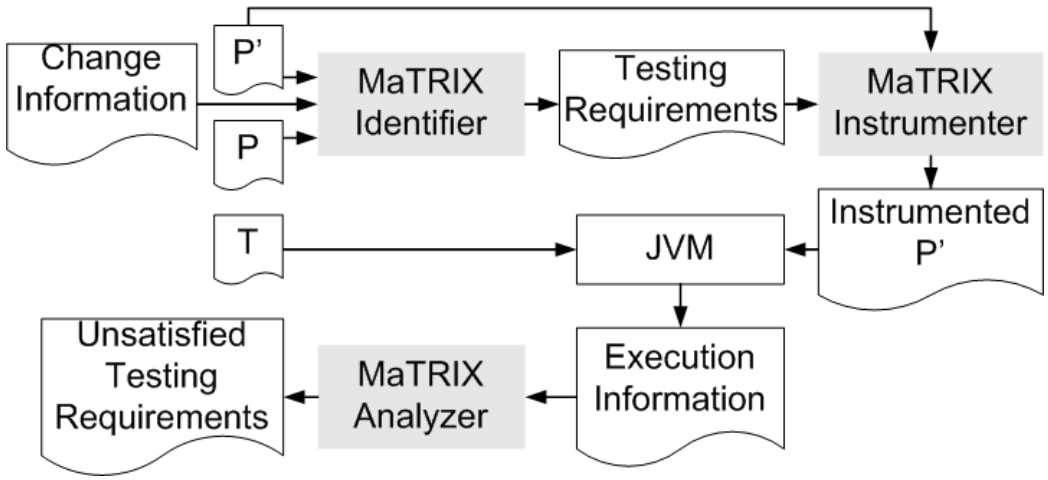


Figure 26: MaTRIX tool set.

5.3 MaTRIX: a Testing-requirements-identification Tool

The MaTRIX (**Ma**intenance-based **T**esting **R**equirements **I**dentifier and e**X**aminer) tool set (shown in Figure 26) implements the three main components of the technique for assessing the quality of test suites and guiding test-input generation: the MaTRIX Identifier identifies testing requirements related to the change from P to P'; the MaTRIX Instrumenter instruments P' so that, when it executes, it will record which testing requirements are satisfied; and the MaTRIX Analyzer examines the recorded information to determine which testing requirements have not been satisfied.

The MaTRIX Identifier, shown in Figure 27 comprises of three main parts: the forward-direct-dependence analyzer; the partial-symbolic-execution engine, which includes the partial symbolic execution (PSE) transformer and the Java PathFinder [72] Virtual Machine (JPF VM); and the testing-requirements constructor. The forward-direct-dependence analyzer is a Java program implemented on top of the static-analyses module of Indus [39]. The analyzer computes both intra- and inter-procedural control- and data-dependence for both local variables and fields. The partial-symbolic-execution engine relies on the virtual machine of Java PathFinder as a symbolic-execution engine [2]. However, to perform symbolic execution on a program using the JPF VM, the tool needs to transform the program, such that the program can operate on symbolic values. Anand and colleagues [3] propose

a technique and a tool (JPF Symbolic Transformer) to transform the program automatically for operating on symbolic values. However, their tool does not support partial symbolic execution. The PSE transformer is a series of program instrumentation and transformation modules for producing a "partial" program that can operate on symbolic values. The Isolator isolates the parts of a program that need to be executed symbolically, which include parts of the methods containing changes, parts of the methods containing affected statements at a specified distance, and parts of all the caller methods up to the least common ancestor of the changed and affected methods in the call graph. This code isolation induces uninitialized variables and fields because the definitions of those variables and fields may have been excluded. Hereafter, uninitialized variables and fields are both referred to as "uninitialized variables," unless otherwise noted. The Uninitialized Variable Finder module is an analysis module for identifying these uninitialized variables. The partial program and the uninitialized-variable information is fed into the LazyInit module, which transforms the partial program such that every uninitialized variable of a reference type is initially assigned a symbolic reference and every uninitialized variable of a primitive type a symbolic value. At each statement that contains a reference to an uninitialized variable of a reference type (i.e., an instance method call or a field reference on an object), that statement is transformed such that the variable is passed into the *lazyinit* method before being referenced. The *lazyinit* method implements the *lazier#* lazy initialization algorithm [19, 20]. The LazyInit module performs this transformation recursively on types (and their super types) of all fields whose enclosing classes may be uninitialized. The StateMonitor then instruments the transformed, partial program with code that monitors and outputs program states at the specified distance. This instrumented, partial program can be considered as a "whole" program for JPF Symbolic Transformer. The resulting program is then run on the JPF VM that outputs the program states at the specified distance. When the program states of both P and P' are obtained, the testing-requirements constructor component can compute the testing requirements using the `ComputeReqs` algorithm discussed in Section 5.2.2.

Currently, the set of Java programs that the MaTRIX Identifier can handle is limited

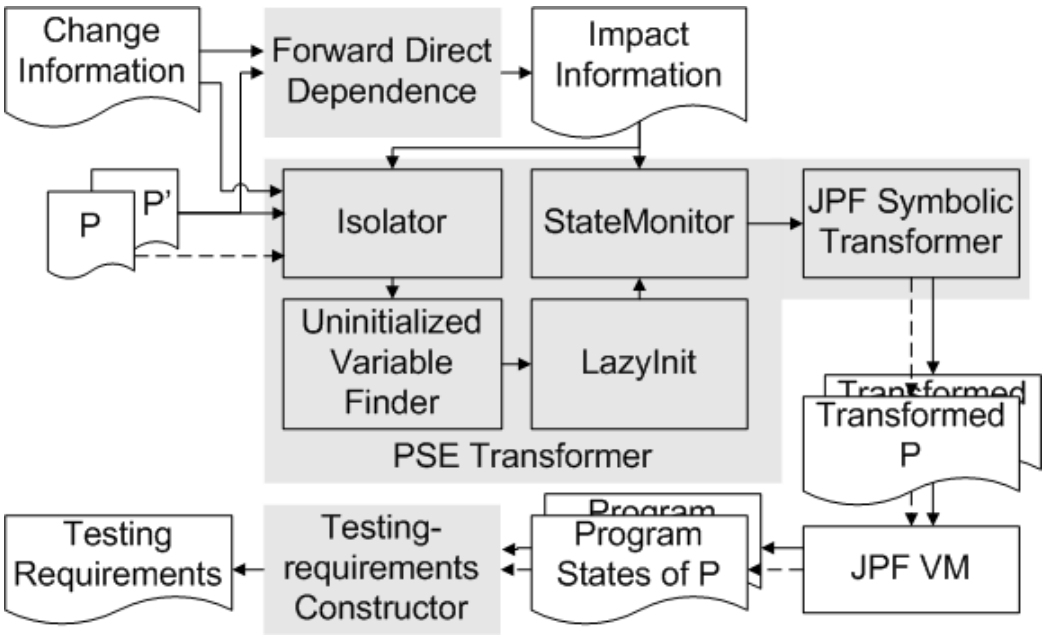


Figure 27: MaTRIX identifier.

by the fragments of logics handled by the symbolic-execution extension to Java PathFinder. This extension uses Yices [22] as its decision procedure; therefore, it can ultimately perform symbolic execution on programs with linear real and integer arithmetic, recursive datatypes, extensional arrays, and fixed-size bit vectors. However, the current interface to Yices implemented by the extension does not support linear real arithmetic and bit-level operations and supports only one-dimentional arrays. Even though Yices can handle constraints involving recursive datatypes, the current implementation uses a separate library to solve such constraints.

The generated requirements are used by two components: the MaTRIX Instrumenter, which instruments the code to collect coverage of our requirements, and the MaTRIX Analyzer, which analyzes the coverage information produced by the instrumented program. Both components are implemented using InsECTJ [68], an instrumentation framework for Java programs.

5.4 Empirical Studies on Testing-requirements Identification

5.4.1 Experimental Setup

We performed two empirical studies to evaluate effectiveness and cost of existing test-adequacy criteria and of our change-based criteria. To evaluate existing criteria, we extended MaTRIX to compute and measure coverage of existing test-adequacy criteria. These studies use as subjects nine versions of two of the Siemens programs [37]: *Tcas* and *Schedule*. We chose *Tcas* and *Schedule*, two small programs, because we wanted to have complete understanding of the subjects' internals to be able to thoroughly inspect and check the results of the studies. Moreover, selecting two small subjects let us use random test case generation to create suitable test suites for the studies. Because *Tcas* and *Schedule* were originally written in C, and MaTRIX tool works on Java programs, we converted all versions of *Tcas* and *Schedule* to Java.

The Java versions of *Tcas* have two classes, 10 methods, and 134 non-comment LOC. The Java versions of *Schedule* have one class, 18 methods, and 268 non-comment LOC. *Schedule* requires some of the C standard library, which results in 102 additional LOC when converted to Java. These studies use one base version (v0) and four modified versions (v1-v4) of *Tcas* and one base version (v0) and five modified versions (v1-v5) of *Schedule*. The changes in the modified versions are faults seeded by Siemens researchers, who deemed the faults realistic based on their experience.

In both studies, we measure the effectiveness of a criterion as the ability of test suites that satisfy the criterion to reveal different behaviors in the old and new versions of a program. To obtain this measure, we first pair a modified version (P') with its base version (P). We then identify the locations of changes between P and P' using JDiff [4] and feed the change information to MaTRIX Identifier to generate a set of testing requirements. We next use MaTRIX Instrumenter to instrument P' based on the generated requirements. Executing the instrumented P' against a test suite generates the information that is used by MaTRIX Analyzer to determine which testing requirements are satisfied by that test suite.

To create coverage adequate test suites for the different criteria considered, we proceeded

as follows. For each modified version of the subject programs and each criterion, we built 50 coverage-adequate test suites by generating random test cases and selecting only test cases that provided additional coverage over those already added. We used a 30-minute time limit for the random generation: if the generator did not create a test input that covered additional testing requirements for 30 minutes, we stopped the process and recorded only the test cases generated thus far. To be able to generate randomly a sufficiently large number of coverage-adequate test suites, we limited the maximum distance to two (i.e., we created test suites for distances zero, one, and two). We measured the effectiveness of a criterion by counting the number of test suites for that criterion that contained at least one test case showing different behaviors in P and P'. As a rough approximation of the cost of a criterion, we used the number of test inputs in the test suites satisfying that criterion.

5.4.1.1 Threats to validity

The main threat to external validity is that these studies are limited to two small subjects. Moreover, these subjects were originally written in C, so they do not use object-oriented features such as inheritance and polymorphism. Therefore, the results may not generalize. Another threat to external validity is that the test suites used in the studies may not be a representative subset of all possible test suites. Threats to internal validity concern possible errors in our implementations that could affect outcomes. Nevertheless, we carefully checked most of our results, thus reducing these threats considerably.

5.4.2 Study 1: Existing Criteria

The goal of this study is to evaluate the effectiveness and cost of existing criteria for testing changes. The test-adequacy criteria we consider are statement and all-uses data-flow criteria. We define these criteria for modified software: the statement adequacy criterion is satisfied if all modified statements are exercised. For the all-uses data-flow adequacy criterion, we expand the criterion into a set of criteria, each of which requires du-pairs up to a specific dependence distance from the changes to be exercised. More precisely, the *all-uses distance-0* criterion requires all du-pairs containing modified definitions to be exercised; and the *all-uses distance-n* criterion requires the du-pairs whose definitions are control- or

Table 11: Percentage of test suites revealing different behaviors over 50 test suites that satisfy the statement adequacy criterion for *Tcas* and *Schedule*.

version	v1	v2	v3	v4
% diff-revealing suites	2	14	22	40

Tcas

version	v1	v2	v3	v4	v5
% diff-revealing suites	0	14	20	10	0

Schedule

Table 12: Percentage of test suites revealing different behaviors over 50 test suites that satisfy all-uses distance-i adequacy criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	0	4	12
v2	6	6	100
v3	18	68	68
v4	80	94	94

Tcas

ver	distance		
	0	1	2
v1	0	0	0
v2	16	30	50
v3	14	30	32
v4	12	30	38
v5	0	0	0

Schedule

data-dependent on the uses of du-pairs at distance $n-1$ to be exercised.

To measure the effectiveness and cost of each criterion, we followed the process described earlier. Note that, for each of the all-uses distance-i criterion, where $i \geq 1$, we built the 50 test suites starting from the test suite satisfying the all-uses distance-$(i-1)$ criterion, rather than generating them from scratch.

Tables 11 and 12 show the percentage of test suites revealing different behaviors over all test suites satisfying statement and all-uses data-flow adequacy criteria, respectively (e.g., Table 12 shows that, for *Schedule* v2, only 16% of test suites satisfying all-uses distance-0 criterion reveal different behaviors). The data in the tables show that, in all but one case, 22% or less of the test suites satisfying the statement adequacy criterion will reveal different behaviors. In the case of the all-uses distance-i adequacy criterion, $0 \leq i \leq 2$, the data also show that the all-uses distance-2 adequacy criterion is adequate for *Tcas* v2. However, none of the all-uses distance-i adequacy criteria, $0 \leq i \leq 2$, is adequate for *Schedule* because the average percentage of test suites revealing different behaviors is only 16.8%. The results confirm our intuition that all-uses adequate test suites are more effective

Table 13: Average number of test cases in test suites that satisfy all-uses distance-i adequacy criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	1.00	1.24	2.22
v2	1.00	1.00	3.00
v3	1.14	1.80	1.80
v4	2.74	4.22	4.22

Tcas

ver	distance		
	0	1	2
v1	1.00	1.54	1.78
v2	1.00	1.68	2.56
v3	1.00	1.68	2.10
v4	1.00	2.08	2.38
v5	1.34	1.44	1.68

Schedule

in revealing different behaviors than statement adequate test suites, and that the longer the dependence distances considered, the more effective the criteria become. However, the results also show that, in many cases, these test-adequacy criteria do not effectively exercise changes.

To measure the cost of generating a test suite satisfying the existing test-adequacy criteria, we measure the average size of the test suites we created. The size of all test suites satisfying the statement-adequacy criterion for any changes is 1. (Therefore, we do not show this result in the tables.) Table 13 shows the average number of test cases in test suites that satisfy an all-uses distance-i criterion for $0 \leq i \leq 2$. For example, the average size of the test suites satisfying all-uses distance-1 adequacy for the changes in *Tcas* v1 is 1.24. The data show that the average size of the test suite satisfying any of the all-uses adequacy criteria is 3.00 or below in most cases, with the exception of the changes in *Tcas* v3 at distances 1 and 2, which is 4.22. Overall, the results show that the cost of generating test suites satisfying data-flow adequacy criteria considering only du-pairs that are only a few dependences away from the changes is not much higher than the cost of generating test suites satisfying the statement adequacy criterion.

We can also use these data to compute a measure of cost-effectiveness of the criteria, by computing the ratio of the percentage of test suites revealing different behaviors to the average size of the test suites. For example, for the all-uses distance-0 and distance-1 adequacy criteria for *Tcas* v3, the ratios are 15.79 (18/1.14) and 37.78 (68/1.8), respectively. The results show that, for the subjects and versions considered, the cost-effectiveness for

the all-uses-based criteria tends to increase with the distance.

5.4.3 Study 2: Change-based Criteria

The goal of this study is to evaluate the effectiveness and the cost of our change-based criteria. We use the same effectiveness and cost measures as in Study 1 and also follow the same process.

Table 14 shows the percentage of test suites revealing different behaviors for each of our distance-i criteria and for each version of our subjects. As the data show, our change-based criteria are more effective than the corresponding all-uses criteria—and much more effective than the statement adequacy criterion—for distances greater than zero. (They are more effective in most cases also for distance 0.) In particular, for *Tcas*, between 90% and 100% of the test suites that satisfy the distance-2 requirements reveal different behaviors between old and modified versions of the program. The results for *Schedule* are not as good from an absolute standpoint, but are still considerably better than the results for the corresponding all-uses criteria.

Note that, for changes in Schedule v1 and v5, none of the test suites that satisfy our criteria reveal different behaviors. After inspecting the subjects, we discovered that the changes in these versions affect the program state but not the control- and data-flow of the program. Criteria based on control- or data-flow are therefore unlikely to reveal these changes, as the results for the statement- and all-uses-based criteria show (see Tables 11 and 12). The reason why our technique does not reveal the difference either is that its current implementation does not generate requirements to exercise differences in the program state, as discussed in Section 5.3.

Table 15 shows the average number of test cases in test suites that satisfy each of our distance-i criteria for each subject version. The results show that our set of criteria needs at most (for *Schedule* v1 and distance 1) about twice as many test cases as the all-uses adequacy criterion at the same distance. Note that, because the test suites for longer distances are built on those for lower distances, and they are not reduced, the number of test cases per test suite for longer distances (for both our change-based criteria and the all-uses criteria) may not accurately reflect the actual test-suite generation costs. This explains why, in some

Table 14: Percentage of test suites revealing different behaviors over 50 test suites that satisfy our distance-i criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	30	30	90
v2	4	100	100
v3	100	100	100
v4	100	100	100

Tcas

ver	distance		
	0	1	2
v1	0	0	0
v2	10	48	94
v3	16	64	82
v4	36	56	60
v5	0	0	0

Schedule

Table 15: Average number of test cases in test suites that satisfy our distance-i criteria for $0 \leq i \leq 2$ and for modified versions of *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	1.00	1.00	1.80
v2	1.00	1.96	1.96
v3	1.70	1.70	1.70
v4	3.76	3.94	4.88

Tcas

ver	distance		
	0	1	2
v1	1.88	3.44	3.44
v2	1.00	1.84	4.50
v3	1.00	2.08	3.42
v4	1.50	2.38	3.20
v5	1.58	2.44	2.64

Schedule

cases, all-uses adequacy criteria require more test cases than our change-based criteria for the same distance and the same subject (e.g., for *Tcas* v1 and distance 2).

In terms of cost-effectiveness, our criteria are more cost-effective than both statement-based and all-uses-based criteria in most cases. (In the following, we do not consider v1 and v5 of *Schedule*, for which none of the criteria generate test cases that can reveal changes in behavior.) For distances greater than zero, our criteria are more cost-effective than the alternative criteria in all but one case (*Tcas* v4). For distance 0, our criteria are more cost-effective in eight out of 14 cases.

CHAPTER 6

CONCLUSIONS AND FUTURE DIRECTIONS

This research addresses the problem of assessing the quality of test suites with respect to changes in a modified version of a program by defining change-based testing criteria, which can be used in the quality assessment, and identifying testing requirements that form the criteria. These testing requirements can also be used to guide the generation of new, effective test inputs targeting the changes.

This research has three components.

- A program differencing technique that computes change information and mappings of program entities in the old and new versions and handles the changes involving object-oriented features.
- Two dynamic-impact-analysis techniques that use change information to identify program entities that are likely affected by changes during at least one of the collected program executions. These two techniques have a trade-off between precision and efficiency.
- A technique that uses change and impact information to identify testing requirements, which form change-based testing criteria. These requirements guarantee that the test suites satisfying them, when executed on the modified version, will result in different control or data flow or different program states at selected program points.

6.1 Merit of this research

First, this research improves the effectiveness of regression testing by identifying testing requirements that enable testers to evaluate the extent to which their test suites exercise changes more effectively than criteria based on control- and data-flow alone. This research also enables testers to develop new test cases targeting modified behavior. Effective regression testing, in turn, reduces the number of software failures in the field. Because field

failures may incur significant direct cost (such as the cost of debugging under tight time constraint and distributing patches to users) and indirect cost (such as user dissatisfaction), a technique that reduces the number of these failures will contribute to considerable cost reduction.

Second, this research produced techniques that, in addition to their applications in testing evolving software, can also be used in other contexts. A number of examples of such applications are presented here. In a collaborative environment where a number of developers modify copies of the same modules at the same time, the precise differencing technique can alert these developers of potential conflicts in their modifications and enable program-merging techniques to incorporate all non-conflict changes into a new version automatically. In the situation where the coverage or profile information for the modified program is required but cannot be reproduced (e.g., information from deployed software), the differencing results, along with the coverage or profile information for the original version, can be used to estimate this information. This approach also eliminates the cost of rerunning the test suite on the modified version of the program to obtain the coverage or profile information. In many software projects where one desired change can be performed in more than one way, the dynamic-impact-analysis technique can be applied to estimate the cost of those proposed modifications and select among them and, thus, contributes to better resource management. The impact information computed by dynamic impact analysis can also be used to evaluate the extent of coupling among multiple software modules and to re-engineer the software design.

Third, this research implemented a number of tools to evaluate the effectiveness and efficiency of the developed techniques. These tools can be integrated into other systems that require their functionalities. For example, these tools can be integrated with regression-test-selection tools, test-suite prioritization and reduction tools, and test-case-generation tools to form a regression testing environment that provides all of these functionalities. Moreover, the JDiff tool has been released and used by other researchers in their own work. The CoverageImpact, EAT, and MaTRIX will be released to the research community.

6.2 Future Directions

Even though we expect that the impact of changes that propagate to statements at a small distance will continue to propagate to output, and our empirical results confirm this intuition, this may not be the case for some changes. To support the identification of testing requirements at great distance, the technique needs to improve the efficiency of partial symbolic execution, which can be achieved by two means. First, the existing lazy-initialization algorithm does not provide an optimal solution for initializing an uninitialized object. For example, when a field of an uninitialized object is referenced, the algorithm needs to initialize that object. To do so, the algorithm splits the symbolic-execution path into several paths, each of which corresponds to a possible alternative (i.e., any of all the previously lazily-initialized objects of compatible types or an unseen object for each compatible type). However, several symbolic-execution paths may analyze the same set of statements during partial symbolic execution (e.g., the path corresponding to an unseen object of type A and the path corresponding to an unseen object of type B when B is a subtype of A and the reference field is not hidden). Rather than splitting into several different paths, the lazy-initialization algorithm could add a constraint on the symbolic reference to indicate that, for this path, this symbolic reference can be either of type A or B. Second, partial symbolic execution needs not execute every statement between a change and an affected statement because the affected statement may not depend (even indirectly) on some of the statements. A slicing approach that filters out all statements that are not involved in the computation of the affected statement will reduce the number of statements needed to be analyzed and, thus, reduce the cost of partial symbolic execution.

The empirical studies on testing-requirements identification have shown the effectiveness of this technique on two subject programs and a limited number of changes. Although the technique requires symbolic execution on a small part of the program (the part close to the changes) and, thus, is expected to scale to large programs, more empirical studies on larger subjects with real-world sets of multiple changes should be conducted to support the generalizability of the current results.

As discussed in Chapter 5, the identified testing requirements can be used as guidelines

to generate new test inputs targeting the changes. However, manually generating test inputs that exercise the changes effectively is a tedious, time-consuming task. Because the testing- requirements representation (i.e., boolean predicates) is formal, a technique to generate automatically test inputs that satisfy these requirements could be developed. Such technique would need to solve the constraints that include the testing requirements and a path condition from the beginning of the program to each of the changes.

REFERENCES

[1] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] Anand, S., Orso, A., and Harrold, M. J., "JPF-SE: A symbolic execution extension to java pathfinder," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction And Analysis of Systems*, (Braga, Portugal), pp. 134–138, March 2007.

[3] Anand, S., Orso, A., and Harrold, M. J., "Type-dependence analysis and program transformation for symbolic execution," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction And Analysis of Systems*, (Braga, Portugal), pp. 117–133, March 2007.

[4] Apiwattanapong, T., Orso, A., and Harrold, M. J., "A differencing algorithm for object-oriented programs," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, (Linz, Austria), pp. 2–13, September 2004.

[5] Atkinson, D. C. and Griswold, W. G., "Implementation techniques for efficient data-flow analysis of large programs," in *Proceedings of the IEEE International Conference on Software Maintenance*, pp. 52–61, November 2001.

[6] Binkley, D., "Using semantic differencing to reduce the cost of regression testing," in *Proceedings of the IEEE Conference on Software Maintenance*, (Orlando, FL, USA), pp. 41–50, November 1992.

[7] Binkley, D., "Semantics guided regression test cost reduction," *IEEE Transactions on Software Engineering*, vol. 23, pp. 498–516, August 1997.

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

[9] Bowring, J. F., Rehg, J. M., and Harrold, M. J., "Active learning for automatic classification of software behavior," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, (Boston, MA, USA), pp. 195–205, July 2004.

[10] Breech, B., Danalis, A., Shindo, S., and Pollock, L., "Online impact analysis via dynamic compilation technology," in *Proceedings of the 20th IEEE International Conference of Software Maintenance*, (Chicago, IL, USA), pp. 453–457, September 2004.

[11] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L., "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computing Systems*, vol. 19, pp. 332–383, August 2001.

[12] Chawla, A. and Orso, A., "A generic instrumentation framework for collecting dynamic information," in *Online Proceedings of the ISSTA Workshop on Empirical Research in Software Testing*, july 2004.

[13] Chen, Y. F., Rosenblum, D. S., and Vo, K. P., "TestTube: A system for selective regression testing," in *Proceedings of the 16th International Conference on Software Engineering*, pp. 211–222, May 1994.

[14] Clarke, L. A. and Richardson, D. J., "Applications of symbolic evaluation," *Journal of Systems and Software*, vol. 5, pp. 15–35, February 1985.

[15] Clause, J. and Orso, A., "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the International Symposium on Software Testing and Analysis*, (London, UK), July 2007.

[16] Cytron, R., Ferrante, J., Rosen, B. K., N.Wegman, M., and Zadeck, F. K., "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[17] Davis, M., Logemann, G., and Loveland, D., "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, pp. 394–397, July 1962.

[18] Davis, M. and Putnam, H., "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, pp. 201–215, July 1960.

[19] Deng, X., Robby, and Hatcliff, J., "Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs," in *Proceedings of the IEEE International Conference on Software Engineering and Formal Methods*, (London, UK), September 2007.

[20] Deng, X., Lee, J., and Robby, "Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems," in *Proceedings of the IEEE International on Automated Software Engineering*, (Tokyo, Japan), pp. 157–166, September 2006.

[21] Duesterwald, E., Gupta, R., and Soffa, M. L., "Rigorous data flow testing through output influences," in *Proceedings of the Second Irvine Software Symposium*, pp. 131–145, March 1992.

[22] Dutertre, B. and de Moura, L., "A fast linear-arithmetic solver for dpll(t)," in *Proceedings of the International Conference on Computer Aided Verification*, vol. 4144 of *LNCS*, pp. 81–94, Springer-Verlag, 2006.

[23] Ernst, M. D., *Dynamically discovering likely program invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, WA, USA, August 2000.

[24] Ferrante, J., Ottenstein, K. J., and Warren, J. D., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[25] Foundation, E., "Eclipse – an open development platform." http://www.eclipse.org, December 2001.

[26] Frankl, P. G. and Weyuker, E. J., "An applicable family of data flow criteria," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1483–1498, October 1988.

[27] Frederick P. Brooks, J., "No silver bullet: Essence and accidents of software engineering," *Computer*, vol. 20, pp. 10–19, April 1987.

[28] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., and Tinelli, C., "Dpll(t): Fast decision procedures," in *Proceedings of the International Conference on Computer Aided Verification*, (Boston, MA), pp. 175–188, July 2004.

[29] Gupta, R., Harrold, M. J., and Soffa, M. L., "Program slicing-based regression testing techniques," *Journal of Software Testing, Verification, and Reliability*, vol. 6, pp. 83–111, June 1996.

[30] Harder, M., Mallen, J., and Ernst, M. D., "Improving test suites via operational abstraction," in *Proceedings of the 25th IEEE and ACM SIGSOFT International Conference on Software Engineering*, pp. 60–71, May 2003.

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

[32] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A., "Regression test selection for java software," in *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 312–326, November 2001.

[33] Holzmann, G. J. and Puri, A., "A minimized automaton representation of reachable states," *International Journal on Software Tools for Technology Transfer*, vol. 2, pp. 270–278, November 1999.

[34] Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (White Plains, NY, USA), pp. 234–246, June 1990.

[35] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, January 1990.

[36] Howden, W. E., "Methodology for the generation of program test data," *IEEE Transactions on Computers*, vol. 24, pp. 554–560, May 1975.

[37] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the 16th international conference on Software engineering*, pp. 191–200, May 1994.

[38] Jackson, D. and Ladd, D. A., "Semantic diff: A tool for summarizing the effects of modifications," in *Proceedings of the Internation Conference on Software Maintenance*, (Victoria, BC, Canada), pp. 243–252, September 1994.

[39] Jayaraman, G., Ranganath, V. P., and Hatcliff, J. in *Proceedings of the Eighth International Conference on Fundamental Approaches to Software Engineering*, (Edinburgh, Scotland), pp. 269–272, April 2005.

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

[41] Larus, J., "Whole program paths," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, GA, USA), pp. 259–269, May 1999.

[42] Laski, J. and Szermer, W., "Identification of program nodifications and its applications in software maintenance," in *Proceedings of the IEEE Conference on Software Maintenance*, (Orlando, FL, USA), pp. 282–290, November 1992.

[43] Laski, J. W. and Korel, B., "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. 9, pp. 347–354, May 1983.

[44] Law, J. and Rothermel, G., "Incremental dynamic impact analysis for evolving software systems," in *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, (Denver, CO, USA), pp. 430–441, November 2003.

[45] Law, J. and Rothermel, G., "Whole program path-based dynamic impact analysis," in *Proceedings of the International Conference on Software Engineering*, (Portland, OR, USA), pp. 308–318, May 2003.

[46] Loyall, J. P., Mathisen, S. A., and Satterthwaite, C. P., "Impact analysis and change management for avionics software," in *Proceedings of the IEEE National Aeronautics and Electronics Conference, Part 2*, pp. 740–747, July 1997.

[47] Maletic, J. I. and Collard, M. L., "Supporting source code difference analysis," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, (Chicago, IL, USA), pp. 210–219, September 2004.

[48] Morell, L., "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, pp. 844–857, August 1990.

[49] Myers, E. W., "An O(ND) difference algorithm an its variations," *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[50] Nevill-Manning, C. and Witten, I., "Linear-time, incremental hierarchy inference for compression," in *Proceedings of the IEEE Data Compression Conference*, (Snowbird, Utah, USA), pp. 3–11, March 1997.

[51] Nieuwenhuis, R. and Oliveras, A., "Decision procedures for sat, sat modulo theories and beyond. the barcelogictools," in *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, (Montego Bay, Jamaica), December 2005.

[52] Nieuwenhuis, R. and Oliveras, A., "Dpll(t) with exhaustive theory propagation and its application to difference logic," in *Proceedings of the International Conference on Computer Aided Verification*, (Edinburgh, Scotland, UK), pp. 321–334, July 2005.

[53] Ntafos, S. C., "On required element testing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 795–803, November 1984.

[54] Orso, A., Harrold, M. J., Rosenblum, D., Rothermel, G., Soffa, M. L., and Do, H., "Using component metadata to support the regression testing of component-based software," in *Proceedings of the International Conference on Software Maintenance*, pp. 716–725, November 2001.

[55] Orso, A., Apiwattanapong, T., and Harrold, M. J., "Leveraging field data for impact analysis and regression testing," in *Proceedings of the 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, (Helsinki, Finland), pp. 128–137, September 2003.

[56] Orso, A., Apiwattanapong, T., Law, J. B., Rothermel, G., and Harrold, M. J., "An empirical comparison of dynamic impact analysis algorithms," in *Proceedings of the 26th IEEE and ACM SIGSOFT International Conference on Software Engineering*, (Edinburgh, Scotland,UK), pp. 491–500, May 2004.

[57] Orso, A., Jones, J., and Harrold, M. J., "Visualization of program-execution data for deployed software," in *Proceedings of the ACM Symposium on Software Visualization*, pp. 67–76, June 2003.

[58] Pfleeger, S. L., *Software Engineering: Theory and Practice*. Englewood Cliffs, NJ, USA: Prentice Hall, 1998.

[59] Raghavan, S., Rohana, R., Loen, D., Podgurski, A., and Augustine, V., "Dex: A semantic-graph differencing tool for studying changes in large code bases," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, (Chicago, IL, USA), pp. 188–197, September 2004.

[60] Rangarajan, K., "Automatic analysis of java program evolution and its relevance to regression testing." http://www.mmsindia.com/JEvolve.html, December 2001.

[61] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O., "Chianti: A tool for change impact analysis of java programs," in *Proceedings of the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (Vancouver, BC, Canada), pp. 432–448, 2004.

[62] Richardson, D. J. and Thompson, M. C., "The RELAY model of error detection and it application," in *Proceedings of the ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pp. 223–230, July 1988.

[63] Rothermel, G. and Harrold, M. J., "Selecting tests and identifying test coverage requirements for modified software," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pp. 169–184, August 1994.

[64] Rothermel, G. and Harrold, M. J., "A safe, efficient regressing test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, pp. 173–210, April 1997.

[65] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C., "An empirical study of the effects of minimization on the fault-detection capabilities of test suites," in *Proceedings of the International Conference on Software Maintenance*, pp. 34–43, November 1998.

[66] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J., "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, pp. 929–948, October 2001.

[67] Ryder, B. G. and Tip, F., "Change impact analysis for object-oriented programs," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 46–53, June 2001.

[68] Seesing, A. and Orso, A., "InsECTJ: A generic instrumentation framework for collecting dynamic information within eclipse," in *Proceedings of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, (San Diego, USA), pp. 49–53, October 2005.

[69] Sinha, S. and Harrold, M. J., "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, pp. 849–871, September 2000.

[70] Srivastava, A. and Thiagarajan, J., "Effectively prioritizing tests in development environment," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 97–106, July 2002.

[71] Turver, R. J. and Munro, M., "Early impact analysis technique for software maintenance," *Journal of Software Maintenance*, vol. 6, pp. 35–52, January 1994.

[72] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F., "Model checking programs," *Automated Software Engineering Journal*, vol. 10, pp. 203–232, April 2003.

[73] Voas, J. M., "PIE:a dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–727, August 1992.

[74] Vokolos, F. and Frankl, P., "Pythia: A regression test selection tool based on text differencing," in *Proceedings of IFIP TC5 WG5.4 the 3rd Internatinal Conference on Reliability, Quality and Safety of Software-Intensive Systems*, pp. 3–21, May 1997.

[75] Wang, Z., Pierce, K., and McFarling, S., "BMAT – a binary matching tool for stale profile propagation," *The Journal of Instruction-Level Parallelism*, vol. 2, May 2000.

[76] Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[77] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P., "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*, pp. 41–50, April 1995.

[78] Xing, Z. and Stroulia, E., "UMLDiff: An algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering*, (Long Beach, CA, USA), pp. 54–65, November 2005.