Applying Aggressive Propagation-based Strategies for Testing Changes

Raul Santelices and Mary Jean Harrold
College of Computing
Georgia Institute of Technology, Atlanta, USA
e-mail: {raul|harrold}@cc.gatech.edu

***Abstract*—Test-suite augmentation for evolving software—the process of augmenting a test suite to adequately test software changes—is necessary for any program that undergoes modifications as part of its development and maintenance cycles. Recently, we presented a new technique for test-suite augmentation based on leveraging the propagation conditions for the effects of changes. Although empirical studies show that this technique can be quite effective for testing changes, the experiments have been limited because of the complexity of the implementation. In this paper, we present a new and more efficient approach for propagation-based testing of changes that can reach much longer propagation-distances and can focus the testing more precisely on those behaviors of changes that can actually affect the output. Using an implementation of this new approach, we performed a study on a set of changes on Java programs for which we compared, to a much larger extent, our propagation-based strategy with other existing techniques for testing changes. The results of the study not only confirm the superior effectiveness of propagation-based strategies over these other techniques for testing changes, but also quantify that superiority and clarify the conditions under which our approach is most effective.**

I. Introduction

Changes to software are problematic for developers and testers because the changes might not behave as expected or may introduce erroneous side effects. After each change cycle, *regression testing* is used to find errors introduced by changes and gain confidence that the changes and the parts of the program affected by those changes behave correctly. Typically, in regression testing, developers reuse an existing test suite T to test the modified program. However, the new functionality or side effects introduced by changes are not necessarily tested adequately by T, so developers must *augment* T (i.e., add new test cases to T) to exercise those untested or undertested behaviors of changes, obtaining a new test suite T'. In previous work, we called this process *test-suite augmentation for evolving software*.

Existing techniques for test-suite augmentation address this problem by identifying program entities potentially affected by changes and requiring their coverage by the augmented test suite. Early techniques [3] [14] use control- and data-dependence analysis [5] [20] to identify entities such as branches and definition-use pairs (du-pairs) that must be covered in the modified program because their execution behavior may be affected by the changes. These techniques, which we call *coverage-based strategies*, resemble typical testing criteria for programs in which entities of a certain type need to be covered. Recent approaches for test-suite augmentation capture the conditions (or a subset) under which the state modifications caused by executing a change propagate and affect other parts of the program [2] [15] by using symbolic execution [9] in addition to dependence analysis. These approaches, which we call *propagation-based strategies*, give testers stronger guarantees than earlier techniques that an adequate variety of effects of changes will be exercised and that those effects will be observed at the output and other points of interest to let testers assess the correctness of changes in different scenarios.

Unfortunately, however, only limited empirical studies have been performed to evaluate the effectiveness of test-suite augmentation strategies, in general, and the improvements achieved by these strategies over coverage-based strategies, in particular. For propagation-based strategies, a major reason for this limitation is the implementation and runtime complexity of combining dependence analysis and symbolic execution. Whereas promising results have been obtained for such strategies [15] [17], the extent of the propagation conditions that can be computed in practice using the current formulations of these techniques is limited to a few dependencies away from each change location.

To evaluate and analyze the real strength of propagation-based strategies for testing changes, we developed, and present in this paper, a new, more aggressive, and more efficient approach for augmenting test suites. Our new approach is equivalent in power to existing propagation-based approaches [2] [15] but achieves much longer propagation distances in practice by using dynamic slicing and monitoring state changes on demand rather than computing propagation conditions beforehand. Using this new approach, we also present in this paper a study of test-suite augmentation on a number of changes in different Java programs. This study evaluates and compares propagation-based strategies with coverage-based strategies to a much larger and conclusive extent than the limited data reported in the literature.

The results of this study let us confirm the superiority of propagation-based strategies over coverage-based strategies for testing changes in overall effectiveness (i.e., number of observable differences revealed by the resulting test suites) and cost-effectiveness (i.e., effectiveness per test added to the test suite). These results also quantify that superiority

and describe the conditions under which propagation-based strategies are most cost-effective. For example, for test suites of size 10, our results show that, on average for the studied subjects, our propagation-based strategy reveals two times more differences per change than randomly creating 10 test cases that cover the change and about twice the differences found by using coverage of affected branches and du-pairs. In terms of cost-effectiveness, our results also show that, on average, any test case in a test suite of any size for our approach is 91% more likely to reveal a difference than randomly testing that change and 70–90% more likely than for branch and du-pair coverage.

Another important result of this study is that the cost-effectiveness of using our new propagation-based strategy, in terms of differences revealed per test case added, is especially superior to the other techniques when the probability that executing a change will cause a difference in the output is the lowest. In other words, we found that our approach is, comparatively, even more effective when it is most needed—when random testing of the change or even branch or du-pair coverage are unlikely to find differences.

One benefit of our work is that it provides a new propagation-based approach to test changes that is computationally more efficient than existing approaches that analyze all possible behaviors of changes, most of which are not tested in practice. For that reason, our approach can analyze longer propagation distances than was previously possible for the same program-analysis budget. Thus, testers can achieve a greater effectiveness in revealing observable differences when augmenting their test suites by using this new approach. Another benefit of our work is the first comprehensive empirical study of test-suite augmentation strategies of any kind, including propagation-based strategies that reach greater distances than in previous studies. This study provides testers with quantitative information about the effectiveness they can expect of different strategies for test-suite augmentation and the kinds of changes for which these strategies are most effective.

The main contributions of this paper are:

- A comparative analysis of existing test-suite augmentation strategies for evolving software.
- A new approach for performing propagation-based testing of changes that makes this testing strategy more efficient and able to reach greater propagation distances.
- A comprehensive study of coverage-based and propagation-based strategies for testing changes that shows that our propagation-based strategy:
  - reveals more differences caused by changes than coverage-based strategies and is also more cost-effective (more differences per test case) and
  - is particularly superior to the alternatives when the difficulty of making changes cause output differences is low.

II. Initial Concepts and Example

In this section, we discuss the relationship between coverage criteria and testing strategies (Section II-A) and introduce an example used in the rest of the paper (Section II-B).

A. Test Adequacy Criteria and Testing Strategies

Many researchers have addressed the adequacy levels for testing programs based on coverage criteria. These criteria define which kinds of entities in the program must be *covered* (i.e., executed at least once) during testing [6]. Some of these criteria require that control-flow entities such as statements or branches are covered, whereas other criteria require the coverage of data-flow entities such as *definition-use pairs* (du-pairs) (see PDG in Section II-B).

For modified programs, the emphasis is not on testing for faults hidden anywhere in the program, but on finding faults caused or uncovered by changes. To assess the correctness of changes, developers and testers must sample an adequate set of differences in the behavior space of the modified program. The measure of success for this testing, in addition to the number of regression faults detected, is the number and variety of differences in behavior exercised—the more and better-distributed kinds of differences detected by testing changes are, the more likely it is that faults related to specific behaviors of changes are detected. For testing modified programs, coverage-based and propagation-based adequacy criteria for test-suite augmentation [14] [15] have been defined (we discuss them in Section III).

One problem with adequacy criteria is that it is often impossible to achieve 100% satisfaction because some test requirements are either *infeasible* (i.e., not coverable by any execution) [6] or too difficult and expensive to cover. Thus, in reality, testers must balance software-quality needs with budget constraints. In practice, we have observed through our industrial partners that rarely, if ever, are "white-box" coverage criteria used. However, it is still crucial to provide testers with guidance that distributes well the testing effort over the space of all behaviors of the program, even if a complete sample for some criterion is not possible. For that reason, in this paper, we use the term *testing strategy* to refer to the use of a test-adequacy criterion (based on coverage or propagation) to guide the creation of new test cases that augment test suites in a well-distributed way.

B. Example Program with Changes

Figure 1 presents the example that we use throughout this paper. The first two columns in the figure list the program, consisting of classes `M`, `B`, `P`, and `Q`, where `P` and `Q` are subclasses of `B`. The entry of the program is the method `M.main`. In lines 1 and 2, the program creates instances of `P` and `Q` that are assigned to references `p` and `q`, respectively. In each case, the constructor of `B` receives arguments `a` and `b`. At line 3, the program decides whether it continues to line 4 or exits. At line 4, one of the two references is assigned to

```
   class M {
     void main(int a, b, c) {
 1:  B p = new P(a); // ch1: a -> -a
 2:  B q = new Q(b); //
 3:  if (a < c || b < p.x) {
 4:    B r = (c%2==0)? p : q;
 5:    if (p.x > 10) {
 6:      if (r.x > 5)
 7:        p.foo(q.x);
         else
 8:        r.foo(q.x);
       }
       else
 9:      print r.foo(0);

10:    while (q.x > c)
11:      print q.x--;
     }}}
```

```
   class B {
     int x;
12:  B(_x) { x = _x; }
     int foo(y) {
13:    if (x > y)
14:      x -= 100*y;
       else
15:      x++;
16:    return -y;
     }
     ...
   }
   class P extends B {
17:  int foo(y) { P.foo(y+1); }
     ...  }
   class Q extends B {
18:  int foo(y) { return y*2; }
     ...  }
```

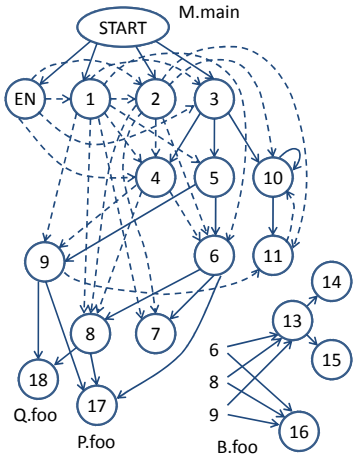

Figure 1. Example program consisting of classes M, B, P, and Q. M has two changes. On the right, a partial PDG for this program.

`r`, depending on the value of `c`. At lines 5–9, conditions on the value of the field `x` of the instances of `P` and `Q` determine which call to `foo` is performed—the call at 7, 8, or 9. Finally, a loop at lines 10–11 prints and decrements `q.x` until it is no longer greater than `c`.

The right of Figure 1 shows an almost complete *Program Dependence Graph* (PDG) [5] for this program. (In general, an interprocedural form of the PDG is needed [7].) The nodes in this PDG are the statements of the program and two special nodes: START for the entry of the program and EN for the entry of `M.main`. In this PDG, we omit the constructor of `B` and parts of `B.foo`.

A solid edge in the PDG represents the control dependence of the target node on the source node. Node n_1 is *control dependent* [5] on node n_2 if the decision taken at n_2 determines whether n_1 is necessarily executed. Control dependencies are created by the presence of branches and other control decisions such as virtual calls. In Figure 1, an example is node 8, which is control dependent on node 6 taking the false branch. In the PDG, the edges coming out of START represent the decision of executing the program. The bottom right part of this figure shows, separately, a partial view of the control-dependencies for `B.foo`.

A dashed edge in the PDG represents a *data dependence* of the target node on the source node, indicating that a variable defined at the source node is used at the target node and that there is a definition-clear path in the program (i.e., a path that does not re-define the variable) between the source and target nodes. Such a pair of nodes, including the variable, is called *du-pair*. For example, in Figure 1, node 6 is data dependent on node 4 because 4 defines `r`, 6 uses `r`, and there is a path (4,5,6) that does not redefine `r` after 4. For space reasons, in this figure, we omitted the interprocedural data dependencies.

III. Existing Change-Testing Strategies

In this section, we compare change-testing techniques proposed in the literature, which we classify as *coverage-based* (Section III-A) or *propagation-based* (Section III-B).

A. Coverage-based Strategies

Researchers have proposed to test the effects of changes by conservatively identifying entities, such as du-pairs, that might be affected by those changes, and requiring the coverage of those entities by a test suite [3] [14]. These techniques identify the affected entities using forward traversal of control and data dependencies from the locations of the changes, which is essentially performing forward slicing [7] [20] from the changes. For example, in Figure 1, and following the approach of Reference [14], the test requirements for change `ch1` at line 1 are the du-pairs in the forward slice for variable `a` at that line. This slice is obtained by traversing the PDG forward from node 1 through both solid and dashed edges and identifying as test requirements the reachable dashed edges—the du-pairs. (From node 1, only the dashed edges for variable `a` are considered.) In this example, the test requirements for `ch1` include du-pairs $\langle$(1,`a`),(12,`_x`)$\rangle$ (i.e., the definition of `a` at 1 and its use as `_x` at 12), $\langle$(12,`p.x`),(3,`p.x`)$\rangle$, and $\langle$(4,`r`),(6,`r`)$\rangle$—this du-pair is affected through the control dependence of 4 on 3, which is the target of $\langle$(12,`p.x`),(3,`p.x`)$\rangle$.

In this paper, we use DU to denote the testing strategy of covering du-pairs affected by a change until all du-pairs are covered or testing reaches a budget limit (e.g., no more coverage is achieved after creating 10 candidate test cases for the test suite). Similarly, we denote by BR the strategy of covering branches affected by changes. Note that the techniques in the literature do not require that the affected entities be covered after the change has been executed, but we will assume in this paper, as a basic condition for BR and DU, that they must be covered after the change. For completeness, we also introduce the RANDOM strategy that simply requires the execution of all changed statements.[1]

[1] RANDOM indicates "random testing of the change"—after reaching the change, the execution characteristics of the test cases are left to chance.

To analyze change-testing strategies, including BR, DU, and the strategies presented in the next section, we use the PIE model [19] adapted for changes [15]: a change reveals a difference in the output if and only if the change **e**xecutes, **i**nfects the state (i.e., creates an *infection*—a difference in the program state), and the infection **p**ropagates to the output. In the example of Figure 1, change `ch1` is executed always (satisfying the execution condition), the state is infected if `a` does not equal 0, and the infection propagates to the output if, for example, line 9 is reached such that `r` equals `p` (i.e., when `c` is even), so that line 9 prints `-a` instead of `a`.

The goal of coverage-based strategies is to execute potentially-faulty entities—satisfying the first condition of the PIE model—and hope that, if the entity is faulty when affected by the change, it will infect the state and the infection will propagate to the output. It is also expected that the change itself is executed first, that it infects the program state, and that this infection propagates to the entity. (Otherwise, assuming that the program was correct before the changes, if no infection from the change reaches the entity, the entity cannot cause an error.) Thus, one limitation of coverage-based strategies is that they satisfy only the execution conditions for changes and entities, leaving to chance the infection and propagation conditions needed to reveal potentially-faulty differences. In other words, these strategies satisfy some necessary but not sufficient conditions for revealing the effects of changes.

Another limitation of existing research on coverage-based strategies is that only subsets of the test requirements they identify have been studied and that those studies are small in scale [15]. In Section V, we remedy this situation.

B. Propagation-based Strategies

To address the limitations of coverage-based strategies, propagation-based techniques have been developed [2] [15]. The method, called MATRIX, computes test requirements in two phases. These requirements are propagation-based because they contain not only the execution condition of the PIE model for each change, but also represent at least some of the infection and propagation conditions from this model. Next, we analyze the two phases of MATRIX.

1) Phase 1: This phase of MATRIX identifies, for each change, all *dependence chains* (i.e., sequences of consecutive dependences in the PDG) starting at that change and uses these chains as test requirements. Each of these chains represents a "propagation path" in the PDG from the change to the output because dependencies are the means through which infections propagate from statement to statement until they reach the output. However, in practice, the number and length of these chains can be too great or even infinite. For that reason, MATRIX uses a length limit d for the chains (the "distance from the change"). Despite this limit, executing a chain guarantees that a propagation path is taken at least up to distance d, which prevents cases in which infections stop propagating before d because a non-propagating path is taken (e.g., a path that re-defines an infected variable before it is used). Thus, chains increase the chances that a propagation path is covered to the output. In this paper, we use CHAIN$_d$ to represent the strategy of satisfying chain requirements for a distance d.

For example, in Figure 1, one dependence chain from `ch1` consists of the PDG edge sequence q_1= ⟨1,12⟩,⟨12,3b⟩,⟨3b,4⟩,⟨4,4p⟩,⟨4p,9⟩[2] where ⟨3b,4⟩ and ⟨4,4p⟩ are control dependencies and the rest are data dependencies. A necessary condition for an infection in `p.x` at line 1 to propagate to line 9 through `r` is to execute this chain because it reaches line 9 and its subsequence ⟨4,4p⟩,⟨4p,9⟩ guarantees that `r` is the same as `p` at 9. Using DU, in contrast, does not guarantee that du-pairs ⟨12,3b⟩ and ⟨4p,9⟩ are covered by the same execution—a test t_1 might cover ⟨12,3b⟩ but take the false branch at 3b whereas, in another test t_2, 3 might be true because `a<c` (i.e., `p.x` at 3b is not reached), so ⟨4p,9⟩ is covered but not infected.[3]

Note that, whenever chain q_1 is covered, the infection from `ch1` will propagate to the output in line 9, so q_1 is a sufficient condition for that propagation. However, this is not the case for all chains. For example, covering chain q_2=⟨1,12⟩,⟨12,3b⟩,⟨3b,5⟩,⟨5,9⟩ with inputs `a`=3, `b`-10, and `c`=-10 results in the same output for the original and the modified program (P and P', respectively) because, in both cases, line 3 evaluates to true, line 5 evaluates to false, and `r` is `q` at line 9 (`q.x` is not infected). Thus, while being a necessary condition for a propagation through those dependencies, covering q_2 is not sufficient.

2) Phase 2: To obtain the sufficient conditions for propagating an infection along a dependence chain, Phase 2 of MATRIX adds to each chain from Phase 1 a set of propagation constraints computed using *partial symbolic execution* [2] [15] [17] which starts at the change instead of the program's entry. These constraints are divided into two parts, one for each of the two cases in which a chain propagates an infection. In the first case, an infection propagates through a chain if, for the same input, the chain is covered in P or P', but the end point of that chain is not reached in the other version of the program. In the second case, the chain is covered in one version and its end point is reached in the other version, but the state of the program computed along this chain differs between the two versions.

Despite the limit in distance for the propagation constraints of Phase 2, satisfying these constraints for a chain guarantees that an infection propagates through that chain at least up its end at distance d, which prevents cases in which the infection stops propagating somewhere in this chain. Thus, Phase 2 increases the chances that the infection

[2] 3b is the second clause in line 3 and 4p is the subexpression of 4 that contains `p`, which is reached when the condition at 4 evaluates to true.

[3] CHAIN$_d$, however, does not subsume DU or BR because there can be affected du-pairs or branches beyond distance d from the change.

will propagate all the way to the output. In this paper, we use PROP$_d$ to represent the strategy that enhances CHAIN$_d$ by adding to each chain these Phase-2 constraints.

For example, whenever the chain q_2 (defined above) is covered in P', an infection propagates through q_2 only if the condition for any of the two control dependencies in this chain evaluates differently in P. Thus, for q_2 and the condition at clause 3b, the Phase-2 requirement is that `b<-a` and `b≥a`, in which symbolic execution replaced `p.x` with `a` for P and `p.x` with `-a` for P'. For the condition at line 5 in this chain, the requirement is that `a≤10` and `-a>10`.

A limitation of existing research on propagation-based strategies is that, like coverage-based strategies, they have been studied only to a limited extent [15] [17]. The main obstacle for such studies is the cost of computing these requirements. Thus, to address this problem, in the next section (Section IV), we define a new approach that enables the comprehensive study presented later in Section V.

IV. NEW PROPAGATION-BASED APPROACH

In this section, we present our new approach for propagation-based strategies that addresses the main limitation that hampers their effectiveness and applicability: the short distance limits they achieve in practice [15] [17]. Section IV-A analyzes the causes for this limitation and Section IV-B presents our solution.

A. Causes of Current Limitations

The distance limit d discussed in Section III-B for propagation-based strategies addresses two practical concerns: (1) the great computational costs of computing these test requirements and (2) the need to contain the number of chains and constraints presented to testers. The current formulation of the MATRIX requirements has its benefits—it specifies a comprehensive set of test requirements as constraints that can be evaluated by executing only one of the programs and can be extended backward for solving and generating inputs for them. However, this formulation is also problematic for two reasons:

1) Symbolic execution does not scale well. Advances for multiple paths [17], compositionality [1], and other optimizations [11] have extended the boundaries of symbolic execution, but it remains intractable and, therefore, the distances for Phase-2 are still limited. Moreover, within the distance limits currently reached, for many chains, some paths that cover them cannot be analyzed because of their great length.
2) A large fraction of the testing requirements, identified by the static analysis of P and P' to find all potential effects of a change, are either infeasible or too difficult to satisfy with any reasonable budget. Thus, many requirements are not satisfied in practice, and the effort spent in computing them is wasted.

B. The New Approach

To address these problems, we developed a new and more efficient approach for CHAIN$_d$ and PROP$_d$ that circumvents the complexities of the current version of MATRIX. Instead of computing all test requirements beforehand (i.e., statically), our new approach checks for dependencies and state differences during execution, identifying the satisfaction of requirements on demand without computing them beforehand. This new approach preserves the precision of MATRIX while avoiding the cost of symbolic execution and avoiding most of the effort that is wasted when a full static analysis is performed for requirements that are not satisfied later. Although our new solution does not directly compute constraints for solving, the significant efficiency improvements of the new approach justify this compromise.

Our new approach consists of the following steps:

1) Initialize the accumulated set of satisfied requirements, *Accum*, to the empty set.
2) Compute a static forward slice from the location of each change, on both P and P'.
3) Instrument P and P' to detect the coverage of the dependencies in the slice, including the values written to the program state at the source of each dependence and, for control dependencies, the branch taken.
4) Receive a new test input t from the tester and execute P and P' for that input.
5) During execution, collect the information reported by the instrumentation including, for each of P and P', which dependencies are covered, which other dependencies they succeed when covered, and which values are computed at the source of those dependencies.
6) After execution, for P and P', find which chains were executed by determining which dependencies from the change were covered and, transitively, which other dependencies succeeded those dependencies. Also, determine which computed values differed between P and P' at the source of each dependence in each chain.
7) Update *Accum* with the new chains covered and the new infection-propagations found, if any.
8) Inform the tester whether t satisfied new requirements and, if so, the new status of *Accum*. Also, notify the tester if all requirements for the chains in the static forward slice have been satisfied.
9) Go back to Step 4 if the tester wants to continue.

In a typical usage scenario for this approach, the tester will first reuse the test cases from a regression test suite T and input them in Step 4 to find the requirements already satisfied by T. Then, the tester will iteratively create new test cases t for Step 4, adding each t to T if it satisfies new requirements, until all requirements are satisfied or a testing budget limit is reached (e.g., number of test cases generated that do not satisfy new requirements, person-hours spent, or number of new test cases whose outcome must be inspected).

Table I
SUBJECTS, TEST SUITE SIZES, AND CHANGES.

Subject	Description	LOC	Tests	Changes
Tot_info	information measure	283	1052	8
Schedule1	priority scheduler	290	2650	8
Schedule2	priority scheduler	317	2710	8
Print_tokens	lexical analyzer	478	4130	9
NanoXML	XML parser	3497	214	7

This new approach requires only static and dynamic dependence analysis (i.e., slicing [7] [20] with some refinements) and runtime state monitoring, in contrast with MATRIX, which, in addition, requires symbolic execution for all paths that can propagate the changes.[4] The efficiency improvements of the new approach significantly increase the distances that can be analyzed for CHAIN$_d$ and PROP$_d$, as our study in Section V shows. Also, our approach removes the burden from the tester of handling a large number of short-distance requirements and, instead, provides the tester with information focused on those long-distance requirements that are actually satisfied.

V. STUDY OF CHANGE-TESTING STRATEGIES

In this section, we present our evaluation and comparison of the effectiveness and cost-effectiveness of the change-testing strategies discussed in Sections II–IV.

A. Empirical Setup

In this section, we describe our toolset and the subjects used in our study.

1) Implementation: To evaluate all five change-testing strategies, we used the DUA-FORENSICS dependence-analysis and instrumentation framework [16] which is based on Soot [18] and analyzes Java-bytecode programs. To implement strategies RANDOM, BR, and DU, we reused the existing functionality in DUA-FORENSICS to monitor the coverage of statements, branches, and du-pairs, which we modified for BR and DU so that only branches and du-pairs affected by a change are monitored. Covered branches and du-pairs are reported only after the change has executed.

For our new CHAIN$_d$ and PROP$_d$ strategies, we extended the dynamic forward slicer provided by DUA-FORENSICS to not only identify which entities (i.e., statements and dependencies) are covered after a change but also identify which dependencies precede which other dependencies—this ability lets us precisely monitor the dependence chains within each dynamic forward slice. In addition, to implement PROP$_d$ we used the ability of DUA-FORENSICS to identify state modifications performed by each statement in a dependence chain.

[4]In our new approach, we could remove the static-slicing and dependence-monitoring steps altogether and, instead, analyze the traces of the programs for dependencies and state differences. This alternative would eliminate the need for most of the static analysis, but it would increase the time and space overhead during execution and post-processing. We intend to explore this alternative design in the future.

2) Subjects: For our study, we considered subject programs for which a large number of test cases are available (to properly simulate the creation of new test cases by a tester) and that include changes seeded by other researchers. Therefore, we decided to start our investigation on a number of subjects from the Siemens suite [8] that we translated from C to Java. These subjects are listed in Table I, where the columns show, respectively, the name of the subject, a short description, the size in lines of code (LOC), the number of test cases available, and the number of changes used in our study. The Siemens subjects can be seen as representative of small programs as well as modules from larger programs that are tested as units. In addition, as Table I also shows, we studied NanoXML, a small XML parser used in real-world scenarios and whose coding style and complexity are representative of modern object-oriented programs. We obtained NanoXML from the SIR repository [4].

3) Execution environment: To run our experiments, we used a machine with two quad-core Intel CPUs and 12 GB RAM running 32-bit Linux and the Hotspot 6 Java Virtual Machine.

B. Goals and Design

The goal of our study was to evaluate both the effectiveness and the cost-effectiveness (i.e., effectiveness per cost unit) of the change-testing strategies discussed in sections III and IV and compare these strategies against each other. Our measure of effectiveness for a test suite is its *difference-detection*—the number of test cases whose outputs in the original and modified versions of the program differ. Our measure of cost-effectiveness, or quality, of the test suite is the ratio of difference-detection to the size of the test suite, which we call *ds-ratio*. The greater the ds-ratio, the better the strategy that created the test suite is at sampling and discriminating test cases that can cause differences.

To achieve this goal, we sought to answer the following three research questions:

RQ1. How (cost-)effective are our propagation-based strategies with respect to existing techniques for testing changes?

RQ2. Which factors affect the effectiveness of change-testing strategies?

RQ3. What is the cost of using propagation-based strategies?

For our study, we designated RANDOM as our baseline strategy because we consider useless any strategy that is less effective than randomly testing a change (i.e., creating tests that just cover the change without regard for any other requirement). Also, because only one test case suffices to satisfy RANDOM for each change in our study, we used RANDOM repeatedly to create test suites of the same sizes as those obtained by the other strategies with the purpose of assessing the cost-effectiveness of those test suites.

To evaluate BR, DU, CHAIN$_d$, and PROP$_d$, for each change in each subject, we used the following method for

Table II
AVERAGES FOR DIFFERENCES DETECTED, TEST-SUITE SIZES, AND DS-RATIOS FOR CHANGE-TESTING STRATEGIES.

Strategy	RANDOM	BR	DU	CHAIN$_d$							PROP$_d$									
				d1	d2	d3	...	d8	d9	d10	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10
differences	.24	1.35	6.81	.39	.45	.69	...	4.28	6.08	8.15	.93	1.54	2.81	4.11	5.73	7.45	9.17	11.35	14.31	17.98
size	1.00	6.69	39.71	1.32	2.42	3.67	...	47.31	54.76	71.11	2.50	4.51	7.33	11.33	21.09	33.15	44.04	59.88	69.89	89.11
ds-ratio	.24	.24	.26	.31	.29	.31	...	.31	.31	.30	.42	.45	.48	.49	.48	.47	.45	.43	.42	.41

each of these strategies:

1) For all test cases available for that subject, use DUA-FORENSICS on P and P' (i.e., the original and modified versions) to perform the analysis, monitoring, and reporting of covered (i.e., satisfied) test requirements (e.g., affected branches, chains from the change) as described in the approach of Section IV-B.
2) Using this coverage information, construct 100 unique test suites by randomly selecting one test case t at a time from the pool and adding t to the test suite if that increases the accumulated coverage of the requirements by that test suite. Stop when the pool contains no more test cases with additional coverage. After each test suite is completed, place back in the pool the selected test cases.
3) Compute the average *difference-detection* (i.e., number of test cases that reveal an output difference), average size, and average ds-ratio of the 100 test suites.

For RANDOM, instead of performing this process, we analytically computed its ds-ratio for each change by determining the number of test-cases in the pool that produce a different output between P and P' and dividing it by the number of test cases in that same pool that cover the change. We then used this ratio to assess each test suite T created for the other strategies by computing the product of this ratio and the size of T, and then comparing this product with the difference-detection achieved by T.

In this study, to support our analysis of factors affecting the effectiveness of our strategies, we defined the *detectability* of a change C as the ds-ratio for RANDOM on that change, which represents the empirically-estimated probability that a test case that covers the change also reveals a difference in the output. The lower the detectability of a change, the less likely it is that any test case will reveal a difference for that change and thus, the more difficult it is to test that change.

C. Results and Analysis

For CHAIN$_d$ and PROP$_d$, we experimented with all distances (values of d) that our implementation was capable of analyzing for each change, which in some cases reached dozens of dependencies from the change. However, to compute overall results per distance, we used as the maximum value for d the shortest distance reached for all studied changes, which in our experiments was 10. This is a considerable improvement over the original formulation of propagation-based strategies, which could reach distances of about four dependencies only in similar subjects.

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

1) RQ1: Cost-effectiveness of strategies: Table II presents the average results for all strategies. In this table, the header row shows the strategy, including all distances for PROP$_d$ and distances 1–3 and 7–10 only for CHAIN$_d$ for space reasons. The rest of the table shows three results for each strategy—one result per row—which are the average number of differences for all tests suites created for all changes using that strategy, the average size of the test suites for all changes in that strategy, and the average of the ds-ratios (differences-to-size ratios) for the test suites for all changes in that strategy. For example, for strategy DU, for all changes and all 100 test suites created for each change using this strategy, the average number of differences in the output detected was 6.81, the average size of the test suites was 39.71, and the average of the ds-ratios for those test suites was 0.26. Note that 0.26 is not the result of dividing the average number of differences by the average size of the test suites—the average of the ratios is not the same as the ratio of the averages.

The results in Table II show that, for existing coverage-based strategies, every change required only one test case to cover it using RANDOM, which was expected due to the small size of the studied changes (a few lines of code, at most). The detectability of the changes (ds-ratios for RANDOM) ranged from 0.001 to 1.0, with an average of 0.24—for any set of 100 test cases, on average for all changes, 24 test cases detect an output difference.

For BR and DU, somewhat surprisingly, the average ds-ratio was indistinguishable from that for RANDOM. Despite results in the literature for the Siemens subjects showing that the whole-program counterparts of BR and, especially, DU, perform better than random testing [8], these strategies were not useful for the parts of the program affected by each change. This result can be explained by the low coverage levels attained for BR and DU in these subjects, which ranged approximately between 50–90%, and which were lower than for whole-program testing for which 100% coverage was often achieved in the same subjects. It is worth noting, however, that for du-pair coverage in the Siemens subjects, the effectiveness over random testing was unequivocably higher only for coverage levels of 93–100% [8]. Another interesting result is the much greater average number of differences detected and test-suite sizes for DU than for BR, which indicates that DU is more expensive

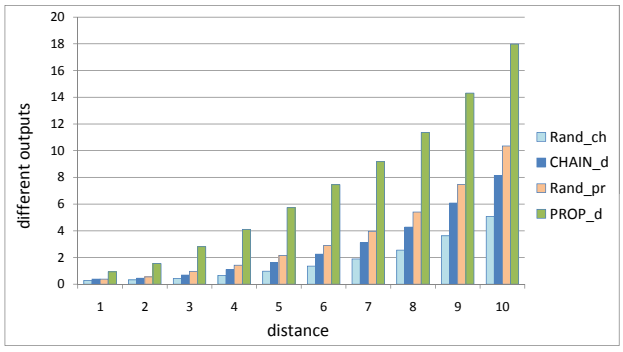

Figure 2. Effectiveness of strategies per propagation distance.

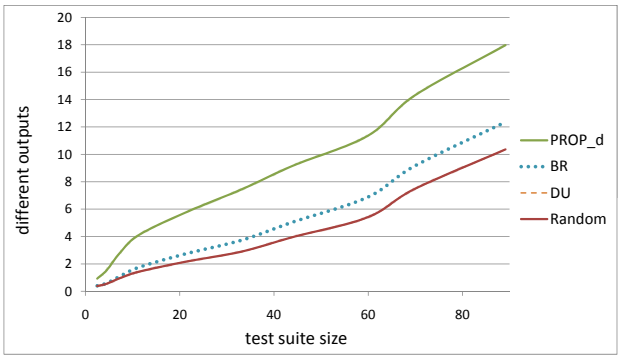

Figure 3. Effectiveness of strategies per test-suite size.

to satisfy but it also provides a greater effectiveness, at about the same ds-ratio as BR.

For our propagation-based strategies CHAIN$_d$ and PROP$_d$, at all distances, the ds-ratios were much better, especially for PROP$_d$. CHAIN$_d$ consistently achieved a ds-ratio of at least .29, although a paired t-test did not show sufficient statistical significance—the p-values with respect to RANDOM, BR, and DU were .065 (i.e., 6.5% probability that CHAIN$_d$ is not really better than RANDOM), .022, and .188, respectively. PROP$_d$, however, performed best of all strategies for all distances with a ds-ratio between .41–.49, whose superiority over RANDOM, BR, DU, and CHAIN$_d$ was statistically significant with a p-value of less than .000002 in all cases. Also, the results show that, on average, CHAIN$_d$ and PROP$_d$ detected more differences than DU at distances 10 and 7–10, respectively. These results for detected differences highlight the importance of our new approach for reaching longer distances in propagation-based strategies that can achieve an effectiveness that surpasses that of existing coverage-based approaches.

An apparent inconsistency can be observed in the average differences and test-suite sizes for CHAIN$_d$ with respect to DU. For distances 8 and 9, although not statistically significant, the average number of differences detected by CHAIN$_d$ was less than for DU whereas the average test-suite size was greater, despite the greater ds-ratio—the test-suite quality—for CHAIN$_d$. This phenomenon is explained by the distortion introduced by several changes for which CHAIN$_d$ required large test suites but only found a small number of differences. In contrast, for those same changes, both DU and BR were usually more cost-effective. Thus, such cases magnified the average number of test cases needed by CHAIN$_d$ for all changes, which explains the apparent inconsistency between these statistics and the greater average ds-ratio observed for CHAIN$_d$ over DU and BR.

Figures 2 and 3 provide a view, on average for all changes, of the effectiveness of our propagation-based strategies per distance and per test-suite size, respectively, with respect to coverage-based strategies. (In Figure 3, CHAIN$_d$ is omitted for reasons explained below.) The bar graph in Figure 2 shows, for each distance (horizontal axis), the average number of differences detected (vertical axis) by test suites created using CHAIN$_d$ and PROP$_d$. In this graph, categories *Rand_ch* and *Rand_pr* show the average effectiveness obtained by using RANDOM for test suites of identical size to those obtained by CHAIN$_d$ and PROP$_d$, respectively. This figure illustrates that, for all distances, and especially for the greatest distances, both propagation-based strategies are quite useful (i.e., considerably better than RANDOM).

The chart in Figure 3 shows the projected curves, for all test-suite sizes observed (horizontal axis), of the effectiveness (vertical axis) for PROP$_d$ and all three coverage-based strategies, where BR and DU were applied repeatedly in a fashion similar to RANDOM to match each observed test-suite size. We omitted the curve for CHAIN$_d$ from this chart because it overlaps with the coverage-based strategies, making it difficult to visualize. In other words, despite exhibiting a greater probability per test case of revealing an output difference (ds-ratio), CHAIN$_d$ did not have a better average difference-detection effectiveness per test-suite size than the coverage-based strategies, the reason for which is, again, that data points with great difference-detection values dominated the average effectiveness. Also, for similar reasons, BR was slightly better than RANDOM and DU in this comparison. PROP$_d$, in contrast, found a statistically-significantly (for p-values of .001 or less) greater number of differences than the other strategies for all test-suite sizes.

In all, for RQ1, we can conclude that, for these changes and subjects:

- PROP$_d$ produces, overall, significantly higher-quality test suites than the other strategies. In other words, PROP$_d$ discriminates and distributes test cases over the behavior space of changes better than the alternatives.
- CHAIN$_d$ provides, overall, a less effective but potentially simpler alternative of greater quality than coverage-based strategies (although not statistically significantly). When large test suites are required, however, CHAIN$_d$ may not perform better than DU or BR.
- The longer distances achieved for propagation-based strategies using our new approach allow this kind of strategies to be more effective, in number of differences detected, than DU or BR.

Table III
CLASSIFICATION OF CHANGES BY DETECTABILITY LIMIT.

Det. limit	1.0	.8	.6	.4	.2	.05	.03	.01	.005
Changes	40	33	32	30	27	18	16	7	6

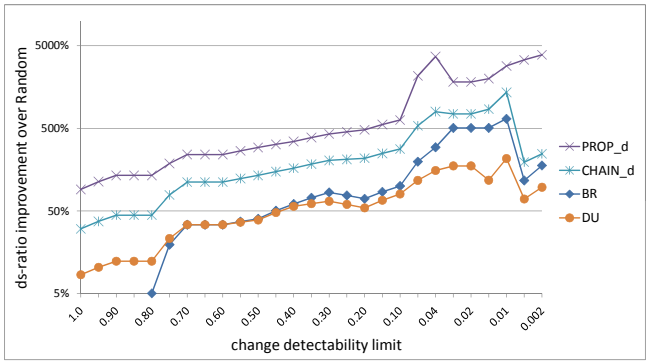

Figure 4. Cost-effectiveness improvement over RANDOM, in a logarithmic scale, of all other strategies per detectability limit.

- DU and BR are virtually indistinguishable from RANDOM, for the coverage levels achieved.

2) RQ2: Factors that affect effectiveness: For RQ1, we discussed the average results of the strategies for all changes, without distinguishing the difficulty of detecting differences (i.e., the detectability) for each change. For RQ2, we investigated how detectability influences the cost-effectiveness of all strategies. Table III shows the number of changes for different *detectability limits* (i.e., the maximum detectability of all changes in a set) in decreasing order of limits. For example, all 40 changes have a detectability limit 1.0 (i.e., detectability of at most 1.0), whereas 30 changes have a limit of 0.4 (i.e., detectability of at most 0.4).

Figure 4 illustrates how much more cost-effective PROP$_d$, CHAIN$_d$, BR, and DU are over RANDOM when we remove from the set of studied changes those whose detectability is above some limit. In this graph, the horizontal axis represents the detectability limit in decreasing order and the vertical axis is the improvement in ds-ratio (cost-effectiveness) over RANDOM, in a logarithmic scale. For example, for detectability limit 1.0 (which includes all changes), PROP$_d$ is, on average over RANDOM, 91% more cost-effective, CHAIN$_d$ is 30% more cost-effective, DU is 8% more cost-effective, and BR is 2% less cost-effective (points below 1% are not shown). However, as the detectability limit decreases, the cost-effectiveness superiority over RANDOM of the other strategies grows considerably. Considering that the graph is in a logarithmic scale, the improvement of CHAIN$_d$ and, especially, PROP$_d$ over coverage-based strategies is dramatic. At detectability limit 0.15, PROP$_d$ is already more than 5 times more cost-effective than using RANDOM, and for limits of 0.05 or less, PROP$_d$ is 18–28 times more cost-effective. The other three strategies exhibit a lower but still considerable improvement over random testing. CHAIN$_d$ is twice as effective as RANDOM at detectability 0.7 or less, whereas BR and DU achieve the same improvement but only for detectability 0.1 or less. Note that, the lower the detectability limit is, the fewer changes are considered and thus the lesser is the confidence for the reported data points.

In all, for RQ2 on these changes and subjects, we can conclude that:

- PROP$_d$ improves cost-effectiveness significantly over the other strategies for all changes and this improvement grows dramatically as detectability decreases, demonstrating that PROP$_d$ is clearly the best strategy for testing changes of all the strategies presented in this paper and in the literature.
- Some changes are "easy" to detect, but about half the changes are quite difficult to detect (detectability 0.05 or less). Hence, RANDOM is insufficient for testing changes. Moreover, the easy-to-detect changes are likely to be detected by an existing regression test suite that already covers them, so the real need for testers is a good strategy (e.g., PROP$_d$) for hard-to-detect changes.

3) RQ3: Cost of propagation-based strategies: Computing propagation constraints in previous work often took one hour for small distances [17]. Our new approach, in contrast, for any distance, usually took a few minutes, whereas the runtime overhead for the test cases was about 600%. This overhead was not as high as we expected, considering than the overhead of monitoring just du-pairs can be 100% or more [16]. Moreover, we expect that our toolset can be optimized considerably to reduce this overhead.

D. Threats to Validity

The main internal threat to the validity of our results is the potential presence of errors in our toolset that can distort the coverage measured for each test. To minimize this threat, we inserted sanity checks at different points in the toolset and manually inspected the results for several changes.

The main external threat is the representativity of our subjects and changes. More changes of different types and size must be studied and greater coverage levels for all strategies should be reached to further generalize our conclusions.

VI. RELATED WORK

Two coverage-based techniques that identify test requirements for modified software [3] [14] have been presented. These techniques define coverage criteria for entities potentially affected by changes. Our analysis in this paper, however, indicates that they can be inadequate for this task. Also, these techniques have been only partially evaluated [15]. In this paper, we presented the first complete study of these techniques, showing that they are not the most effective.

Propagation-based techniques for augmenting test suites have also been presented [2] [15]. These techniques use dependence analysis and symbolic execution to identify all possible paths along which the effects of changes can

propagate and the conditions for that propagation—up to a distance limit. Although the cost-effectiveness of this analysis has been further improved [17], we showed in this paper a new and much more efficient approach to implement these techniques that reaches longer distances for a greater effectiveness.

Other techniques have been proposed for test-suite augmentation that attempt to generate a test case that exposes a difference in the output [12] or improves program coverage after the program changes [21]. These test-generation techniques are mostly complementary to ours—they can potentially be used to create test cases that satisfy the requirements that our approach identifies. Our approach, however, distinguishes and distributes test cases that achieve different kinds of propagations of the effects of changes, producing more effective test suites than random or coverage-based test suites, as shown in our study.

At a fundamental level, our approach is related to works by Richardson and Thompson [13], Morell [10], and Voas [19] that describe the conditions under which faults create erroneous states, propagate, and manifest in the output. Our work applies these models to changes instead of faults. In particular, we used the PIE model [19] in this paper to analyze and compare change-testing strategies.

VII. Conclusion and Future Work

In this paper, we analyzed the existing change-testing techniques for test suite augmentation, comparing coverage-based with propagation-based strategies, and presented a new and more efficient approach for applying propagation-based strategies more aggressively than previous formulations, reaching longer propagation distances as a result.

We also presented in this paper, using our new approach, the first complete study of both coverage-based and propagation-based strategies for testing changes. The results of this study suggest that coverage-based strategies are not significantly better than randomly testing changes, whereas propagation-based strategies, boosted by our new approach that reaches long distances, are considerably better than existing techniques.

We are currently extending our toolset and our studies of change-testing techniques for more and larger changes to generalize and better characterize the results observed in this paper. We also plan to incorporate techniques such as abstraction (e.g., [17]) to further improve the ability of requirements to identify a larger and more diverse set of behavioral differences caused by changes.

References

[1] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Proc. of Tools and Alg. for the Constr. and Analysis of Syst.*, Mar. 2008, pp. 367–381.

[2] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, "MATRIX: Maintenance-oriented testing requirement identifier and examiner," in *Proc. of TAIC PART*, Aug. 2006, pp. 137–146.

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

[4] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Emp. Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[5] J. Ferrante, K. Ottenstein, and J. Warren, "The program dependence graph and its use in optimization," *ACM Trans. on Prog. Lang. and Systems*, vol. 9, no. 3, pp. 319–349, 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] 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.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of Int'l Conf. on Softw. Eng. (ICSE 94)*, 1994, pp. 191–200.

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

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

[11] S. Person, M. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proc. of Symp. on Foundations of Softw. Eng.*, Nov. 2008, pp. 226–237.

[12] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, Sep. 2008.

[13] 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.*, Jul. 1988, pp. 223–230.

[14] 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*, Aug. 1994, pp. 169–184.

[15] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, Sep. 2008, pp. 218–227.

[16] R. Santelices and M. J. Harrold, "Efficiently monitoring data-flow test coverage," in *Proc. of Int'l Conf. on Automated Softw. Eng.*, Nov. 2007, pp. 343–352.

[17] ——, "Exploiting program dependencies for scalable multiple-path symbolic execution," in *Proc. of Int'l Symp. on Softw. Testing and Analysis*, Jul. 2010, pp. 195–206.

[18] "Soot: a Java Optimization Framework," http://www.sable.mcgill.ca/soot/, Sable Project, McGill University.

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

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

[21] Z. Xu, Y. Kim, M. Kim, M. Cohen, and G. Rothermel, "Directed test suite augmentation: Techniques and tradeoffs," in *FSE*, Nov. 2010.