Test-suite Augmentation for Evolving Software*

Raul Santelices,† Pavan Kumar Chittimalli,‡ Taweesup Apiwattanapong,+
Alessandro Orso,† and Mary Jean Harrold†

†College of Computing, Georgia Institute of Technology, USA
‡Tata Research Development & Design Center, TCS Ltd., India
+Software Engineering Technology Laboratory, NECTEC, Thailand
E-mail: {raul|orso|harrold}@cc.gatech.edu,
pavan.chittimalli@tcs.com, taweesup.apiwattanapong@nectec.or.th

Abstract

One activity performed by developers during regression testing is test-suite augmentation, which consists of assessing the adequacy of a test suite after a program is modified and identifying new or modified behaviors that are not adequately exercised by the existing test suite and, thus, require additional test cases. In previous work, we proposed MATRIX*, a technique for test-suite augmentation based on dependence analysis and partial symbolic execution. In this paper, we present the next step of our work, where we (1) improve the effectiveness of our technique by identifying all relevant change-propagation paths, (2) extend the technique to handle multiple and more complex changes, (3) introduce the first tool that fully implements the technique, and (4) present an empirical evaluation performed on real software. Our results show that our technique is practical and more effective than existing test-suite augmentation approaches in identifying test cases with high fault-detection capabilities.*

1. Introduction

Regression testing is the activity of retesting a program after it has been modified to gain confidence that existing, changed, and new parts of the program behave correctly. This activity is typically performed by rerunning, completely or partially, a set of existing test cases (i.e., its *regression test suite*). Given a program P and a modified version of the program P', the regression test suite can reveal differences in behavior between P and P' and, thus, help developers discover errors caused by the changes or by unwanted side effects of the changes introduced in P'.

There is much research on making regression testing more efficient by (1) identifying test cases in a regression test suite that need not be rerun on the modified version of the software (e.g., [4], [16]), (2) eliminating redundant test cases in a test suite according to given criteria (e.g., [8], [22]), and (3) ordering test cases in a test suite to help find defects earlier (e.g., [17], [18]). Little research, however, has focused on the effectiveness of the regression test suite with respect to the changes. To evaluate such effectiveness, it is necessary, when performing regression testing, to (1) check whether existing test suites are adequate for the changes introduced in a program and, if not, (2) provide guidance for creating new test cases that specifically target the (intentionally or unintentionally) changed behavior of the program. We call this problem *test-suite augmentation*.

Existing test-suite augmentation approaches address this problem by defining criteria that require exercising single control- or data-flow dependences related to program changes (e.g., [2], [3], [7], [15]). In our experimentation, we found that considering the effects of changes on single control- and data-flow relations alone does not adequately exercise either the effects of software changes or the modified behavior induced by such changes. To illustrate this inadequacy, consider a change c and a statement s that is data and/or control dependent on c. First, the effects of c may propagate to s only along some of the (possibly many) paths between c and s. Thus, just covering c and s may not exercise the effects of c on s. Second, even if a path that may propagate c's effects to s is covered, such effects may manifest themselves only under specific conditions. Therefore, criteria that simply require the coverage of program entities (e.g., statements and data-flow relations) are often inadequate for testing changed behavior.

To address the limitations of existing techniques, we present an approach that combines dependence analysis and symbolic execution to identify test requirements that are likely to exercise the effects of one or more changes to a program. Our approach is based on two main intuitions. The *first intuition* is that test criteria for changed software must require test cases to reach potentially affected areas

* *Patent pending, Georgia Institute of Technology.*

of the code along different, relevant paths. Therefore, our approach requires that specific chains of data and control dependences be exercised. The *second intuition* is that test requirements for changed software must account for the state of the software and the effects of changes on that state. Thus, our approach leverages partial symbolic execution [1], a type of symbolic execution that is performed starting from a change, focuses on the parts of the program state affected by that change, and encodes the effects of the change at specific program points.

Because of the cost of the analyses involved—in terms of both their computational complexity (for symbolic execution) and the number of test requirements they may generate—our approach limits its analysis to a given distance from the changes considered. By doing this, our approach can leverage the power of its underlying analyses while still being practical. A test suite that satisfies the test requirements generated by our approach for a set of changes is guaranteed to exercise, up to a given distance, the effects of those changes along relevant paths.

Intuitively, given a program P, its modified version P', the set of changes between P and P', a test suite T for P, and a distance d, our approach performs four main steps: (1) computes all chains of dependences from the changes to statements at distance up to d from the changes; (2) computes the conditions under which the statements in the identified chains behave differently in P and P'; (3) expresses both chains and conditions as test requirements; (4) assesses the extent to which T satisfies (i.e., covers) the identified requirements. The computed coverage indicates the adequacy of T with respect to the changes between P and P'. Moreover, the requirements that are not satisfied can be used to guide the generation of additional test cases.

In previous work, we presented our general approach, the definition of partial symbolic execution, a first instantiation of the approach for single changes based on partial symbolic execution, symbolic state differencing, and dependence distance, and a case study that shows the feasibility of the approach [1]. In this paper, we extend our previous definition of the technique in two directions. First, we define requirements for change-effects propagation along individual dependence chains, so identifying all relevant paths to be exercised. Second, we define the way in which to handle multiple program changes.

In this paper, we also discuss a tool, called MATRIXRELOADED, that fully implements our approach. Finally, the paper discusses the results of an empirical study that we performed using MATRIXRELOADED on two subjects and a set of changes for these subjects. The results of the study provide evidence that our approach is practical and more effective than existing test-suite augmentation approaches in identifying test cases with high likelihood of detecting change-related faults.

This paper provides the following main contributions:

- A test-suite augmentation technique that extends our previous approach by targeting all relevant dependence chains and considering multiple changes.
- A tool, MATRIXRELOADED, that fully implements our approach for programs written in Java.
- An empirical evaluation that shows the practicality and effectiveness of our approach when applied to real software and a set of changes for this software.

2. Background

This section describes two main concepts for our paper: control and data dependences, and symbolic execution.

2.1. Control and Data Dependences

The effects of a change propagate to other statements through control and data dependences. Informally, statement s_1 is *control-dependent* [5] on statement s_2 if s_2 has at least two outgoing control edges, and for at least one but not all of these edges, every path covering that edge also covers s_1. We represent a control dependence as (a, b), where b is control dependent on a. To illustrate, consider the program, E, shown in Figure 1(a), whose control-flow graph (CFG) is shown in Figure 1(b). This program takes two inputs, x and y, and outputs one of two values: 0 or 1. Statement 2 is control-dependent on statement 1, represented as $(1, 2)$.

Statement s_1 is *data dependent* on statement s_2 if (1) s_2 defines a variable, v, (2) there is a *definition-clear* path from s_2 to s_1 (i.e., a path that contains no redefinition of v), and (3) s_1 uses v. We represent a data dependence as a triple (a, b, v), where statement b is dependent on statement a for variable v. For example, in Figure 1, statement 6 is data dependent on statement 2 for variable y, represented as $(2, 6, y)$.

A *program dependence graph* (*PDG*) represents both control and data dependences: nodes represent statements and edges represent control and data dependences. Figure 1(c) shows the PDG for E. Statements in a procedure p that are not control dependent on any other statement in p (e.g., statements 2 and 6) are control dependent on p's entry node.

A sequence of two or more dependences forms a *dependence chain* if the target statement of one dependence is the source of the next dependence in the sequence. For example, control dependence $(1, 2)$ and data dependence $(2, 6, y)$ form a chain of length two because statement 2 is control dependent on statement 1, and statement 2 is the source of data dependence $(2, 6, y)$. In this paper, we treat a single statement as a chain of length zero. Table 1 shows the dependence chains of lengths 1, 2, and 3 that

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

(a)

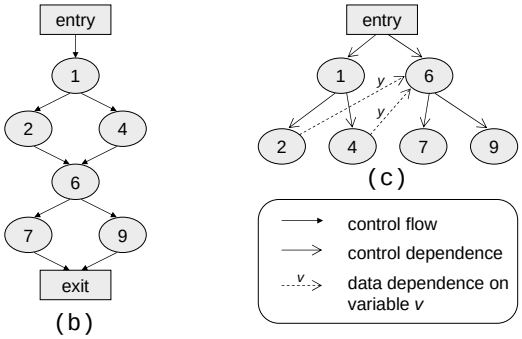

Figure 1. (a) Example program E, (b) CFG for the program, and (c) PDG for the program.

Table 1. Dependences in Figure 1, change 1

length	chains	inputs	difference
1	$ch_{1,1} = (1, 2)$	80	16
	$ch_{1,2} = (1, 4)$	20	4
2	$ch_{2,1} = (1, 2), (2, 6, y)$	80	16
	$ch_{2,1} = (1, 4), (4, 6, y)$	20	4
3	$ch_{3,1} = (1, 2), (2, 6, y), (6, 7)$	72	16
	$ch_{3,2} = (1, 2), (2, 6, y), (6, 9)$	8	0
	$ch_{3,3} = (1, 4), (4, 6, y), (6, 7)$	14	0
	$ch_{3,4} = (1, 4), (4, 6, y), (6, 9)$	6	4

start at `change 1` (statement 1) in Figure 1. (The last two columns of Table 1 are used in Section 3.) A chain $ch_{n,i}$ corresponds to the i^{th} chain of length n.

2.2. Symbolic Execution

Symbolic execution [10] analyzes a program by executing it with symbolic inputs along some program path. Symbolically executing all paths in a program to a given point (if feasible) effectively describes the semantics of the program up to that point. Symbolic execution represents the values of program variables at any given point in a program path as algebraic expressions by interpreting the operations performed along that path on the symbolic inputs. The *symbolic state* of a program at a given point consists of the set of symbolic values for the variables in scope at that point. The set of constraints that the inputs must satisfy to follow a path is called a *path condition* and is a conjunction of constraints p_i or $\neg p_i$ (depending on the branch taken), one for each predicate traversed along the path. Each p_i is obtained by substituting the variables used in the corresponding predicate with their symbolic values. Symbolic execution on all paths to a program point represents the set of possible states at that point as a disjunction of clauses, one for each path that reaches the point. These clauses are of the form $PC_i \Rightarrow S_i$, where PC_i is the path condition for path i, and S_i is the symbolic state after executing path i.

For example, consider program E in Figure 1, without changes. Symbolic execution first assigns symbolic values x_0 and y_0 to inputs x and y, respectively. When statement 1 is executed, the technique computes the path conditions for the *true* and *false* branches, which are $x_0 \leq 2$ and $x_0 > 2$, respectively. These conditions are shown in column *path condition* of Table 2, for statements 2 and 4. The values of variables at the entry of each statement are shown in column *symbolic state*. For example, after evaluating statement 2, the technique updates the value of y to y_0+1. The execution of the remaining statements is performed analogously. Each row in Table 2 shows the path conditions and the symbolic values at the entry of the corresponding statement, after traversing all paths to that statement. For example, there are two path conditions and symbolic states for statement 7. Each path condition in the second column implies the state on its right in the third column. Symbolic execution associates with each statement the disjunction of all rows for that statement, where each row represents a path condition and its corresponding symbolic state.

Table 2. Symbolic execution for E in Figure 1

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

3. The Test-suite Augmentation Technique

In this section we present our test-suite augmentation technique. In Section 3.1, we present an overview of our technique. In Section 3.2, we present our technique for single changes. Finally, in Section 3.3, we describe how our technique handles multiple changes.

3.1. Overview of the Technique

Evolving software requires a test suite that evolves with it. Changes in a program add, remove, or modify functionality. Thus, it is necessary to assess the adequacy of an existing test suite after changes are made and provide guidance in adding new test cases that adequately exercise all effects of changes. Just as testing requirements are necessary for software as a whole, specific testing

requirements are also necessary to test changes. We call these specific requirements *change-testing requirements*.

Our test-suite augmentation technique is inspired by the PIE model [20] of fault propagation, in which the testing criterion for a fault should ensure that the fault is executed (E), that it infects the state (I), and that the infected state propagates to the output (P). The goal of our technique is to provide requirements to achieve such a propagation for all possible effects of a change. In general, however, producing such requirements is infeasible, and even if feasible in particular cases, it is not practical due to the explosion in the number of constraints as the distance between changes and output grows.

To address this problem, our technique sets a limit on the distance from the change to which the requirements guarantee propagation of the infection. The intuition is that a test case that propagates the effects of a change up to this distance has an improved chance of propagating the effects to the output. Test cases that do not propagate the infection to this distance definitely do not propagate the infection to the output and are disregarded.

In earlier work, we proposed a first instantiation of our approach for generating requirements to test single changes [1]. That approach requires test cases to propagate a state infection to a certain dependence distance from the change. In this paper, we call these requirements *state requirements*. State requirements are only partially effective, in that they do not consider *how* the infection propagates to the given distance; often, there are multiple ways in which the effects of a change can propagate to a point. To address this limitation, in this paper, we introduce the concept of multiple-chain propagation requirements, which complement state-based requirements and improve the overall effectiveness of the approach. Figure 2 provides an intuitive depiction of the way our approach works. Our goal is to create requirements for the *difference-revealing* subset of test cases, which are the test cases that produce an observable different behavior in the original and modified programs. This subset is first approximated by test cases that achieve *simple change coverage* (i.e., they just cover the change), and then increasingly better approximated by subsets that satisfy our chain and state requirements—as distance increases, test suites for our requirements get closer to the difference-revealing subset.

3.2. Single-change Requirements

In this section, we formulate our requirements generation technique based on state-infection propagation along multiple dependence chains. Our technique consists of two phases. The first phase, presented in Section 3.2.1, identifies dependence-chain coverage requirements. The second phase, presented in Section 3.2.2, adds state-infection propagation requirements for each chain.

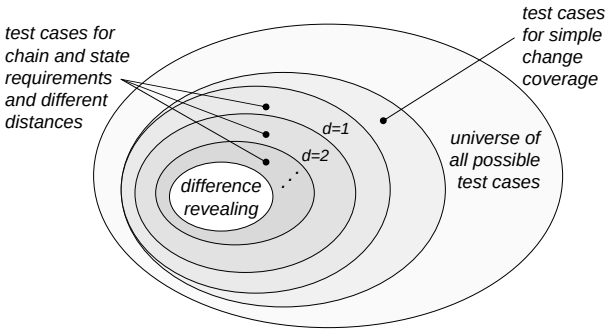

Figure 2. Intuitive view of change-testing criteria.

3.2.1. Phase 1: Chain requirements. A simple strategy for testing a change is to cover all changed statements. However, this strategy often fails to guarantee that all possible effects of the change on different parts of the program are tested. Consider program E in Figure 1, with `change 1` applied. We call this modified version E_1. The example program has $10\times10{=}100$ valid test inputs. By examining the code, we can see that the change produces a different behavior in E_1 (i.e., it produces a different output with respect to E) if and only if $y \in [2, 3]$, which corresponds to 20 inputs, or 20% of the input space. Hence, randomly selecting an input that covers the change has a small chance of revealing a difference.

In many cases, an adequate test suite for the simple change-coverage strategy exercises only a small fraction of all dependence chains along which a change may propagate. Because the effects of a change can propagate forward along any of the chains starting from the change, producing a potentially different infected state per chain, an adequate testing strategy should require the coverage of all such chains. Consider the requirement of propagating the change along chains of length one (i.e., along direct dependences) from `change 1` in Figure 1. Two control dependences must be tested: $(1, 2)$ and $(1, 4)$. As Table 1 shows in columns *input* and *difference*, for $(1, 2)$ there are 80 covering inputs, 16 of which reveal a difference. For $(1, 4)$, there are 20 possible inputs, four of which reveal a difference. Each chain has a probability of 20% of revealing a difference. A test suite T that covers all direct dependences from the change in this example has thus a 36%[1] chance of revealing a difference. Therefore, covering all chains of length one increases the probability of difference detection from 20% to 36%.

Consider now dependence chains of length greater than one. The number of chains can grow exponentially with their length, but in practice it is possible to require coverage of chains of a limited length. In the example for `change 1`, there are four chains of length 3 reaching

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

Algorithm COMPUTEREQS()

Input: P, P': original and modified programs
outStmts: set of output statements in P'
changes: set of changes in P' respect to P
maxDist: maximum length of chains
Output: *chainReqs*, *stateReqs*: chain and state requirements

```
      // Phase 1: chain requirements
(1)   alignPrograms (P, P', changes)
(2)   chainSet = allStmts (changes)
(3)   for d = 1 to maxDist
(4)     foreach chain ∈ chainSet
(5)       chainSet −= chain
(6)       if end(chain) ∈ outStmts
(7)         chainReqs ∪= chain
(8)       endif
(9)       foreach dep ∈ nextDep (P', end(chain))
(10)        newChain = append (chain, dep)
(11)        chainSet ∪= newChain
(12)      endfor
(13)    endfor
(14)  endfor
(15)  chainReqs ∪= chainSet
      // Phase 2: state requirements
(16)  foreach chain ∈ chainSet
(17)    if existsClearPath (P', start(chain), changes))
(18)      (S',S) = PSE (chain, P', P)
(19)      stateReqs ∪= (chain, live(S') ≠ live(S))
(20)      stateReqs ∪= (chain, pc(S') ∧ ¬ pc(S))
(21)    endif
(22)  endfor
(23)  return chainReqs, stateReqs
```

Figure 3. The algorithm to compute chain- and state-requirements.

output statements 7 and 9. Two of these chains can reveal a difference: $ch_{3,1}$ (Table 1), which is covered by 72 inputs, revealing a difference in 16 cases (22.2%), and chain $ch_{3,4}$, which is covered by six inputs, revealing a difference in four cases (66.7%). A test suite that covers all chains of length 3 would thus reveal a difference with probability 74.1%. A random test suite of size 4, in comparison, has a 59% chance of revealing a difference. This example shows that longer chains can improve the chances of revealing a different behavior after a change, compared to shorter chains or simple change coverage (i.e., chains of length 0), by requiring a well-distributed coverage of propagation paths from the change.

Figure 3 presents algorithm COMPUTEREQS, which computes the testing requirements for a set of changes in two phases. COMPUTEREQS inputs the original and modified programs (P and P'), the set *outStmts* of output statements in P', the set of changes from P to P', and the distance *maxDist*. Phase 1 (Lines 1–15) computes the chain-coverage requirements discussed in this section. (Phase 2 is explained in the next section). Line 1 calls *alignPrograms*, which adapts both programs to match all dependence points. For example, a definition of `x` deleted in P' is compensated by inserting `x = x` at the same point in P'. In Line 2, the algorithm initializes the set *chainSet* of chains to cover the chains of length 0 corresponding to all changed statements. The loop at lines 3–14 extends the chains *maxDist* times. Each chain is extracted from *chainSet* in lines 4–5, and Line 6 checks whether the chain has reached the output (using function *end* to get the end point of the chain). If the chain reaches the output, it is added at this time as a requirement (line 7). Lines 9–12 extend the chain by creating, for each dependence *dep* from the last element of the chain (provided by function *nextDep*), a new chain. This new chain, which consists of the original one with *dep* appended to it, is then added to *chainSet*. A chain with no additional dependences from its final element is therefore discarded (i.e., not added to *chainSet*). Finally, Line 15 adds all chains of length *maxDist* to the chain requirements.

3.2.2. Phase 2: State requirements. Chain requirements can increase the probability of finding output differences, but they cannot guarantee that the effects of the change propagate to the end of the chain. For example, in Figure 1 with `change 1` (i.e., E_1), only 16 out of 72 inputs (22%) that cover chain $ch_{3,1}$ (Table 1) reveal a difference at the end of that chain. The reason is that the program state after covering chain $ch_{3,1}$ and reaching statement 7 in E_1 might be the same as the state at statement 7 in E (the unmodified example), for the same input. Hence, to guarantee infection propagation to the end of a chain, our COMPUTEREQS algorithm produces additional state requirements.

Defining a requirement to guarantee propagation of the state infection to the output is infeasible in general. Instead, our technique computes state requirements to guarantee propagation of the infection to the end of each chain, by comparing the symbolic states of the program at the end of each chain in P' (the modified program) and the corresponding end point in P (the original program). These states are expressed in terms of the state variables at the entry of the change. A test case that reaches the endpoint n of a chain in P' would manifest a different behavior in P and P' if it satisfies one of the following two conditions:

- **Condition 1**: reach n in P through any path after the change and have different symbolic values for the live[2] variables at n (differences in dead[3] variables have no effect).
- **Condition 2**: do not reach n in P after the change, that is, the path condition to reach n after the change must not hold in P.

These two state conditions for a chain reduce the space of test cases for the chain to those that propagate an infection to the end of the chain, thus increasing the chances of propagating the infection to the output.

2. A variable v is *live* at a program point n if there is a definition-clear path from n to a use of v.

3. A variable v is *dead* at a program point n if v has no definition-clear path from n to any use.

In Phase 2 of our algorithm in Figure 3, lines 16–22 compute the state requirements for all chains. Phase 2 uses the function *PSE*, which performs *partial symbolic execution*, a form of symbolic execution that starts at an arbitrary point in the program instead of the entry point, and uses as input the state variables at that point [1]. Line 17 is explained in Section 3.3. Line 18 calls *PSE* for each chain in P', obtaining symbolic states S' and S. S' contains the path conditions for traversing the chain to its end point in P', whereas S contains the path conditions to reach the same point in P through *all* paths. Path conditions are accessed with function *pc*, whereas symbolic values of live variables are accessed with function *live*. Note that the algorithm regards Conditions 1 and 2 as separate requirements (lines 19–20) because we consider that both exhibit unique change-propagation qualities.

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

3.3. Multiple-change Requirements

In this section, we extend our technique to handle multiple changes by accounting for their effect on one another. Given a change c and a state requirement r for c, we assume that the set of variables V that appear in r's state constraints have the same values at the entry of c in P and P'. Hence, if a test case executes another change c' before c, and c' causes one or more variables in V to have a different value at the entry of c, r is no longer applicable for that test case. More precisely, a change c' *infects* a state requirement r at change c if both changes are executed, c' executes before c, and there exists a variable used in r whose value at the entry of c is infected by c'. Our technique handles these cases by not considering the state constraints for r as covered if any of the variables used in r is infected at the entry of the corresponding change.

Consider program $E_{1,2}$, for instance, obtained by applying `change 1` and `change 2` to our example in Figure 1. The state requirement for chain $(5, 6, y)$ from `change 2` requires for the value of y at statement 6 to be different in E and $E_{1,2}$ due to the additional execution of $y *= 2$ in $E_{1,2}$. However, the value y_0 of y at the entry of `change 2` will be different in E and $E_{1,2}$ because `change 1` executes first and alters y_0. The coverage of a chain requirement in these circumstances does not qualify as coverage of all requirements for that chain; the corresponding state requirement remains unsatisfied.

A more conservative but simpler alternative is to assume that the whole state of the program is infected after executing a change. In this case, our technique statically classifies changes in two categories: (1) changes for which there is a *change-clear* path, which is a path from the entry of the program to the change containing no other change, and (2) changes for which there is no such path.

Our technique computes state requirements only for changes in the first category. Changes in this category are identified by the call to *existsClearPath* at Line 17 in algorithm COMPUTEREQS (Figure 3). Function *existsClearPath(P, s, C)* returns *true* if there is a path from the entry of program P to statement s (excluding s) that contains no change in C. At runtime, our technique monitors state requirements until a change is executed, after which our technique monitors only chain requirements.

To illustrate, in $E_{1,2}$ in Figure 1, `change 2` can only execute after `change 1`. Therefore, `change 2` is in the second category, and only chain requirements are monitored for it. However, if we imagine the original program E in Figure 1 with changes at statements 2 and 4 instead, neither of these changes affects the other. Such changes are in the first category, and the technique produces state requirements for both changes. Furthermore, monitoring of these requirements is never disabled at runtime.

4. Empirical Evaluation

In this section, we present a set of empirical studies we performed to evaluate our test-suite augmentation technique. Section 4.1 describes the toolset we implemented for our technique. Section 4.2 presents our studies, results, and analysis for single and multiple changes. Finally, Section 4.3 discusses threats to the validity of our evaluation.

4.1. The MATRIXRELOADED Toolset

We implemented the MATRIXRELOADED toolset in Java, using the Soot Analysis Framework (http://www.sable.mcgill.ca/soot/) to analyze and instrument software in Java bytecode format. MATRIXRELOADED consists of two main parts: (1) a program analyzer, which implements algorithm COMPUTEREQS (see Figure 3) to identify change-testing requirements, and (2) a monitoring component, which instruments a modified program and monitors at runtime the coverage of our change-testing requirements.

We implemented two key technologies within our program analyzer: a dependence analyzer and a symbolic-

execution engine. Our dependence analyzer identifies control and data dependences from any point in the program. Our symbolic-execution engine supports *partial symbolic execution* [1], which systematically explores all paths between a pair of starting and ending points in the program. The resulting symbolic states are expressed in terms of the symbolic values of live variables at the entry of the starting point. In our current implementation, the symbolic executor performs no more than one iteration per loop and replaces Java libraries with simple approximations. To improve the efficiency of the symbolic executor, our implementation leverages lazy-initialization [19].

To perform our empirical evaluation, we also developed a supporting tool that creates, given an existing pool of test cases, test suites that satisfy, to the largest extent possible, different change-testing criteria, including ours.

4.2. Studies

Table 3 lists the subject programs we used in our studies, their sizes, and the number of test cases and changes provided with each subject. The changes we considered are faults that were seeded in the subjects by other researchers. Our first subject, Tcas, is an air traffic collision-avoidance algorithm used in avionics systems. We used a version of Tcas translated to Java from the original version in C, and considered its first six changes. As a second subject, we used releases v1 and v5 of NanoXML, an XML parser, available from the SIR repository (http://sir.unl.edu). We considered all changes from v1 and two changes from v5.

These two subjects represent software of different nature and complexity. Tcas represents modules with straightforward logic that can be found in avionics systems and other industrial domains. NanoXML, in contrast, represents more complex, object-oriented software, and relies on Java libraries. Note that, because our technique focuses on changes and their effects at a certain distance, program size does not affect the scalability of our approach, but mainly its effectiveness.

We performed our studies on an Intel Core Duo 2 GHz machine with 1 GB RAM.

4.2.1. Study 1: Effectiveness for single changes. First, we studied the effectiveness of our technique in detecting output differences after selecting adequate test suites for single changes, and compared this effectiveness with the effectiveness of other change-testing strategies. We used the following criteria:

1) **stmt**: changed statements
2) **ctrl**: control-only dependence chains
3) **data**: data-only dependence chains
4) **chain**: our chain requirements (Section 3.2.1)
5) **full**: our chain and state requirements (Section 3.2.2)

Table 3. Description of experimental subjects

program	description	LOC	tests	changes
Tcas	air traffic	131	1608	6
NanoXML-v1	XML parser	3497	214	7
NanoXML-v5	XML parser	4782	216	2

We selected Criteria 1–3 for comparison against our technique because they are representative of previous research other than ours. Criteria 4 is a simplified version of our technique that considers chain requirements only. Finally, Criteria 5 is our technique, which considers both chain and state requirements. Studying Criteria 4 helps us assess the extent to which the two components of our requirements, chains and state, contribute to the effectiveness of our technique.

When applicable (i.e., for all criteria except the first), we used the greatest distance our tool could analyze within the memory constraints of the machine used. On the performance side, our tool always took less than eight minutes to successfully analyze a pair of program versions and generate requirements. Thus, for distances within the reach of our tool, the time required to generate our requirements was not an issue.

For each criterion and distance, we randomly generated 100 different satisfying test suites from the pool of available test cases, determining for each test suite whether it produced any difference in the output with respect to the unchanged version of the subject. On average, the maximum size of a test suite for a change in Tcas was 5.8 test cases, whereas the maximum for NanoXML was 2.4 test cases. Tables 4 and 5 present, for Tcas and NanoXML, respectively, the average *difference detection* (ability to reveal a difference in the output) for the test suites generated for each criterion and change. In each table, the first column identifies the change *ch*, the second column shows the maximum distance *d* from the change our tool analyzed, and the remaining five columns show the percentage of *difference-revealing* test suites (i.e., test suites revealing a difference in the output) for the studied criteria. The last two columns, corresponding to our technique, are separated from the rest by a double line.

For the six changes studied in Tcas, Table 4 shows that detection effectiveness rates for statement coverage are low—between 0.7% and 41.1%—with all but one below 16%. Control- and data-dependence chains provide almost no improvement. In contrast, our technique increases the detection rate considerably in most cases. Some improvements are due to the chain requirements alone, such as in the case of change 4. In other cases, such as for changes 1, 2, and 6, the improvements observed are due to the addition of state requirements (i.e., our full technique) to chain requirements. Nevertheless, there are cases in which, despite the improvements, detection rates remain low (e.g., changes 3 and 5) after applying our technique.

Table 4. Difference detection for Tcas

ch	d	stmt	ctrl	data	chain	full
1	3	41.1%	41.1%	47.3%	47.3%	100%
2	6	15.2%	15.2%	15.2%	16.9%	54.1%
3	6	2%	3.2%	2%	18.4%	18.6%
4	6	10%	19.4%	10%	100%	100%
5	6	0.7%	0.7%	0.7%	3.5%	3.7%
6	5	2.1%	3.7%	2.1%	4.6%	100%

Table 5. Difference detection for NanoXML

ch	d	stmt	ctrl	data	chain	full
v1-1	2	52.2%	100%	52.2%	100%	100%
v1-2	3	67.2%	67.2%	67.2%	67.2%	67.2%
v1-3	3	0%	0%	10.4%	10.4%	10.4%
v1-4	3	13.4%	13.4%	48.7%	48.7%	100%
v1-7	3	18.7%	18.7%	18.7%	18.7%	18.7%
v5-3	4	35%	35%	35%	66%	66%
v5-4	4	21.9%	21.9%	21.9%	42.2%	42.2%

In such cases, we hypothesize that the distances we used may not be sufficient to provide acceptable confidence of propagation of the infection to the output; greater distances, which would require enhanced tool support and hardware resources, may be necessary.

In NanoXML (Table 5), we studied all seven changes provided with release *v1* of this subject. Changes in NanoXML are named *vX-N*, where X is the release (1 or 5) and N is the change number (e.g., 1–7 in *v1*). Changes *v1-5* and *v1-6* are of little interest because all test cases that cover these changes reveal an output difference, with no need for further requirements. Therefore, we omit them in Table 5. To increase the number of interesting changes considered, we added two changes from release *v5* of NanoXML. The first two changes in *v5* are located in unreachable code, as we confirmed manually, so we included changes *v5-3* and *v5-4* instead.

Our technique shows an improvement in detection with respect to simpler criteria for 3 out of 7 changes listed in Table 5: *v1-4*, *v5-3*, and *v5-4*. Control- and data-dependence chains, which are subsumed by our chain requirements, sufficed in *v1-1* and *v1-3* for improving on simple change coverage, although for *v1-3* the detection was still low. Meanwhile, simple change coverage showed the same performance as the other criteria for the remaining *v1-2* and *v1-7* changes. By manually inspecting the code we found that, in these two cases, (1) the infected states can reach an output, (2) the dependence chains from the changes to the output are longer than the maximum distance our toolset could analyze, and (3) the existing test cases that satisfy our requirements do not always propagate the infected part of the state to the output. In other words, our technique pushed the infected state up to a certain distance, often achieving propagation to the end, but could not guarantee that the output itself would be infected.

The improvements due to our technique in the case of NanoXML are not as dramatic as in the case of Tcas. There are important differences between the two subjects that help explain these results. First, NanoXML makes extensive use of strings and containers, and we found that the infected state is often confined within objects of these types. Because our current implementation replaces library objects with simpler approximations, it is limited in generating requirements for such objects. Second, NanoXML makes heavier use of heap objects and of polymorphism than Tcas, therefore imposing a considerably higher burden on the lazy-initialization part of symbolic execution. These complications make it more difficult for our current implementation of dependence analysis and symbolic execution to operate on NanoXML than on Tcas, as reflected by the shorter analysis distances achieved on NanoXML. Nevertheless, our results for NanoXML, even at short distances, are promising; they show that our approach can exercise more change-related behaviors in real object-oriented software, compared to traditional change-testing criteria. Our results for Tcas are also promising because they indicate that our technique may be quite effective for modules similar to Tcas, but within larger software.

4.2.2. Study 2: Multiple-changes case. In this section, we study a case of interaction between two changes, and the consequences on requirements monitoring and difference detection when applying the more conservative version of our multiple-change handling approach to these changes.

An interesting interaction occurs between changes *v1-1* and *v1-4* in NanoXML-v1. Change *v1-1* is located in the DTD (Document Type Definition) parsing component, whereas change *v1-4* alters the final step of XML element attribute processing. We call P_1 the program version with change *v1-1* only, P_4 the version with change *v1-4* only, and $P_{1,4}$ the version with both changes. Out of 214 test cases available, 42 test cases show an output difference in $P_{1,4}$. As expected from the results in the previous study, all test suites we created for our technique obtained a 100% difference detection for $P_{1,4}$.

In $P_{1,4}$, change *v1-4* is executed by 164 test cases, out of which only seven test cases are difference-revealing. In all seven difference-revealing test cases, change *v1-4* is infected by change *v1-1*. (Although 135 test cases reach change *v1-4* first, none of these cases reveal a difference.) Therefore, state requirements in $P_{1,4}$ for change *v1-4* are never satisfied, even though many test cases cover this change without infection. In P_4, in contrast, change *v1-4* is never infected and is covered by 26 difference-revealing test cases. Because in P_4 change *v1-4* greatly benefits from satisfying state requirements over chain requirements alone (see Table 5), we would expect in $P_{1,4}$ a reduction in the ability of our technique to detect differences in test cases that cover change *v1-4*.

To measure the impact of infection on change *v1-4* in $P_{1,4}$, we executed all 164 test cases that cover change *v1-4*

in this version, and we measured the coverage of our chain requirements only for change *v1-4*. From these 164 test cases, we generated 100 unique test suites satisfying our chain requirements, of which 23.7% revealed a difference. In contrast, difference detection in $P_{1,4}$ for simple coverage of change *v1-4* was 3.6%. Considering that our technique achieves 100% detection for change *v1-4* in P_4, there is a considerable decrease of 76.3% in the detection ability of our technique for change *v1-4* when combined with change *v1-1*, due to infection from change *v1-1*. Yet, our technique maintains in this scenario an advantage of 20.1% over simple change coverage.

4.3. Threats to Validity

The main internal threat to the validity of our studies is the potential presence of implementation errors in our toolset and experimental setup. To reduce this threat, we tested and debugged our system with example programs, including the subjects of our study.

The main external threat for our studies is that we studied only two subjects and a limited number of changes, due to the complexity of the technologies involved. More subjects and changes need to be studied to properly assess the applicability and expected benefits of our technique.

5. Related Work

The predecessor of our technique, MATRIX [1], is the work most closely related to our current technique. In this previous work, we introduced the concepts of partial symbolic execution and symbolic state differencing for modified software. That early work focuses on single changes and presents only a preliminary study with a partial implementation that does not include symbolic execution or chain coverage. In this paper, we extend our previous technique by specifying chain requirements and handling multiple changes. We also present an improved implementation of our technique and perform a more extensive study.

Several other techniques are related to test-suite augmentation and change-testing criteria. These techniques can be classified into three categories.

Techniques in the first category define general testing criteria for software in terms of coverage of program entities, such as statements, branches, and definition-use pairs (e.g. [6], [11], [13]). These criteria can be adapted, as we did in our study, for modified software by considering only changed statements or affected control or data dependence chains. Our empirical studies show that test suites satisfying our state and chain requirements have a greater likelihood of revealing different behavior than those that are based on control or data flow only.

Techniques in the second category share with our technique the overall goal of generating requirements for testing changes. Binkley [3] and Rothermel and Harrold [15] present techniques that use System Dependence Graph (SDG) based slicing [9] to produce testing requirements involving individual data- and control-flow relations affected by a change. Gupta and colleagues [7] propose a technique based on an on-demand version of Weiser's slicing algorithm [21]. These techniques have a number of limitations. SDG-based techniques incur high computing costs and are memory-intensive in computing summary edges. Furthermore, these techniques generate requirements that involve complete chains of control and data dependences from a change to output statements, which are likely difficult to satisfy because these chains may include most of the program. Our technique differs in two main ways. First, our technique controls the cost by computing dependences only up to a given distance from a change. Second, our technique incorporates state requirements for increased effectiveness. As our study results suggest, testing requirements based on short-distance dependences can be practical and effective at the same time.

Techniques in the third category generate requirements for fault-based testing that integrate propagation conditions. Richardson and Thompson present the RELAY framework [14], which computes a precise set of conditions to propagate the effects of faults to the output. Morell [12] presents a theory of fault-based testing, which uses symbolic execution to determine fault-propagation equations. The goal of these techniques is to propagate the effects of faults, not changes, to the output. Moreover, they rely on symbolic execution of an entire program, which is generally impractical. Our technique, in contrast, limits the generation of testing requirements to the selected distance, and these requirements incorporate conditions that guarantee propagation up to that distance. Furthermore, our technique does not need to solve such conditions to compute the adequacy of a test suite. Our technique only checks whether these conditions are satisfied at runtime.

6. Conclusion and Future Work

In this paper, we presented our test-suite augmentation technique that improves upon our previous approach, MATRIX [1], in two important ways: (1) generating requirements for propagation of the effects of changes along different dependence chains, which increases the chances of revealing new or affected behaviors, and (2) handling multiple changes, which makes the approach applicable in realistic scenarios.

Our technique has two main applications: assessing the adequacy of a regression test suite after changes are made in a program, and guiding developers and tools in generating new test cases that exercise untested behaviors

introduced by the changes. To achieve these goals, our technique leverages dependence analysis, partial symbolic execution [1], symbolic state differencing, and runtime monitoring. In this way, our technique can identify and check coverage of thorough requirements for change-propagation within a given distance, which can increase the chances of propagating erroneous states to the output while still remaining practical.

We also described a toolset that implements our technique and presented two empirical studies performed using this toolset. The first study shows that, for two subjects, our technique can considerably improve the chances of producing output differences. The improvement in effectiveness depends on the complexity of the subject, the characteristics of the changes involved, and the distance used. Chain requirements alone appear to be more effective than simpler criteria in many cases, whereas state requirements provide a strong complement to chain coverage in other cases. The second study shows how our technique can handle scenarios with multiple changes, which allows changes to be tested with some degree of effectiveness even when they are affected by other changes.

We believe that the theoretical foundations of our technique and the results of our studies encourage future research that could further improve the effectiveness of test-suite augmentation based on change-infection propagation. In particular, we can identify several directions for future work. On the experimental side, we expect to strengthen our partial symbolic-execution engine to generate requirements at longer distances, cover more paths, and better handle encapsulation in object-oriented software. We are optimistic about the scalability of our technique because its costs and limitations depend mostly on the complexity of the program in the vicinity of changes, rather than on the overall size of the program. A second direction for future work is to leverage partial symbolic execution and program state comparison in novel ways for specifying additional, complementary requirements that can guide test-suite augmentation. Another direction for future work is the investigation of ways of automatically generating test cases that can cover untested new behaviors identified by our requirements. Finally, static and dynamic analyses could be used to determine in a more precise way whether the state at the entry of a change has been infected by another change and use that information to improve our approach for handling multiple changes.

Acknowledgements

This work was supported in part by Tata Consultancy Services and by NSF awards CCR-0306372, SBE-0123532, and CCF-0725202 to Georgia Tech. The anonymous reviewers provided useful comments and suggestions that improved the presentation of this paper.

References

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirement identifier and examiner. In *Proc. of Testing and Academic Industrial Conf. Practice and Research Techniques (TAIC PART)*, pp. 137–146, Aug. 2006.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. of ACM Symp. on Principles of Prog. Lang.*, pp. 384–396, Jan. 1993.

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

[4] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. Testtube: A system for selective regression testing. In *Proc. of Int'l Conf. on Softw. Eng.*, pp. 211–222, May 1994.

[5] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Systems*, 9(3):319-349, July 1987.

[6] P. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Trans. on Softw. Eng.*, 14(10):1483-1498, Oct. 1988.

[7] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *Journal of Softw. Testing, Verif., and Reliability*, 6(2):83–111, June 1996.

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

[9] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Prog. Lang. and Systems*, 12(1):26-60, Jan. 1990.

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

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

[12] L. Morell. A Theory of Fault-Based Testing. *IEEE Trans. on Softw. Eng.*, 16(8):844-857, Aug. 1990.

[13] S. C. Ntafos. On required element testing. *IEEE Trans. on Softw. Eng.*, 10(6):795–803, Nov. 1984.

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

[15] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proc. of Int'l Symp. on Softw. Testing and Analysis*, pp. 169–184, Aug. 1994.

[16] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.

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

[18] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of Int'l Symp. on Softw. Testing and Analysis*, pp. 97–106, July 2002.

[19] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proc. of TACAS 2004. SIGSOFT Softw. Eng. Notes*, pp. 97–107, Mar. 2004.

[20] J. Voas. PIE:A Dynamic Failure-Based Technique. *IEEE Trans. on Softw. Eng.*, 18(8):717–727, Aug. 1992.

[21] M. Weiser. Program slicing. *IEEE Trans. on Softw. Eng.*, 10(4):352–357, July 1984.

[22] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proc. of Int'l Conf. on Softw. Eng.*, pp. 41–50, Apr. 1995.