Recomputing Coverage Information to Assist Regression Testing

Pavan Kumar Chittimalli and Mary Jean Harrold, *Senior Member*, *IEEE*

Abstract—This paper presents a technique that leverages an existing regression test selection algorithm to compute accurate, updated coverage data on a version of the software, P_{i+1}, without rerunning any test cases that do not execute the changes from the previous version of the software, P_i to P_{i+1}. The technique also reduces the cost of running those test cases that are selected by the regression test selection algorithm by performing a selective instrumentation that reduces the number of probes required to monitor the coverage data. Users of our technique can avoid the expense of rerunning the entire test suite on P_{i+1} or the inaccuracy produced by previous approaches that estimate coverage data for P_{i+1} or that reuse outdated coverage data from P_i. This paper also presents a tool, RECOVER, that implements our technique, along with a set of empirical studies on a set of subjects that includes several industrial programs, versions, and test cases. The studies show the inaccuracies that can exist when an application—regression test selection—uses estimated or outdated coverage data. The studies also show that the overhead incurred by selective instrumentation used in our technique is negligible and overall our technique provides savings over earlier techniques.

Index Terms—Regression testing, regression test selection, testing, maintenance.

1 INTRODUCTION

SOFTWARE systems continually evolve during development and maintenance. The software is changed for a variety of reasons, such as correcting errors, adding new features, and improving performance. After the software is changed, *regression testing* is applied to the modified version of the software to ensure that it behaves as intended, and that modifications have not adversely impacted its quality. Regression testing is expensive. Reports estimate that regression testing consumes as much as 80 percent of the testing budget [1][1] and can consume up to 50 percent of the cost of software maintenance [2], [3].

One approach to regression testing saves the test suite T_i used to test one version of the program P_i and uses it to test the next (modified) version of the program P_{i+1}. As it is sometimes too expensive or time-consuming to rerun all of T_i on P_{i+1}, researchers have developed techniques to improve the efficiency of the retesting. For example, regression test selection (RTS) techniques select a subset of T_i, T_i' and use it to test P_{i+1} (e.g., [4], [5], [6], [7], [8], [9]). If the RTS technique is *safe*, then the test cases that it omits (i.e., $T_i - T_i'$) will give the same results on P_i and P_{i+1}, and thus, do not need to be rerun on P_{i+1} [7], [10]. Studies have shown that RTS can be effective in reducing the cost of regression testing (e.g., [5], [6], [7], [9]).

1. Los Altos Workshop on Testing (LAWST) held on 1-2 February 1997 and reported in [1].

- *P.K. Chittimalli is with Tata Research Development & Design Centre, 54, Hadapsar Industrial Estate, Hadapsar, Pune, Maharashtra, India 411 013.*
- *M.J. Harrold is with the College of Computing, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: harrold@cc.gatech.edu.*

Manuscript received 11 Feb. 2008; revised 24 June 2008; accepted 4 Dec. 2008; published online 15 Jan. 2009.
Recommended for acceptance by G. Canfora, L. Tahvildari, and H.A. Müller. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-02-0067. Digital Object Identifier no. 10.1109/TSE.2009.4.

Many of these regression testing techniques use coverage data collected when testing P_i using T_i to assist in identifying the testing that should be performed on P_{i+1}. For example, several RTS techniques collect coverage data, such as which statements [9], branches [6], [7], [8], or methods [5] are covered when P_i is executed with T_i, to use in selecting test cases from T_i to include in T_i' for testing P_{i+1}. As subsequent versions of P_i are created, coverage data for use on these subsequent versions are needed for regression testing tasks. In presentations of these regression testing techniques, especially to practitioners, there are usually questions about how the coverage data will be obtained for these subsequent versions, when only a subset of T_i is used to test P_{i+1}. The coverage data on P_i for those test cases in T_i that are not run on P_{i+1} (i.e., $T_i - T_i'$) cannot simply be copied for P_{i+1} unless the development environment maintains a mapping between entities (such as statements, branches, and methods) in P_i and entities in P_{i+1}. Because this mapping is not typically maintained, another approach for obtaining the coverage data for test cases in $T_i - T_i'$ is needed.

One approach is to reuse the coverage data collected when T_i is run on P_i for tasks on P_{i+1} and subsequent versions so that the expense of recomputing it for each subsequent version of P_i is avoided; we call this *outdated* coverage data. Elbaum et al. [11] have shown that even relatively small changes to the software can have a significant impact on code-coverage data. Thus, regression testing tasks, based on P_i, may be inaccurate for P_{i+1} and subsequent versions. For example, for RTS used with inaccurate coverage data, the selected test suite may contain test cases that do not need to be rerun and, more importantly, may omit test cases that traverse the changes, and thus, *do* need to be rerun. Our empirical studies, reported in Section 4, show many instances, where using

outdated coverage data cause unnecessary test cases to be selected and important test cases to be omitted.

Instead of reusing the coverage data computed for T_i for testing activities on P_{i+1}, two approaches have been used. The first approach reruns all test cases in T_i on P_{i+1} to get accurate coverage data on P_{i+1}; we call this *updated* coverage data. However, this approach defeats the purpose of techniques that aim to reduce the number of test cases that need to be rerun because it reruns all test cases in T_i on P_{i+1}. A variation of this approach would rerun all test cases in T_i on P_{i+1} after the critical period of regression testing to get accurate coverage data on P_{i+1} for use on the next version of the software. This retesting could be done between releases. However, this approach cannot be used when the time between new program versions is short, such as in test-driven development or in systems that are built and tested often. The second approach estimates coverage data for P_{i+1} based on coverage data for P_i; we call this *estimated* coverage data. Several researchers have reported results for such estimation (e.g., [12], [13]). However, the accuracy of the estimated coverage data is highly dependent on the locations and types of changes, and thus, activities based on it may admit useless test cases or omit important test cases. Like the results using outdated coverage data, our empirical studies for estimated coverage data, reported in Section 4, show that there are often unnecessary test cases selected and important test cases omitted.

To address the problem of maintaining updated coverage data, without incurring the expense of rerunning the entire test suite or the inaccuracy of using outdated or estimated coverage data, we have developed, and present in this paper, a technique that leverages an existing RTS technique [6] to compute accurate, updated coverage data without rerunning any test cases that do not execute the change. Our technique involves several steps. First, the technique creates and initializes a matrix, called the *coverage matrix*, in which it stores the coverage data. Second, the technique augments the RTS algorithm so that as it identifies the changed entities between P_i and P_{i+1} and selects T_i' to run on P_{i+1}, it also computes a set of mappings between corresponding statements in P_i and P_{i+1} that are not dominated[2] by changes. Third, the technique selectively instruments P_{i+1} by inserting *probes* that will report coverage data about affected parts of P_{i+1} as P_{i+1} executes; this selective instrumentation results in fewer probes than traditional instrumentation. Fourth, the technique runs the selectively instrumented version of P_{i+1} with T_i' to get updated coverage data for affected parts of P_{i+1}. Finally, the technique uses the mapping computed in the second step to transfer the coverage data for test cases not selected for rerunning (i.e., those in $T_i - T_i'$),[3] and uses the mapping to compute updated coverage data for unaffected entities for test cases in T'.

The main benefit of our technique is that it is the first to provide a method for getting updated coverage data *without* rerunning the entire test suite, while providing the same coverage data that would be obtained by rerunning the entire suite (i.e., it is accurate). Thus, users of regression testing techniques like RTS can avoid the expense of rerunning the entire suite or the inaccuracy produced by approaches that use outdated or estimated coverage data. Another benefit of our technique is that, because it performs most of its actions while RTS is being performed, the additional overhead is negligible, and, when selective instrumentation is used, overall, our technique provides savings to regression testing. Thus, the technique can be used in practice.

This paper also presents a tool, RECOVER (Recomputing Coverage Data), that implements our technique, along with a set of empirical studies conducted on a set of Java programs ranging from 1 to 105 KLOC. These studies show the inaccuracies that can exist in results of an application—RTS—when the outdated or estimated coverage data are used. For the six subjects we used, RTS applied to outdated coverage data resulted in, on average, 62.61, 83.26, 85.12, 83.38, 79.41, and 99.01 percent false positives,[4] respectively, and 14.61 percent false negatives[5] over RTS used with updated coverage data. For the six subject we used, RTS applied to estimated coverage data resulted in, on average, 0.68, 54.52, 70.12, 75.30, 68.96, and 90.56 percent false positives, respectively, and 9.28 percent false negatives over RTS used with updated coverage data. The studies also show the efficiency of our technique. For the six subjects we used, when RTS is augmented with our technique to compute the mappings and selectively instrumented programs are run with the test cases selected, the overall regression testing time is reduced, on average, 11.53 percent over RTS and 65.74 percent over retest-all.

The main contributions of this paper are:

- a description of a novel technique that computes accurate, updated coverage data when a program is modified, without rerunning unnecessary test cases,
- a discussion of a tool, RECOVER, that implements the technique and integrates it with RTS, and
- a set of empirical studies that show, for the subjects we studied, that our technique provides an effective and efficient way to update coverage data for use on subsequent regression-testing tasks.

In the next section, we provide required background, along with a motivating example that is used throughout the rest of the paper to illustrate our technique. Then, in Section 3, we present our algorithm for updating coverage information. In Section 4, we present a set of studies that demonstrates the effectiveness and efficiency of our technique. We discuss related work in Section 5 and compare our work to it. Finally, in Section 6, we summarize, conclude, and discuss future work.

2. Statement A *dominates* statement B if every path from the beginning of the program to B goes through A.

3. If the RTS technique is *safe*, test cases in $T_i - T_i'$ do not execute the changes and will behave the same as they did in P_i.

4. *False positives* represent test cases that do not need to be rerun but were selected; these test cases would not have been selected using accurate, updated coverage data.

5. *False negatives* represent test cases that need to be rerun because they go through changes and, thus, should have been selected, but were not selected; these test cases would have been selected using accurate, updated coverage data.

```
  public class Grade {
    public int calcGrade(int finalsScore, int midTermScore) {
s1    int grade = 0;
s2    if ( finalScore > 70 ) {
s3      if ( midTermScore > 80 ) {
s4        grade = 4;
        } else {
s5        grade = 3;
        }
s6    } else if ( finalScore < 50 ) {
s7      grade = 2;
      } else {
s8      grade = 3;
      }
s9    System.out.println("Grade = " + grade);
s10   return grade;
    }
    ...
  }
```

	Original Coverage			
	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	1	0
$s8$	0	0	0	1
$s9$	1	1	1	1
$s10$	1	1	1	1

Fig. 1. Version v_0 of program *Grade* and its statement-coverage matrix.

2 Background and Motivating Example

In this section, we present an example that illustrates the problem we are solving, and that we use throughout the rest of the paper to illustrate our technique. In the example, we use outdated coverage data for regression test selection for subsequent versions of a base program and show how the regression test selection results are affected by the outdated data. Although we do not show it in the example, our empirical studies (Section 4) show that using estimated coverage data for regression test selection also has a significant affect on the results. We also provide some background that is required for our algorithm and illustrate it with the example.

To illustrate the impact that changes can have on the coverage information, consider the example in Figs. 1, 2, and 3, which shows version v_0 and subsequent versions v_1 and v_2, respectively, of a program consisting of class *Grade* and method *calcGrade*. Version v_1 shows changes $c1$ and $c2$ from v_0 and version v_2 shows change $c3$ from v_1. The test suite T for *calcGrade* is shown in Table 1. Figs. 1, 2, and 3 also show the corresponding coverage matrices based on statements (i.e., *statement coverage matrices*) for the versions. In the matrices, for a particular test case, $t_i, 1 \leq i \leq 4$, "1" indicates that a statement *was covered* during execution of t_i and "0" indicates that a statement *was not covered* during execution of t_i.

For version v_0 (shown in Fig. 1), the matrix shows the original coverage data because T is run with the base version of the program (i.e., v_0); note that version v_0 has 100 percent statement coverage with respect to T. For versions v_1 and v_2 (Figs. 2 and 3, respectively), the matrix on the left shows the outdated coverage data when the coverage data for v_0 are used for the subsequent versions, and the matrix on the right shows the updated coverage data when T is run on each version. A "?" in the coverage matrices for v_1 and v_2 denotes unknown coverage for the entity in the coverage matrix because there is no corresponding statement in v_0.

The example shows that the coverage data degrade when they are applied, without recomputation, to subsequent versions of the program. For example, the outdated coverage data for v_2 show that *all* four test cases execute $s9$ but the updated (accurate) coverage data show that *none* of the test cases actually executes $s9$.

To illustrate the impact that outdated coverage data can have on a regression testing activity that uses them, we consider RTS, which was briefly described in Section 1. We use an RTS technique implemented as DEJAVOO [6]. DEJAVOO creates control-flow graphs for the original (P_{orig}) and modified (P_{mod}) versions of a program.[6] The technique traverses these graphs synchronously over like-labeled edges. *Like-labeled* edges in P_{orig} and P_{mod} are such that both edges have no label, a *true* label, a *false* label, or a matching label in a switch (or case) statement. The technique performs the traversal in a depth-first order to identify *dangerous edges*—edges whose sinks differ and for which test cases in T that executed the edge in P_{orig} should be rerun on P_{mod} because they may behave differently in P_{mod}.

To illustrate the RTS algorithm, consider running DEJAVOO for v_0 to v_1. The control-flow graphs for v_0 and v_1 are shown in Fig. 4, with v_0 on the left and v_1 in the center. Using the control-flow graphs, DEJAVOO traverses like-labeled edges starting at the *Entry* node of each graph and finds all sinks on the path $s1, s2, s3, s4, s9, s10, Exit$ in v_0 identical to the sinks on the path $s1, s2, s3, s4, s11, s12, Exit$ in v_1. Note

6. Control-flow graphs are constructed using JABA, which was created by the Aristotle Research Group at Georgia Tech (see http://www.cc.gatech.edu/aristotle) and accounts for object-oriented features, such as polymorphism, aliasing, and use of libraries.

```
   public class Grade {
     public int calcGrade(int finalsScore, int midTermScore){
s1     int grade = 0;
s2     if ( finalScore > 70 ) {
s3       if ( midTermScore > 80 ) {
s4         grade = 4;
         } else {
s5         grade = 3;
         }

s6     } else if ( finalScore > 60 ) { // change c1
s7       grade = 3;
s8     } else if ( finalScore < 35 ) { // change c2
s9       grade = 1;
       } else {
s10      grade = 2;
       }

s11    System.out.println("Grade = " + grade);
s12    return grade;
     }
     ...
   }
```

	Outdated Coverage			
	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	1	0
$s8$	0	0	0	1
$s9$	1	1	1	1
$s10$	1	1	1	1
$s11$	?	?	?	?
$s12$	?	?	?	?

	Updated Coverage			
	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	0	0
$s8$	0	0	1	1
$s9$	0	0	1	0
$s10$	0	0	0	1
$s11$	1	1	1	1
$s12$	1	1	1	1

Fig. 2. Version v_1 of program *Grade* and its statement-coverage matrices.

that, although they have different node numbers, $s9$ and $s11$ are identical, as are $s10$ and $s12$. When the traversal continues at $s3$ in the two graphs, it finds that $s5$ in v_0 and $s5$ in v_1 match. However, when the traversal reaches edges $e4$ in the two graphs (with source $s2$ and sink $s6$), it finds that the sinks, $s6$ in both graphs, differ (an inspection of the code shows that the statements corresponding to $s6$ in v_0 and v_1 differ). The algorithm then marks $e4$ in v_0 as a dangerous edge. Using the updated matrix for v_0, we see that test cases $t3$ and $t4$ traverse $e4$ in v_0, and thus, they are selected to run on v_1.

Now consider running DEJAVOO on v_1 and v_2. Using the control-flow graphs for v_1 and v_2 in Fig. 4, DEJAVOO traverses like-labeled edges and finds that all corresponding nodes match until edge $e10$ is reached. Because the sinks of $e10$ differ in the two programs, $e10$ is marked as a dangerous edge, and test cases are selected for it. Using the outdated matrix for v_1 (Fig. 2), DEJAVOO selects the set that executed $s8$—$\{t4\}$. However, using the updated matrix for v_1, DEJAVOO selects $\{t3, t4\}$. This example does not illustrate that the use of outdated coverage data can include test cases that do not need to be rerun—that is, the use of outdated coverage data can cause *imprecision* in analyses that use the data. (Our empirical results in Section 4 show that imprecision occurs frequently.) However, it does illustrate an even bigger problem—the use of outdated coverage data causes the RTS algorithm to omit an important test case $t3$ that executes the change and should be rerun. In this case, the use of outdated coverage data causes *unsafety* in the analysis that used the data. Our empirical studies in Section 4 show the extent to which outdated coverage data can affect regression test selection.

3 ALGORITHM

Our algorithm, RECOMPUTEMATRIX, shown in Fig. 5, for recomputing coverage data after changes are made to a program provides the same coverage data as rerunning all test cases in the original test suite but requires running only those test cases selected to run on the modified program. Studies show that running only the test cases in the original test suite selected by RTS can provide significant savings in regression testing time [6]. Thus, our algorithm can be an important part of an efficient regression testing process.

RECOMPUTEMATRIX takes four inputs: P_{orig} and P_{mod}, the original and modified versions of a program, respectively; T, a set of test cases that was run on P_{orig}; and m_{orig}, the coverage matrix for P_{orig} when it was run with T, represented as a matrix $[|E| \times |T|]$, where E is the set of entities in P_{orig}. RECOMPUTEMATRIX outputs m_{mod}, an accurate, updated coverage matrix for P_{mod}. The algorithm uses procedure SELECTIVEINSTRUMENT(), shown in Fig. 6, to perform selective instrumentation on P_{mod} by placing probes for monitoring coverage only at parts of P_{mod} affected by the changes.

RECOMPUTEMATRIX consists of five main steps:

1. creating and initializing the coverage matrix m_{mod} for P_{mod} (lines 1 and 2);

```
  public class Grade {
    public int calcGrade(int finalsScore, int midTermScore) {
s1    int grade = 0;
s2    if ( finalScore > 70 ) {
s3      if ( midTermScore > 80 ) {
s4        grade = 4;
        } else {
s5        grade = 3;
        }
s6    } else if ( finalScore > 60 ) {
s7      grade = 3;

s8    } else if ( (finalScore+midTermScore)/2 < 35 ) { // change c3

s9      grade = 1;
      } else {
s10     grade = 2;
      }
s11   System.out.println("Grade = " + grade);
s12   return grade;
    }
    ...
  }
```

	Outdated Coverage			
	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	1	0
$s8$	0	0	0	1
$s9$	1	1	1	1
$s10$	1	1	1	1
$s11$	?	?	?	?
$s12$	?	?	?	?

	Updated Coverage			
	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	0	0
$s8$	0	0	1	1
$s9$	0	0	0	0
$s10$	0	0	1	1
$s11$	1	1	1	1
$s12$	1	1	1	1

Fig. 3. Version v_2 of program *Grade* and its statement coverage matrices.

2. identifying T' the set of test cases in T to rerun on P_{mod} and computing the entity mappings *entityMap* between P_{orig} and P_{mod} (line 3);
3. creating the selectively instrumented version of P_{mod}, $P_{mod-inst}$, (line 4);
4. running $P_{mod-inst}$ with T' to get coverage data for the affected entities in P_{mod} (lines 5-7); and
5. transferring the coverage data for the unaffected parts of P_{mod} using the mappings stored in *entityMap* and the affected entities in *insEntities* (lines 8-18).

TABLE 1
Test Suite T for the *calcGrade*

Test cases	Inputs
$t1$	finalScore=71, midTermScore=81
$t2$	finalScore=71, midTermScore=71
$t3$	finalScore=34, midTermScore=60
$t4$	finalScore=52, midTermScore=50

RECOMPUTEMATRIX then returns m_{mod} (line 19). Fig. 7 shows some of the steps in the creation of matrix m_{v1} that is created by the algorithm for the changes from v_0 to v_1 of *Grade* (Figs. 1 and 2, respectively).

In Step 1, the algorithm creates m_{mod}, the coverage matrix for P_{mod}, with $|E'|$, the number of entities in P_{mod}, rows and $|T|$, the number of test cases in T, columns (line 1). Next, RECOMPUTEMATRIX initializes all entries in this matrix to "0" indicating that none of the entities in P_{mod} are covered (line 2). For example, when RECOMPUTEMATRIX is run on v_0 and v_1, it creates m_{v1} with 12 rows and four columns, and initializes all entries to "0." The matrix on the left in Fig. 7 shows m_{v1} after it is created and initialized.

In Step 2, RECOMPUTEMATRIX runs the modified version of DEJAVOO, MOD-DEJAVOO. Recall that DEJAVOO (described in Section 2) first traverses control-flow graphs for the original and modified versions of the program to identify dangerous edges and then uses the coverage matrix to identify test cases that need to be rerun. As MOD-DEJAVOO traverses the graphs to find dangerous edges, it stores, in C, change information from P_{orig} to P_{mod}, and it also stores, in *entityMap*, the mapping information between the entities in P_{orig} and P_{mod} that it visits. Thus, in

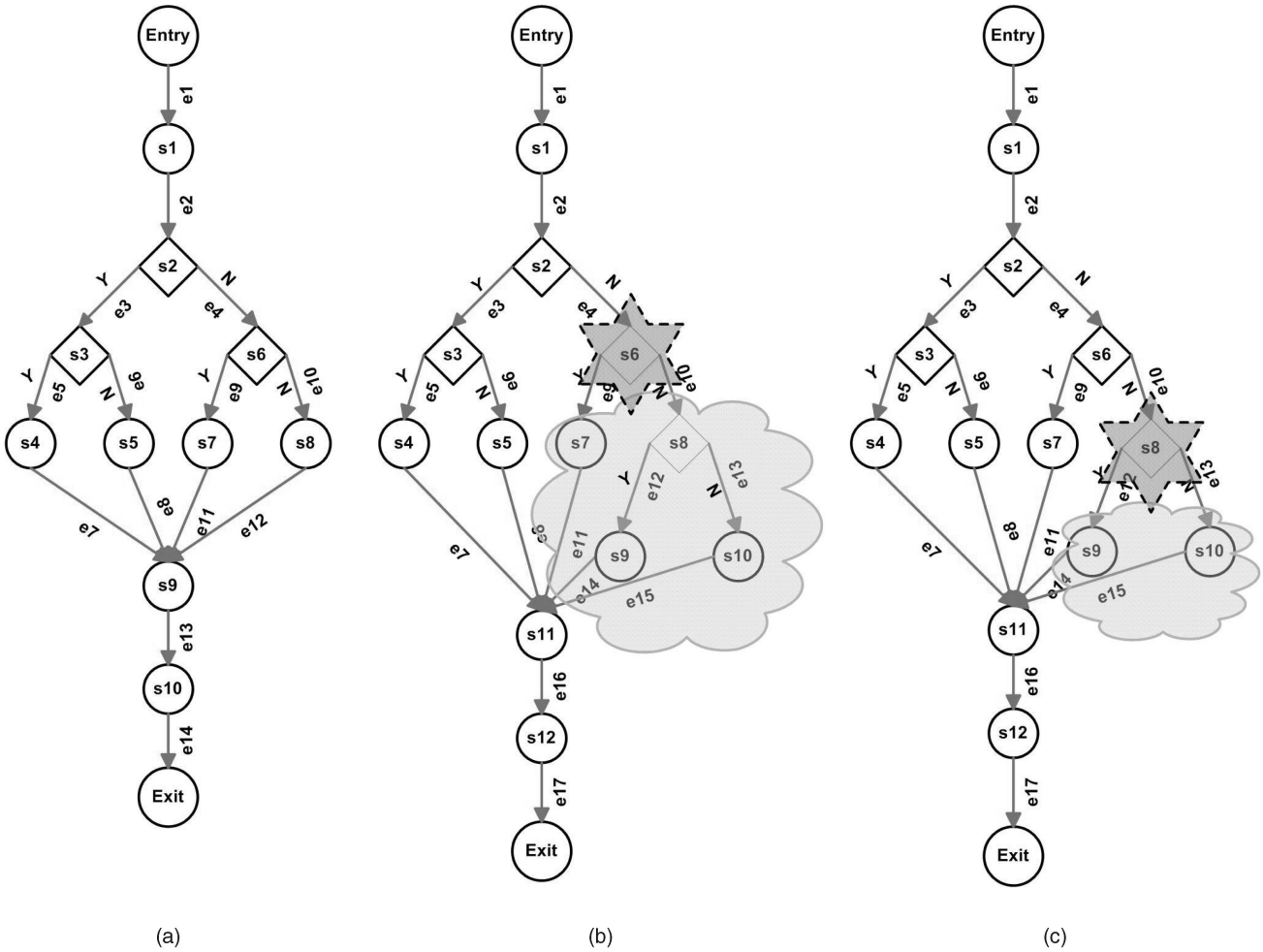

Fig. 4. Control-flow graphs for (a) v_0, (b) v_1, and (c) v_2. The starred region in v_1 depicts the change from v_0 to v_1; likewise, the starred region in v_2 depicts the change from v_1 to v_2. The clouded region in v_1 depicts the nodes that are not examined when the algorithm is run on v_0 and v_1; likewise, the clouded region in v_2 depicts the nodes that are not examined when the algorithm is run on v_1 and v_2.

addition to returning the test cases to rerun T', MOD-DEJAVOO returns C and *entityMap* (line 3).

To illustrate, consider Fig. 4, wherein the control-flow graphs of programs v_0 (Fig. 1), v_1 (Fig. 2), and v_2 (Fig. 3) are shown. When MOD-DEJAVOO traverses v_0 and v_1, it returns $\{t3, t4\}$ for T', $\{s6\}$ for C, and statement pairs $\{(s1, s1),\ (s2, s2), (s3, s3), (s4, s4)(s5, s5), (s9, s11), (s10, s12)\}$ for *entityMap*.

As MOD-DEJAVOO traverses the graphs, it partitions the nodes (and thus, the statements) into three sets: nodes that are examined and *match*, nodes that represent *changes*, and nodes that are *not examined*. A node in the changes set can dominate[2] nodes representing other changes that may not be reached during traversal of the graphs. As a result, these unreached changes may be added to the not-examined set. The graphs in Fig. 4 illustrate these sets by showing them in the modified program—the changes sets are shown in starred regions, the not-examined sets are shown in clouded regions, and the remainder of the nodes are in the match sets. For example, when MOD-DEJAVOO examines v_0 and v_1, the graph in the center of Fig. 4 representing v_1 shows that $s6$ is in the changes set, nodes $s7$-$s10$ are in the not-examined set, and the rest of the nodes are in the match set. Change $c1$ dominates change $c2$, and as a result, statement $s8$ is not added to C.

In Step 3, RECOMPUTEMATRIX uses procedure SELECTIVEINSTRUMENT() to instrument the entities in P_{mod} to get $P_{mod-inst}$ so that new coverage data can be collected for affected parts of P_{mod} (line 4). SELECTIVEINSTRUMENT(), shown in Fig. 6, inputs P_{mod}, the modified program, and $changeSet$, the change set. The procedure performs reachability analysis on the control-flow graph of P_{mod} from each change and instruments only these affected, reachable entities. SELECTIVEINSTRUMENT() returns the instrumented program $P_{mod-inst}$ and all affected entities *insEntities*. For example, consider the algorithm on v_0 of *Grade* (Fig. 1) and v_1 of *Grade* (Fig. 2). For this example, SELECTIVEINSTRUMENT() returns the instrumented version of v_1 and *insEntities* $\{s6, s7, s8, s9, s10, s11, s12\}$.

In Step 4, RECOMPUTEMATRIX executes $P_{mod-inst}$ with all $t \in T'$ (lines 5-7) to get coverage data for all affected entities (*insEntities*) with respect to T'. The algorithm stores this coverage data in m_{mod}. For example, consider again the algorithm on v_0 of *Grade* (Fig. 1) and v_1 of *Grade* (Fig. 2). The instrumented version of v_1 is executed with test cases $t3$ and $t4$ to capture coverage information for the affected statements—$\{s6, s7, s8, s9, s10, s11, s12\}$. This coverage data

ALGORITHM RECOMPUTEMATRIX()

Input: P_{orig}, P_{mod}: Original and modified versions, respectively

T : set of test-cases run on P_{orig}

m_{orig} : coverage matrix $[|E| \times |T|]$ for P_{orig} when run with T, where E is a set of entities in P_{orig}

Output: m_{mod}: coverage matrix $[|E'| \times |T|]$ for P_{mod}

Use: *SelectiveInstrument(P,C)* : selectively instruments program P with changes C for coverage

Declare: *entityMap* : $\{(e, e')$: e, e' entities in P_{orig}, P_{mod}, respectively$\}$

$P_{mod-inst}$: instrumented version of P_{mod}

T': set of test-cases

C: set of change nodes in P_{mod}

E': set of entities in P_{mod}

e, e': entities in E from P_{orig} and E' from P_{mod}, respectively

insEntities: set of instrumented entities in $Pmod$

(1) m_{mod} = create coverage matrix for P_{mod} // Step 1
(2) $m_{mod}[e', t] = 0$ for all $e' \in E'$, $t \in T$
(3) (T', C, $entityMap$) = MOD-DEJAVOO(P_{orig}, P_{mod}) // Step 2
(4) ($P_{mod-inst}$, $insEntities$) = $SelectiveInstrument(P_{mod}, C)$ // Step 3
(5) **foreach** $t \in T'$ // Step 4
(6) $run\_Program(t, P_{mod-inst})$
(7) **endfor**
(8) **foreach** $(e, e') \in entityMap$ // Step 5
(9) **foreach** $t \in T$
(10) **if** $t \in T - T'$
(11) $m_{mod}[e', t] = m_{orig}[e, t]$
(12) **else** // $t \in T'$
(13) **if** $e' \notin insEntities$
(14) $m_{mod}[e', t] = m_{orig}[e, t]$
(15) **endif**
(16) **endif**
(17) **endfor**
(18) **endfor**
(19) return m_{mod}

Fig. 5. RECOMPUTEMATRIX algorithm.

are recorded in m_{v1}, which is shown, after this step, in Fig. 7b. Note that at this stage in the algorithm, only coverage data for entities affected during execution of $t3$ and $t4$ are recorded.

In Step 5, RECOMPUTEMATRIX considers entities in *entityMap* and all test cases in T (lines 8 and 9). Because of the technique used by DEJAVOO, and thus, by MOD-DEJAVOO, for traversing the graphs, entities in the *entityMap* are on at least one path from the beginning of the program that does not go through the other changes (i.e., these entities are not dominated by changes). There are two cases to consider. First, the algorithm considers test cases in $T - T'$ (line 10). These test cases do not execute the change from P_{orig} to P_{mod}, and thus, will behave the same in both P_{orig} and P_{mod}. Consequently, the coverage data for these test cases remain unchanged for these entities. To update the coverage data for these entities, the algorithm need only use *entityMap* to transfer the coverage data for these entities from m_{orig}, the coverage matrix for P_{orig}, to m_{mod}, the coverage matrix for P_{mod} (line 11).

To illustrate, consider again Fig. 4. When MOD-DEJAVOO traverses v_0 and v_1, it returns *entityMap* $\{(s1, s1), (s2, s2), (s3, s3), (s4, s4), (s5, s5), (s9, s11), (s10, s12)\}$. For test cases $\{t1, t2\}$, which are not selected for inclusion in T' by MOD-DEJAVOO, and thus, are in $T - T'$, the coverage data from m_{v0} for $s1$-$s5, s9$, and $s10$ are transferred to $s1$-$s5, s11$, and

PROCEDURE SELECTIVEINSTRUMENT()

Input: P_{mod}: modified version of program P_{orig}

$changeSet$: set of changed entities from P_{orig} to P_{mod}

Output: $P_{mod-inst}$: instrumented version of P_{mod}

insEntities: set of instrumented entities in P_{mod}

Use: *instrument(e)* : inserts a probe to record coverage of entity e

reachablefrom(c) : returns the set of nodes that are reachable from change c

Declare: e, e': entities in P_{orig} and P_{mod}, respectively

$matchSet$: set of mapped entities in P_{mod}

E: set of entities

(1) $insEntities = \phi$
(2) **foreach** $c \in changeSet$
(3) $E =$ *reachablefrom* (c)
(4) $insEntities = insEntities \cup E$
(5) **foreach** $e \in E$
(6) *instrument* (e)
(7) **endfor**
(8) **endfor**
(9) return $P_{mod-inst}, insEntities$

Fig. 6. SELECTIVEINSTRUMENT() procedure.

$s12$, respectively, in m_{v1}. Rows $s1$-$s5$, $s11$, and $s12$ and columns $t1$ and $t2$ of m_{v1}, on the right in Fig. 7c, show the transferred coverage data.

Second, the algorithm considers test cases in T' (line 12). These test cases do execute the change from P_{orig} to P_{mod}, and thus, they will behave the same for those entities in P_{mod} that are not affected by the change. These are exactly the entities in *entityMap* that are in not instrumented (i.e., *not* in *insEntities*) (line 13). To update the coverage data for these entities, the algorithm need only use *entityMap* to transfer the coverage data for these entities from m_{orig}, the coverage matrix for P_{orig}, to m_{mod}, the coverage matrix for P_{mod} (line 14).

For example, consider again Fig. 4, and the results of running MOD-DEJAVOO on v_0 and v_1, where it returns

	$t1$	$t2$	$t3$	$t4$
$s1$	0	0	0	0
$s2$	0	0	0	0
$s3$	0	0	0	0
$s4$	0	0	0	0
$s5$	0	0	0	0
$s6$	0	0	0	0
$s7$	0	0	0	0
$s8$	0	0	0	0
$s9$	0	0	0	0
$s10$	0	0	0	0
$s11$	0	0	0	0
$s12$	0	0	0	0

(a)

	$t1$	$t2$	$t3$	$t4$
$s1$	0	0	0	0
$s2$	0	0	0	0
$s3$	0	0	0	0
$s4$	0	0	0	0
$s5$	0	0	0	0
$s6$	0	0	1	1
$s7$	0	0	0	0
$s8$	0	0	1	1
$s9$	0	0	1	0
$s10$	0	0	0	1
$s11$	0	0	1	1
$s12$	0	0	1	1

(b)

	$t1$	$t2$	$t3$	$t4$
$s1$	1	1	1	1
$s2$	1	1	1	1
$s3$	1	1	0	0
$s4$	1	0	0	0
$s5$	0	1	0	0
$s6$	0	0	1	1
$s7$	0	0	0	0
$s8$	0	0	1	1
$s9$	0	0	1	0
$s10$	0	0	0	1
$s11$	1	1	1	1
$s12$	1	1	1	1

(c)

Fig. 7. The matrices show m_{v1} at various stages when RECOMPUTEMATRIX applied to v_0 (shown in Fig. 2) and v_1 (shown in Fig. 3): matrix (a) shows m_{v1} after Step 1 of the algorithm (after initialization); matrix (b) shows m_{v1} after Step 4 of the algorithm (after execution of T'); and matrix (c) shows m_{v1} after Step 5 (the final step) of the algorithm (after updating using *entityMap*).

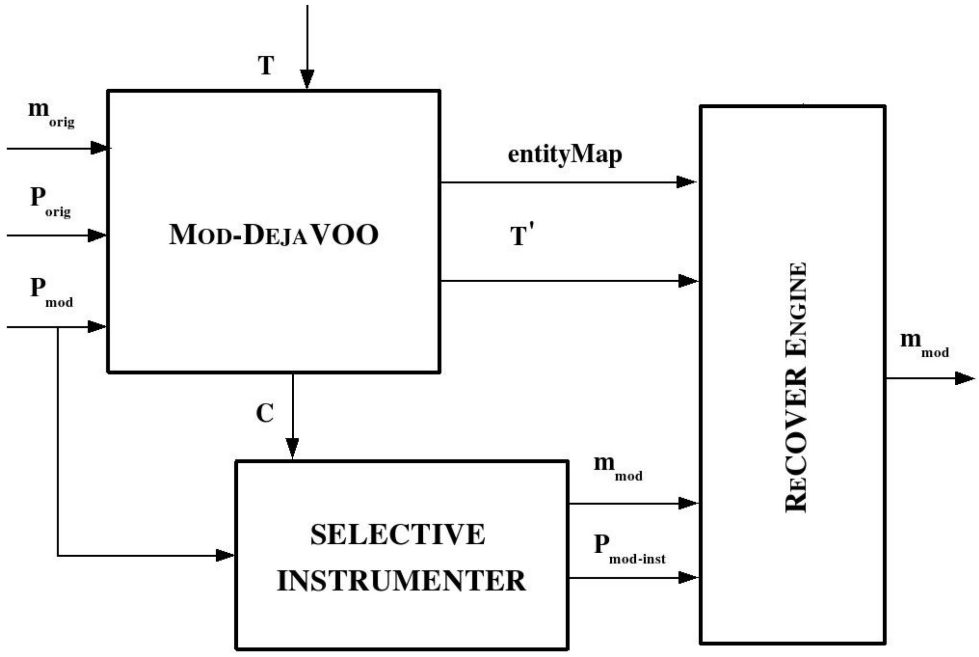

Fig. 8. Architecture of RECOVER.

entityMap $\{(s1, s1), (s2, s2), (s3, s3), (s4, s4), (s5, s5), (s9, s11), (s10, s12)\}$. Of the statements in $v_1, s1, s2, s3, s4$, and $s5$ are unaffected by the change, and thus, not in *insEntities*. For test cases $\{t3, t4\}$, which are selected as T' by MOD-DEJAVOO, the coverage data from m_{v0} for $s1$-$s5$ are transferred to m_{v1} for $s1$-$s5$, respectively. Rows $s1$-$s5$ and columns $t3$ and $t4$ of the matrix on the right in Fig. 7c show the results.

At this point in RECOMPUTEMATRIX, the coverage data are updated and the algorithm returns the updated matrix m_{mod}. The final updated coverage matrix for the change from v_0 to v_1 is shown on the right in Fig. 7.

The correctness of RECOMPUTEMATRIX depends on the safety of the RTS algorithm used. Rothermel and Harrold [7] proved that under certain conditions (described in detail in their paper), their RTS algorithm is safe. Under their definition, an RTS algorithm is *safe* if, for a program P_{orig}, a test suite T that was used to test P_{orig}, and a modified version of P_{orig}, P_{mod}, the algorithm selects every test case in T that could reveal a fault from P_{orig} to P_{mod}; they call these test cases *fault-revealing* test cases. Their proof shows that there is no effective procedure that, in general, precisely identifies the fault-revealing test cases in T. The proof then shows that, under controlled regression testing,[7] the test cases that execute changes from P_{orig} to P_{mod} (i.e., *modification-traversing* test cases) are a superset of the fault-revealing tests, and that their RTS algorithm finds all modification-traversing test cases. Thus, their RTS algorithm is safe because it selects all fault-revealing test cases from T. In later work, Harrold et al. [10] extended this RTS algorithm to create a safe RTS algorithm for Java programs and implemented it as DEJAVOO.

7. *Controlled regression testing* tests P_{mod} under conditions that are equivalent to those that were used to test P_{orig}.

Our modified version of this RTS algorithm, MOD-DEJAVOO, is also safe with respect to the modification-traversing test cases. These test cases in T' may behave differently on P_{orig} and P_{mod}, and thus, may have different coverage data from P_{orig} to P_{mod}. RECOMPUTEMATRIX populates m_{mod} for test cases in T' by rerunning the test cases and recording the coverage data in m_{mod}.

Selecting all modification-traversing test cases for inclusion in T' means that test cases in $T - T'$ are nonmodification-traversing and will behave the same way they did when run on P_{orig}. This selection also guarantees that entities in the matched set have the same coverage data on P_{orig} and P_{mod} for $T - T'$ when RECOMPUTEMATRIX uses *entityMap*. RECOMPUTEMATRIX populates m_{mod} for test cases in $T - T'$ by transferring the coverage data for these test cases from m_{orig} to m_{mod} using *entityMap*.

Thus, RECOMPUTEMATRIX provides correct updated coverage data, given the safe RTS technique that MOD-DEJAVOO uses.

4 EMPIRICAL STUDIES

To evaluate our technique, we developed a tool called RECOVER that implements our algorithm, and used it to conduct three empirical studies. In this section, we first describe RECOVER, then present our empirical setup, and finally, present the three studies.

4.1 ReCover

Fig. 8 shows the architecture of RECOVER, which consists of three components:

- MOD-DEJAVOO implements the MOD-DEJAVOO component of algorithm RECOMPUTEMATRIX. MOD-DEJAVOO inputs program P_{orig} and its coverage matrix m_{orig}, along with modified version P_{mod}

TABLE 2
Details of the Subjects Used in the Empirical Studies

Name	Number of versions	Size in KLOC	Test-suite size	Statement coverage (on average)	Branch coverage (on average)
Jakarta Regexp	4	1 - 2	148	82.13%	76.21%
NanoXML	33	3 - 4	214 - 216	88.76%	77.15%
ProAX	4	30 - 35	312	91.45%	85.43%
Assent	2	60 - 66	300-314	84.91%	77.65%
JABA	309	50 - 80	707	81.32%	69.33%
Darpan	4	100 - 105	1025-1102	90.22%	88.42%

and test suite T. MOD-DEJAVOO outputs the entity correspondence *entityMap* between P_{orig} and P_{mod}, along with T', the set of test cases to run on P_{mod}, and C, the set of changes from P_{orig} to P_{mod}.

- SELECTIVE INSTRUMENTER inputs P_{mod} and change set C. SELECTIVE INSTRUMENTER performs three activities: 1) uses an instrumentation tool, INSECT [14], to create coverage matrix m_{mod} for P_{mod}; 2) computes reachable entities from the change set C along control-flow paths in P_{mod}; and 3) uses INSECT to add probes to these reachable entities in P_{mod} to create $P_{mod-inst}$ so that, as $P_{mod-inst}$ executes, it records coverage and populates the coverage matrix.
- RECOVER ENGINE inputs *entityMap*, T', m_{mod}, and P_{mod_inst}. RECOVER ENGINE performs five activities:
  1. initializes each entry in coverage matrix m_{mod} to "0";
  2. populates the coverage information for $T - T'$, according to the algorithm, and transfers coverage data for the matched entities from m_{orig} to m_{mod} using *entityMap*;
  3. populates the coverage information for T' by running $P_{mod-inst}$ with the set of test cases in T';
  4. populates coverage information for noninstrumented entities for T' using *entityMap* and *insEntities*;
  5. outputs the updated coverage matrix m_{mod}.

4.2 Subjects

For our studies, we used the six subjects shown in Table 2: *Jakarta Regexp*, *NanoXML*, *ProAX*, *Assent*, *JABA*, and *Darpan*. Each of the subjects contained a program, a number of versions of that program, and a test suite. We used the versions and test suites as we obtained them—we did not make additional changes to the versions or add test cases to the test suites.[8]

Jakarta Regexp[9] is a Java regular-expression package that has four versions and 1,000-2,000 lines of code, depending on the version. *Jakarta Regexp* has a test suite with 148 test cases. The test suite for *Jakarta Regexp* achieves, on average, 82.13 percent statement coverage and 76.21 percent branch coverage.

NanoXML[10] is an XML processor that has six versions and 3,000-4,000 lines of code, depending on the version. Some of these versions have additional versions that can be obtained by enabling different numbers of faults: v_1 has seven versions, v_2 has seven versions, v_3 has 10 versions, and v_5 has nine versions. Using these versions, we performed our studies on 33 versions of *NanoXML*. *NanoXML* has test suites of sizes ranging from 214 to 216, also depending on the version. The test suites for *NanoXML* achieve, on average, 88.76 percent statement coverage and 77.15 percent branch coverage.

ProAX[11] is a program analysis and transformation framework [15] that inputs a file containing specifications to perform various kinds of program analyses and transformations, and outputs a Java class file. *ProAX* has four versions, with sizes ranging from 30 to 35 KLOC, depending on the version, and a test suite consisting of 312 test cases. The test suites for *ProAX* achieve, on average, 91.45 percent statement coverage and 85.43 percent branch coverage.

Assent[11] is a framework of standards checking tools that automatically ensure that programs adhere to the defined standard. *Assent* has two versions—one with 60 KLOC and the other with 66 KLOC—and test suites of sizes 300 and 314, respectively. The test suites for *Assent* achieve, on average, 84.91 percent statement coverage and 77.65 percent branch coverage.

JABA (Java Architecture for Bytecode Analysis)[12] is an extensible API that supports research in program analysis for Java programs and the development of program-analysis-based software engineering tools for Java. *JABA* has 309 versions, with the sizes of the versions ranging from 50 to 80 KLOC, depending on the version, and a test suite consisting of 707 test cases. The test suite for *JABA* achieves, on average, 81.32 percent statement coverage and 69.33 percent branch coverage.

8. Although it is not central to these studies, we note that DEJAVOO has been shown in previous studies [6] to be effective in reducing the number of test cases that need to be rerun when applied to large, industrial-strength programs. Its underlying analysis handles complex Java programs with extensive use of polymorphic calls, aliases, etc.

9. http://jakarta.apache.org/regexp/index.html.

10. *NanoXML* is available for Software-artifact Infrastructure Repository (SIR) at http://sir.unl.edu/portal/index.html.

11. *ProAX*, *Assent*, and *Darpan* are tools developed at TCS, Pune, India; see http://www.tcs-trddc.com/software-tools.htm.

12. http://www.cc.gatech.edu/aristotle/jaba.php.

TABLE 3
DEJAVOO Results on *Jakarta Regexp* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	148	0	0	148	0	0	148
(v_1, v_2)	148	133	0	17	2	0	15
(v_2, v_3)	148	145	0	3	0	0	3

Darpan is a static-analysis framework that builds an analyzer from analysis specifications. The language front end creates the intermediate representation of the program under analysis and the analyzer creates analysis results by processing the intermediate representation. *Darpan* has four versions, with sizes ranging from 100 to 105 KLOC, depending on the version, and test suites of sizes ranging from 1,025 to 1,102. The test suites for *Darpan* achieve, on average, 90.22 percent statement coverage and 88.42 percent branch coverage.

4.3 Study 1

The goal of Study 1 is to address research question RQ1.

> What are the effects of the three techniques for providing coverage data—outdated, estimated, and updated—on regression test selection (RTS)?

To answer this research question, we used all six subjects described in Section 4.2. For these subjects, we populated outdated, estimated, and updated coverage data. For outdated coverage data, we ran T on v_0 to collect m_0, the coverage data for version v_0 of program P. We then used m_0 for RTS activities on subsequent versions of v_0. For estimated coverage data, we used JDIFF [12] to estimate the coverage data and populate m_{i+1} for each version v_{i+1} using m_i, the coverage data for v_i. JDIFF compares two Java programs, v_i and v_{i+1}, and identifies both differences and correspondence between the two versions. Because JDIFF uses heuristics to determine differences and correspondences, it can result in both false positives and false negatives. The technique is based on a representation of object-oriented programs that handles object-oriented features, and thus, can capture the behavior of the program. Using the correspondence, which is a mapping between statements in the two versions, it estimates the coverage for v_{i+1} using the coverage data from v_i and uses it to populate m_{i+1}. For updated coverage data, we used our tool RECOVER to calculate m_{i+1} for version v_{i+1} using m_i for version v_i. Recall that the updated coverage data that our technique computes are identical to those computed if all test cases were rerun. As a check of our RECOVER implementation, we computed the updated coverage data by running all test cases on the versions of P and comparing these accurate coverage data with those obtained using RECOVER. In all cases, the coverage data were the same.

Tables 3, 4, 5, 6, 7, and 8 show the results of the study for the six subjects, respectively. In each table, the first column shows the versions on which DEJAVOO was run, with (v_i, v_{i+1}) representing the change from version v_i to version v_{i+1}. The next three columns show the results when DEJAVOO is run using outdated coverage data: the number of test cases selected, the number of false positives[4] in that set of test cases, and the number of false negatives[5] in that set of test cases. The next three columns show similar results when DEJAVOO is run using coverage data estimated with JDIFF. The last column shows the number of test cases selected by DEJAVOO using updated coverage data (the same coverage data as would be obtained by rerunning all test cases in the test suite). For *Jakarta Regexp*, *ProAX*, *Assent*, and *Darpan*, the tables show the results of running DEJAVOO on all pairs of versions. For *NanoXML* and *JABA*, the tables show only a representative subset of the results of running DEJAVOO on all pairs of versions.

To understand the results, consider the first row of Table 3, which shows that when DEJAVOO is run on the original version (v_0) of *Jakarta Regexp* and one subsequent version (v_1) of the same subject, it selects the same number of test cases for the outdated, estimated, and updated coverage data. These results are identical because the coverage data are accurate for v_0. However, when DEJAVOO is run with subsequent versions, the results degrade. For both (v_1, v_2) and (v_2, v_3) DEJAVOO selects all test cases (e.g., 148) using the outdated coverage data, many of which are false positives. For example, for (v_1, v_2), there are 133 false positives because only 15 test cases are selected with accurate coverage data, and for (v_2, v_3), there are 145 false positives because only three test cases are selected with accurate coverage data. The results show that for this subject, using estimated coverage data results in a set of test cases that is closer to that selected with accurate, updated coverage data. However, there are still some test cases selected that need not be rerun (i.e., there are two false positives for (v_1, v_2)).

The results degrade and vary even more with larger programs. To see this, next consider the results on the subject in Table 4. The results show many false positives. For example, for $(v_2, v_{2.1})$ in the table, only 35 test cases actually need to be rerun. However, with outdated coverage data, 198 are selected, and with estimated coverage data, 50 are selected. Of these, 188 are false positives for outdated coverage data and 17 are false positives for estimated

TABLE 4
DEJAVOO Results on *NanoXML* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	166	0	0	166	0	0	166
(v_1, $v_{1.1}$)	167	167	3	41	38	0	3
($v_{1.1}$, $v_{1.2}$)	182	180	2	40	36	0	4
($v_{1.2}$, $v_{1.3}$)	196	188	10	35	17	0	18
($v_{1.3}$, $v_{1.4}$)	201	188	0	43	30	0	13
($v_{1.4}$, $v_{1.5}$)	210	206	2	29	23	0	6
($v_{1,5}$, $v_{1.6}$)	214	209	0	25	20	0	5
($v_{1.6}$, $v_{1.7}$)	214	198	0	37	11	1	16
(v_1, v_2)	182	171	21	61	31	2	32
(v_2, $v_{2.1}$)	198	188	25	50	17	2	35
($v_{2.1}$, $v_{2.2}$)	214	180	0	68	67	3	4
($v_{2.2}$, $v_{2.3}$)	214	190	0	78	65	5	18
($v_{2.3}$, $v_{2.4}$)	214	201	0	83	74	4	13
($v_{2.4}$, $v_{2.5}$)	214	208	0	106	105	5	6
($v_{2.5}$, $v_{2.6}$)	214	209	0	115	114	4	5
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
($v_{5.7}$, $v_{5.8}$)	214	199	0	run-all	199	0	15
($v_{5.8}$, $v_{5.9}$)	214	182	0	214	182	0	32

coverage data. Furthermore, for these versions of *NanoXML*, 25 are false negatives for outdated coverage data and two are false negatives for estimated coverage data. These false negatives show the unsafety of using either outdated or estimated coverage data, and motivate the need to use updated coverage data.

To understand the results on larger and industrial programs, consider Tables 5, 6, 7, and 8. For *ProAX*, shown in Table 5, the results also degrade. For example, when DEJAVOO is run on (v_2, v_3), it selects 134 test cases using outdated coverage data and 99 test cases using estimated coverage using JDIFF, whereas only 45 test cases are selected using accurate coverage information. Of the selected test cases, 120 are false positives for outdated coverage data and 82 are false positives for estimated coverage data. Furthermore, for these versions of *ProAX*, 31 are false negatives using outdated coverage data and 28 are false negatives using estimated coverage data. For *Assent*, shown in Table 6, the results degrade significantly even for a single version. When DEJAVOO is run on (v_0, v_1), it selects 102 test cases using outdated coverage data and 87 test cases using estimated coverage using JDIFF, whereas only 42 test cases are selected using accurate coverage information. Of the selected test cases, 81 are false positives for outdated coverage data and 60 are false positives for estimated coverage data. Furthermore, for these versions of *Assent*, 21 are false negatives using outdated coverage data and 15 are false negatives using estimated coverage data.

For *JABA*, shown in Table 7, the results also degrade and are often significant. For example, when DEJAVOO is run on (v_8, v_9), it selects 582 test cases using outdated coverage data and 431 test cases using estimated coverage using JDIFF, whereas only 302 test cases are selected using accurate coverage information. Of the selected test cases, 401 are false positives for outdated coverage data and 231 are false positives for estimated coverage data. Furthermore, for these versions of *JABA*, 121 are false negatives using outdated coverage data and 72 are false negatives using estimated coverage data.

For *Darpan*, shown in Table 8, the results show significant degradation. When DEJAVOO is run on (v_2, v_3), it selects 589 test cases using outdated coverage data and 323 test cases using estimated coverage using JDIFF, whereas only 107 test cases are selected using accurate coverage information. Of the selected test cases, 409 are

TABLE 5
DejaVOO Results on *ProAX* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	85	58	10	76	46	3	33
(v_1, v_2)	82	73	10	72	58	5	19
(v_2, v_3)	134	120	31	99	82	28	45

TABLE 6
DejaVOO Results on *Assent* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	102	81	21	87	60	15	42

TABLE 7
DejaVOO Results on *JABA* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	6	0	0	6	0	0	6
(v_1, v_2)	57	54	0	41	38	0	3
(v_2, v_3)	182	180	12	56	42	0	14
(v_3, v_4)	196	178	40	115	67	0	58
(v_4, v_5)	201	188	20	63	30	0	33
(v_5, v_6)	210	206	12	39	23	0	16
(v_6, v_7)	707	602	0	325	220	0	105
(v_7, v_8)	707	681	0	37	12	1	26
(v_8, v_9)	582	401	121	431	231	72	302
(v_9, v_{10})	198	192	25	46	17	2	31
(v_{10}, v_{11})	707	666	0	68	67	3	41
:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:
(v_{308}, v_{309})	707	505	0	707	505	0	202

false positives for outdated coverage data and 304 are false positives for estimated coverage data. Furthermore, for these versions of *Darpan*, 107 are false negatives using outdated coverage data and 88 are false negatives using estimated coverage data.

The results of this study clearly show that, for our subject programs, in many cases, the results using outdated and estimated coverage data can differ significantly from the results using updated, accurate coverage data. In particular, the results show that there can be many test cases selected

TABLE 8
DEJAVOO Results on *Darpan* Using Outdated, Estimated, and Updated Coverage Data

	Outdated coverage data			Estimated coverage data using JDiff			Updated coverage data
	Test cases selected	False positives	False negatives	Test cases selected	False positives	False negatives	Test cases selected
(v_0, v_1)	812	801	232	232	186	167	213
(v_1, v_2)	623	614	142	381	358	128	151
(v_2, v_3)	589	409	107	323	304	88	107

that do not need to be rerun (i.e., false positives) and many test cases omitted that do need to be rerun because they traverse changes (i.e., false negatives).

4.4 Study 2

The goal of Study 2 is to evaluate research question RQ2.

> What is the effect of selective instrumentation in reducing the expense of running the test suite selected by the RTS algorithm?

To answer this research question, we performed two experiments. In the first experiment, we measured and compared the number of branches instrumented by the full instrumentation (i.e., instrumenting all branches in the program) and by the selective instrumentation (i.e., instrumenting only those branches that are computed as reachable by our method). We used the six subjects described in Section 4.2 for the study. Table 9 shows the results of the study. In the table, the first column shows the subject on which the experiment was performed. The second column shows the average number of branches instrumented when full instrumentation is applied to the subject and the third column shows the average number of branches instrumented when selective instrumentation is applied to the subject. Finally, the fourth column shows the savings that our selective instrumentation achieved over full instrumentation as a percentage of the number of branches instrumented for full instrumentation.

To understand the results, consider the fifth row of Table 9, which shows that *JABA* has 26,310 branches instrumented, on average, for all versions when full instrumentation is used, and 26,033 branches instrumented when selective instrumentation is used. For this subject, the use of selective instrumentation results in a savings of 1.06 percent, on average, over all versions of *JABA*. For another example, consider the third row of the table, which shows that *ProAX* has 19,672 branches instrumented, on average, for all versions when full instrumentation is used, and 14,573 branches instrumented when selective instrumentation is used. For this subject, the use of selective instrumentation results in a savings of 25.92 percent, on average, over all versions of *ProAX*.

The results show that, in all cases, our technique provides a savings in the number of branches instrumented (on average, 2 percent over all subjects) when selective instrumentation is used, although the results vary widely and often provide little savings. Thus, we performed a second experiment to further assess the savings that could be achieved using selective instrumentation. In the second experiment, we measured and compared the time to run the selected test suite T' on P_{mod} instrumented with full instrumentation (call this $P_{mod-full}$) and the time to run T' on P_{mod} instrumented with selective instrumentation (i.e., $P_{mod-inst}$). Table 10 shows the results of this study. In the table, the first column represents the subject on which the experiment was performed. The second column shows the time (in seconds) to run $P_{mod-full}$ on T' and the third column shows the time (in

TABLE 9
Average Number of Branches Instrumented on All Subjects with Full Instrumentation and with Selective Instrumentation

Subject	Full instrumentation (all branches)	Selective instrumentation (branches reachable from the change)	Savings
Jakarta Regexp	162	158	2.46%
NanoXML	596	581	2.51%
ProAX	19672	14573	25.92%
Assent	49269	49187	0.16%
JABA	26310	26033	1.06%
Darpan	82315	81842	0.57%

TABLE 10
Average Savings in the Time to Run the Subjects with Full Instrumentation and Selective Instrumentation on T'

Subject	Full instrumentation (time in seconds	Selective instrumentation (time in seconds)	Savings
Jakarta Regexp	32.13	28.69	10.71%
NanoXML	14.09	10.60	24.77%
ProAX	100.42	89.23	11.41%
Assent	151.04	138.94	8.01%
JABA	221.94	179.75	19.01%
Darpan	441.71	420.79	4.74%

seconds) to run $P_{mod-inst}$ on T'. Finally, the fourth column shows the savings in time that our selective instrumentation ($P_{mod-inst}$) achieved over full instrumentation ($P_{mod-full}$) as a percentage of number of seconds saved.

To understand the results, consider the second row of Table 10, which shows that *NanoXML* has savings 24.77 percent, on average, using selective instrumentation over full instrumentation. The results show that selective instrumentation reduces, on average, 26.13 percent of the time to run T'. Comparing these results with those in Table 9 shows that the savings in the number of probes is not an indicator of the savings in time. For example, the greatest savings in time was achieved by *NanoXML*, which had little savings in the number of probes.

The results of this study clearly show that there can be significant savings in the time to run the test suite selected by our RTS algorithm when selective instrumentation is used. The results further show that the reduction in the number of probes is not the only indicator of the savings that can be achieved.

4.5 Study 3

The goal of Study 3 is to evaluate research question RQ3.

> What is the efficiency of our technique for updating coverage data as part of a regression testing process?

To answer this question, we measured and compared regression-testing time for four approaches:

1. running all test cases in T on all versions of the program P (i.e., retest-all);
2. selecting T' using DEJAVOO and running the test cases in T' on all modified versions of P;
3. selecting T' and recording mappings using MOD-DEJAVOO, updating coverage data for $T - T'$ using RECOVER, instrumenting the modified versions of P with full instrumentation, and running the test cases in T' on the fully instrumented modified versions of P; and
4. selecting T' and recording the mappings using MOD-DEJAVOO, updating coverage data for $T - T'$ using RECOVER, instrumenting modified versions of P using selective instrumentation, and running test cases in T' on the selectively instrumented modified versions of P.

Table 11 shows the average timings for regression testing for the four techniques studied. In the table, the first column shows the subject on which the experiment was performed. The second column shows the sum of the time to perform RTS using DEJAVOO and the time to run the selected test cases T'. The third column shows the sum of the time to perform RTS

TABLE 11
Average Timings (in Seconds) for Regression Testing for Retest-All, RTS Using DEJAVOO only, RTS Using MOD-DEJAVOO with RECOVER and Full Instrumentation, and RTS Using MOD-DEJAVOO with RECOVER and Selective Instrumentation

Subject	Retest all (time in seconds)	DEJAVOO (time in seconds)	MOD-DEJAVOO + RECOVER (time in seconds)	MOD-DEJAVOO + RECOVER with Selective Instrumentation (time in seconds)
Jakarta Regexp	75.75	36.25	36.75	32.81
NanoXML	301.81	15.12	15.39	11.63
ProAX	266.83	109.75	115.37	98.56
Assent	406.02	211.25	235.49	199.15
JABA	490.29	254.75	262.13	212.06
Darpan	1447.91	499.82	515.33	478.9

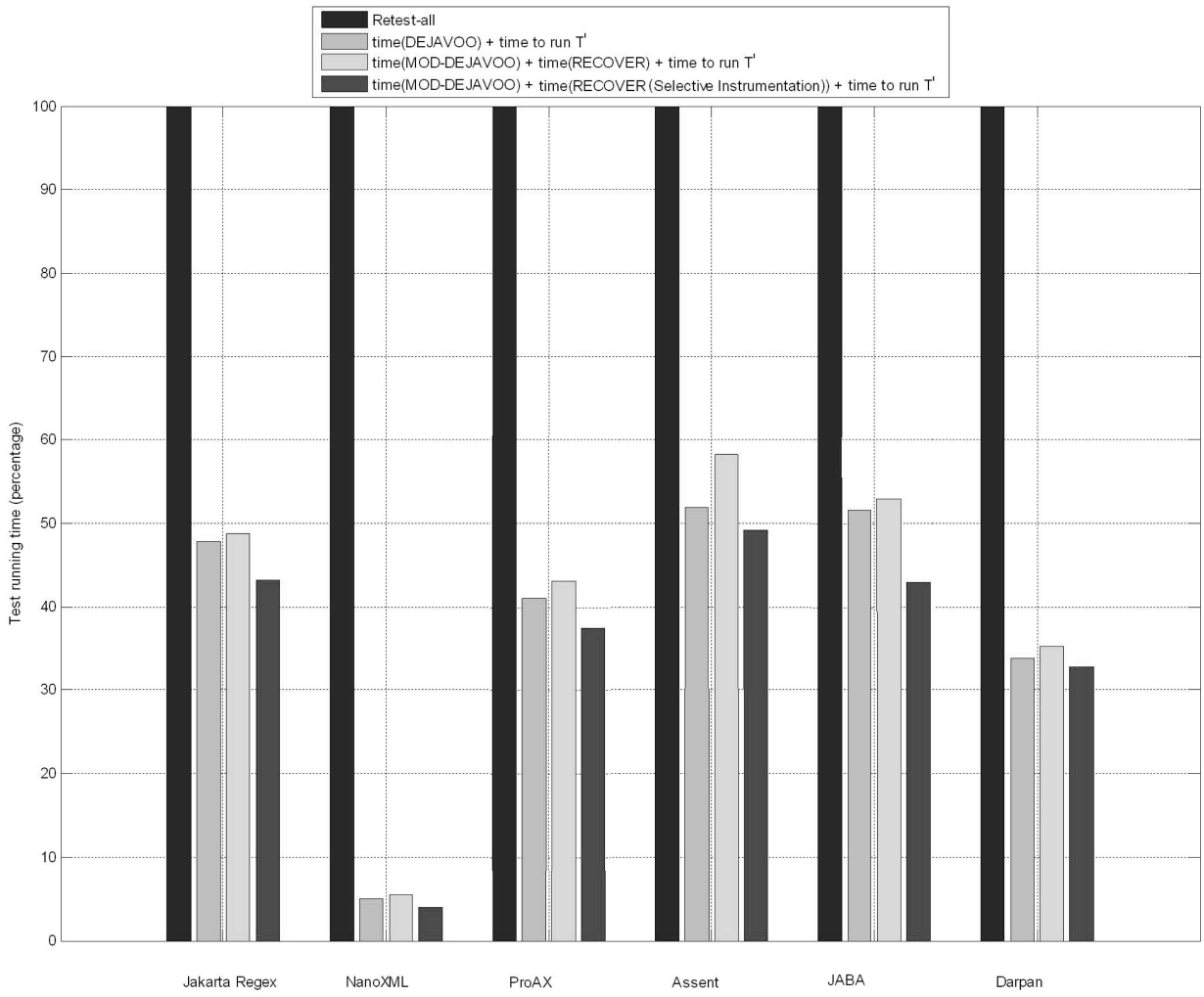

Fig. 9. Overall time for regression testing using retest-all, DEJAVOO only, MOD-DEJAVOO with RECOVER, and MOD-DEJAVOO with RECOVER and SELECTIVE INSTRUMENTATION.

using MOD-DEJAVOO with RECOVER and the time to run the selected test cases T'; in this case, the modified versions of the subject were fully instrumented. The fourth column shows the sum of the time to perform RTS using MOD-DEJAVOO with RECOVER and selective instrumentation and the time to run the selected test cases T' with the selectively instrumented modified versions of the subject.

To understand the results, consider the fifth row of Table 11, which shows that *JABA* took, on average, 490.92 seconds for retest-all, 254.75 seconds for DEJAVOO, 262.13 seconds for MOD-DEJAVOO with RECOVER, and 212.06 seconds for MOD-DEJAVOO with selectively instrumented RECOVER, with savings of 42 seconds per version. The results show that, in all cases, our technique provides savings in regression-testing time.

Fig. 9 shows the results of the study as a chart, where the bars represent the time for regression testing as a percentage of retest-all. In the chart, the horizontal axis represents the subjects and the vertical axis represents the time for performing regression testing using each of the four methods. For each subject, the chart shows four bars: the first bar represents the time for retest-all (100 percent in all cases); the second bar represents the sum of the time to perform RTS using DEJAVOO and the time to run the selected test cases T', as a percentage of the time for retest-all; the third bar represents the sum of the time to perform regression test selection using MOD-DEJAVOO, the time to recompute coverage information using RECOVER, the time to fully instrument the modified versions of the subject, and the time to run the selected test cases T', as a percentage of retest-all; the fourth bar represents the sum of the time to perform regression test selection using MOD-DEJAVOO, the time to recompute coverage information using RECOVER with selective instrumentation, and the time to run the selected test cases T', as a percentage of retest-all. The difference between the first and second bars shows the savings of DEJAVOO over retest-all. The difference between the second and third bars shows the overhead of our technique with respect to DEJAVOO using MOD-DEJAVOO without selective instrumentation. The difference between the third and fourth bars shows the savings of running MOD-DEJAVOO with selective instrumentation. Like Table 11, the chart shows that, in all cases, the overhead incurred by RECOVER, which uses MOD-DEJAVOO, over DEJAVOO is small.

Our technique using RECOVER with selective instrumentation saves, on average, 17.35 percent of the

regression-testing time for all our experimental subjects. More importantly, however, is that after using RECOVER, the coverage data are up-to-date for all test cases. In contrast, DEJAVOO selects only the subset of test cases T' to rerun on the modified version of the software, and thus, when those test cases in T' are rerun, only coverage data for those test cases can be updated. Coverage data for test cases in $T - T'$ are not updated—that coverage data are obtained by rerunning the test cases in $T - T'$, which wastes time, or using outdated or estimated coverage data, which can produce imprecise and unsafe results. Thus, in addition to not adding to the regression test selection time of DEJAVOO (and actually reducing it with selective instrumentation), our algorithm leaves the coverage matrix in a state that is ready for use on the next version of the program. Thus, it is efficient and effective when used in a regression testing process.

5 Related Work

To our knowledge, no other technique has been presented to solve the problem of providing accurate coverage information without rerunning all test cases in the test suite. However, several techniques are related in that they confirm the existence of the problem or provide alternative approaches.

In their empirical studies, Elbaum et al. showed the impact of software evolution on coverage data [11]. Their results, thus, motivate the need for our technique that provides accurate coverage data as the software evolves without requiring rerunning of all test cases as each software change is made. Our empirical studies, performed on a different set of subjects than Elbaum and colleagues, confirm their result on our subjects and highlight the savings that can be achieved using our technique.

ECHELON [13], a test suite prioritization tool, uses BMAT [16] (binary matching tool) to estimate coverage data without running the test cases in the test suite. BMAT performs matching on both code and data blocks between two versions of a program given in binary format. BMAT uses several heuristics to find matches for as many blocks as possible. JDIFF [12] is a differencing tool for Java bytecodes that can also be used to estimate coverage without running test cases. JDIFF performs matching on the bytecodes using a hierarchical reduction of structures of the control-flow graph. The studies in [12] show that the quality of the estimated coverage degrades when the number of changes increases. Our empirical studies confirm this degradation and the effect that this inaccurate coverage can have for regression test selection.

6 Conclusion

In this paper, we have presented a technique that provides updated coverage data for a modified program without running all test cases in the test suite that was developed for the original program and used for regression testing. The technique is safe and precise in that it computes exactly the same information as if all test cases in the test suite were rerun, assuming that the regression test selection technique that it leverages is safe. Our technique leverages the information provided by a safe regression test selection tool DEJAVOO and modifies it to get MOD-DEJAVOO, which provides additional information for mapping of matching nodes. Our selective instrumenter instruments only the affected branches, and thus, reduces the amount of instrumentation. By running the test cases selected by MOD-DEJAVOO (which are the same as those selected by DEJAVOO), the technique updates coverage data for test cases that exercise the change. Using the mapping information provided by MOD-DEJAVOO, the technique updates coverage data for test cases that do not exercise changes.

In this paper, we also presented the results of three empirical studies on a set of subject programs of varying sizes, along with versions of those programs and test suites used to test them. Three of the subjects, PROAX, ASSENT, and DARPAN, are industrial applications developed at Tata Consultancy Services, Ltd. The first study confirms that regression test selection using outdated and estimated coverage data causes the regression test selection algorithm to both select unnecessary test cases and omit important test cases. Thus, updated, accurate coverage data are required for effective regression test selection. The updated coverage data that our technique computes are identical to that computed if all test cases were rerun (assuming a safe regression test selection algorithm). Thus, it provides the same results as rerunning the entire test suite after each version is created and provides accurate (updated) coverage data for use in regression test selection for the next version of the program. The second study shows that selective instrumentation saves in the number of probes that are required for running the test cases selected by the regression test selection algorithm. This reduction results in a savings in the time to run the test cases selected, and thus, reduces the overall regression testing time. The third study shows that our technique with selective instrumentation reduces the time required for regression testing over DEJAVOO. Thus, our technique provides an efficient way to keep updated coverage data for use in software-maintenance tasks. Overall, our studies showed that our technique can provide savings in regression-testing time.

Encouraged by the reduction that resulted by the simple selective instrumentation (our current technique provides only simple reachability analysis), we plan to investigate efficient dataflow and slicing techniques to further reduce the instrumentation so that we can get even greater savings in the time for the regression testing. We are currently integrating RECOVER into a selective-retest environment that includes regression test selection, test case prioritization, test suite augmentation, fault localization, and visualization.

Acknowledgments

This work was supported in part by US National Science Foundation (NSF) awards CCF-0541049, CCF-0429117, and CCF-0306372 to Georgia Tech and Tata Consultancy Services, Ltd. The anonymous reviewers provided many helpful suggestions that improved the presentation of this paper. An earlier version of this paper, entitled "Recomputing Coverage Information to Assist Regression Testing," appeared in the *Proceedings of the 23rd IEEE*

International Conference on Software Maintenance, Paris, France, pages 164-173, October 2007.

REFERENCES

[1] C. Kaner, "Improving the Maintainability of Automated Test Suites," *Proc. Quality Week Conf.,* May 1997.

[2] B. Beizer, *Software Testing Techniques.* Van Nostrand Reinhold, 1990.

[3] H.K.N. Leung and L.J. White, "Insights into Regression Testing," *Proc. IEEE Conf. Software Maintenance,* pp. 60-69, Oct. 1989.

[4] T. Ball, "On the Limit of Control Flow Analysis for Regression Test Selection," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 134-142, Mar. 1998.

[5] Y.F. Chen, D.S. Rosenblum, and K.P. Vo, "Testtube: A System for Selective Regression Testing," *Proc. 16th ACM/IEEE Int'l Conf. Software Eng.,* pp. 211-222, May 1994.

[6] A. Orso, N. Shi, and M.J. Harrold, "Scaling Regression Testing to Large Software Systems," *Proc. 12th ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 241-252, Nov. 2004.

[7] G. Rothermel and M.J. Harrold, "A Safe, Efficient Regression Test Selection Technique," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 2, pp. 173-210, Apr. 1997.

[8] G. Rothermel, M.J. Harrold, and J. Dedhia, "Analyzing Regression Test Selection Techniques," *IEEE Trans. Software Eng.,* vol. 22, no. 8, pp. 529-551, Aug. 1996.

[9] F. Vokolos and P. Frankl, "Pythia: A Regression Test Selection Tool Based on Text Differencing," *Proc. IEEE Int'l Conf. Reliability, Quality and Safety of Software Intensive Systems,* pp. 3-21, June 1997.

[10] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, and A. Gujarathi, "Regression Test Selection for Java Software," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 312-326, Nov. 2001.

[11] S. Elbaum, D. Gable, and G. Rothermel, "The Impact of Software Evolution on Code Coverage Information," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 170-179, Nov. 2001.

[12] T. Apiwattanapong, A. Orso, and M.J. Harrold, "A Differencing Algorithm for Object-Oriented Programs," *Proc. 19th IEEE Int'l Conf. Automated Software Eng.,* pp. 2-13, Sept. 2004.

[13] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment," *Proc. 2002 ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 97-106, July 2002.

[14] A. Chawla and A. Orso, "A Generic Instrumentation Framework for Collecting Dynamic Information," *Proc. ISSTA Workshop Empirical Research in Software Testing,* July 2004.

[15] P.K. Chittimalli, M. Bapat, and R.D. Naik, "ProAX: A Program Analysis and Transformation Framework," *Proc. Theoretical Computer Science Technical Architect's Conf.,* Dec. 2004.

[16] Z. Wang, K. Pierce, and S. McFarling, "BMAT—A Binary Matching Tools for Stale Profile Propagation," *J. Instruction-Level Parallelism,* vol. 2, pp. 1-20, Mar. 2000.

Pavan Kumar Chittimalli received the MTech degree in computer science from the Indian Institute of Technology, Guwahati, India. Since 1999, he has been with Tata Research Development and Design Center (TRDDC), an R&D center of Tata Consultancy Services Limited, Pune, India. His research interests include program analysis, reverse engineering, and testing. To date, his research has investigated regression test selection and test adequacy criteria generation for software under maintenance. He is a member of the ACM and ACM SIGSOFT.

Mary Jean Harrold received the PhD degree in computer science from the University of Pittsburgh. She is the ADVANCE professor of computing and the Associate Dean for Faculty Affairs in the College of Computing at the Georgia Institute of Technology. She performs research in analysis and testing of large evolving software, in fault localization and failure identification using statistical analysis, machine learning, and visualization, and in monitoring deployed software to improve quality. She has received funding for her research from government agencies, such as the US National Science Foundation (NSF) and NASA, and industries, such as Boeing Commercial Airplanes, Tata Consultancy Services, IBM, and Microsoft. She served on the editorial boards of ACM TOPLAS and TOSEM and JSTVR, served as program chair of ACM ISSTA 2000, program cochair of the ACM/IEEE ICSE 2001, and general chair of the ACM SIGSOFT FSE 2008. She also serves as a member of the Board of Directors for the Computing Research Association (CRA). She actively works to increase the participation of women in computing. She is a member and past cochair of CRA-W and a member of the Leadership Team of the National Center for Women and Information Technology (NCWIT). She received an NSF NYI Award and was named an ACM fellow. She is a senior member of the IEEE and a member of the IEEE Computer Society.