Re-computing Coverage Information to Assist Regression Testing

Pavan Kumar Chittimalli  
Tata Research Design and Development Center  
Tata Consultancy Services Limited  
pavan@cc.gatech.edu

Mary Jean Harrold  
College of Computing  
Georgia Institute of Technology  
harrold@cc.gatech.edu

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 $P_{i+1}$. Users of our technique can avoid the expense of re-running the entire test suite on $P_{i+1}$ or the inaccuracy produced by previous approaches that estimate coverage data for $P_{i+1}$ or reuse outdated coverage data from $P_{i}$.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. The studies show the inaccuracies that can exist when an application—regression-test selection—uses estimated and outdated coverage data. The studies also show that the overhead incurred by our technique is negligible.

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, and can account for as much as one-half of the cost of software maintenance [3, 8].

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 $P_{i+1}$. Because 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., [2, 4, 9, 10, 11, 13]). If the RTS technique is safe, then the test cases that it omits ($T_{i} - T'_{i}$) will give the same results on $P_{i}$ and $P_{i+1}$, and thus, do not need to be rerun on $P_{i+1}$ [6, 10]. Studies have shown that RTS can be effective in reducing the cost of regression testing (e.g., [4, 9, 10, 13]).

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 done on $P_{i+1}$. For example, an RTS technique collects coverage data, such as statement, branch, or method coverage, 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 is 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 be simply copied for $P_{i+1}$ unless the development environment maintains a mapping between entities (such as statements, branches, or 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 and colleagues [5] 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, 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 causes 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 applied 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., [1, 12]). 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 outdated or estimated coverage data, we have developed, and present in this paper, a technique that leverages an existing RTS technique [9] 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 augments the RTS algorithm so that as it identifies the changed entities in $P_i$ 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 by changes. Second, the technique instruments $P_{i+1}$ to collect coverage data for $P_{i+1}$, and runs the instrumented version of $P_{i+1}$ with $T'_i$ to get updated coverage for affected parts of $P_{i+1}$. Third, the technique uses the mapping computed in the first step to transfer the coverage data for test cases not selected for rerunning (i.e., those in $T_i - T'_i$). Finally, the technique combines the coverage data from the second and third steps to get the updated, accurate coverage data for $P_{i+1}$ with respect to $T_i$.

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, such as 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.

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 KLOC to 80 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 three subjects we used, RTS applied to outdated coverage data resulted in, on average, 62.61%, 83.26%, and 85.12% false positives, respectively, and 3.49% false negatives over RTS used with updated coverage data; RTS applied to estimated coverage data resulted in, on average, 0.68%, 54.52%, and 70.12% false positives, respectively, and 2.12% false negatives over RTS used with updated coverage data. The studies also show the efficiency of our technique. For the three subjects we used, when RTS is augmented with our technique to compute the mappings, the overhead is 1.50%, on average, over RTS without our augmentation.

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.
- 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.

## 2 Background, 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. 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 Figures 1, 2, and 3, which show 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. Figures 1, 2, and

---

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

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

3 False positives represent test cases that do not need to be rerun and would not be selected with accurate, updated coverage data.

4 False negatives represent test cases that should have been selected, because they go through changes, but were not.
public class Grade {
    public int calcGrade(int finalScore, int midTermScore) {
        int grade = 0;
        if (finalScore >= 100) {
            grade = 4;
        } else if (finalScore >= 90) {
            grade = 3;
        } else if (finalScore >= 80) {
            grade = 2;
        } else if (finalScore >= 70) {
            grade = 1;
        } else if (finalScore >= 60) {
            grade = 0;
        }
    return grade;
    }
}

public class Grade {
    public int calcGrade(int finalScore, int midTermScore) {
        int grade = 0;
        if (finalScore >= 100) {
            grade = 4;
        } else if (finalScore >= 90) {
            grade = 3;
        } else if (finalScore >= 80) {
            grade = 2;
        } else if (finalScore >= 70) {
            grade = 1;
        } else if (finalScore >= 60) {
            grade = 0;
        }
    return grade;
    }
}

Figure 1. Version \( v_0 \) of program \( \text{Grade} \) and its statement-coverage matrices.

Figure 2. Version \( v_1 \) of program \( \text{Grade} \) and its statement-coverage matrices.

Figure 3. Version \( v_2 \) of program \( \text{Grade} \) and its statement-coverage matrices.

3 also show the corresponding 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 each version, the matrix on the left shows the outdated coverage data when the coverage data for \( v_0 \) is used for the subsequent versions, and the matrix on the right shows the updated coverage data when \( T \) is run on each version. The “?” 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 \( v_0 \). Note that the outdated coverage data and the updated coverage data are the same for \( v_0 \) because \( T \) is run with this base version. Also note that version \( v_0 \) has 100% statement coverage with

<table>
<thead>
<tr>
<th>Test cases</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 )</td>
<td>finalScore=71, midTermScore=81</td>
</tr>
<tr>
<td>( t_2 )</td>
<td>finalScore=71, midTermScore=71</td>
</tr>
<tr>
<td>( t_3 )</td>
<td>finalScore=34, midTermScore=60</td>
</tr>
<tr>
<td>( t_4 )</td>
<td>finalScore=52, midTermScore=50</td>
</tr>
</tbody>
</table>

Table 1. Test suite \( T \) for the \( \text{calcGrade} \).
To illustrate the impact that outdated coverage data can have on a regression-testing activity that uses it, we consider regression-test selection (RTS), which was briefly described in Section 1. We use an RTS technique implemented as DEJAVOO [9]. DEJAVOO creates control-flow graphs for the original \((P_{\text{orig}})\) and modified \((P_{\text{mod}})\) versions of a program. The technique traverses these graphs synchronously over like-labeled edges. Like-labeled edges in \(P_{\text{orig}}\) and \(P_{\text{mod}}\) 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 \(P_{\text{orig}}\) should be rerun on \(P_{\text{mod}}\) because they may behave differently in \(P_{\text{mod}}\).

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 Figure 5, with \(v_0\) on the left and \(v_1\) in the center. Using the control-flow graphs, DEJAVOO traverses like-labeled edges at the Entry node of each graph, and finds all sinks on the path \(s_1, s_2, s_3, s_4, s_9, s_{10}\) (\(s_0\) is equivalent to \(s_0\) in \(v_0\) identical to the sinks on the path \(s_1, s_2, s_3, s_4, s_{11}, s_{12}\) (\(s_{12}\) is equivalent to \(s_{12}\) in \(v_1\)) Note that, although they have different node numbers, \(s_9\) and \(s_{11}\) are identical, as are \(s_{10}\) and \(s_{12}\). When the traversal continues at \(s_3\) in the two graphs, it finds that \(s_4\) in \(v_0\) and \(s_4\) in \(v_1\) match. However, when the traversal reaches edges \(e_4\) in the two graphs (with source \(s_2\) and sink \(s_6\)), it finds that the sinks, \(s_6\) in both graphs, differ (an inspection of the code shows that the statements corresponding to \(s_6\) in \(v_0\) and \(v_1\) differ). The algorithm then marks \(e_4\) in \(v_0\) as a dangerous edge. Using either the outdated or updated matrix for \(v_0\), which are identical, we see that test case \(t_4\) traverses \(e_4\) in \(v_0\), and thus it is selected to run on \(v_0\).

Now consider running DEJAVOO on \(v_1\) and \(v_2\). Using the control-flow graphs for \(v_1\) and \(v_2\) in Figure 5, DEJAVOO traverses like-labeled edges and finds that all corresponding nodes match until edge \(e_{10}\) is reached. Because the sinks of \(e_{10}\) differ in the two programs, \(e_{10}\) is marked as a dangerous edge, and test cases are selected for it. Using the outdated matrix for \(v_1\), DEJAVOO selects the set that executed \(s_8\) \(\{t_4\}\). However, using the updated matrix for \(v_1\), DEJAVOO selects \(\{t_3, t_4\}\). 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, \(t_3\), that executes the change and should be rerun. In this case, the use of outdated coverage data caused 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 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 [9]. Thus, our algorithm provides updated coverage data without rerunning all test cases in the original test suite. Our algorithm, ReCOMPUTE\(\text{MATRIX}\), is shown in Figure 4. ReCOMPUTE\(\text{MATRIX}\) takes four inputs: \(P_{\text{orig}}\) and \(P_{\text{mod}}\), original and modified versions, respectively; \(T\), a set of test cases that was run on \(P_{\text{orig}}\); and \(m\), the coverage data for \(P_{\text{orig}}\) when run with \(T\), represented as a matrix \([n(E) \times n(T)]\), where \(E\) is the set of entities and \(n()\) returns the size of the set. ReCOMPUTE\(\text{MATRIX}\) outputs \(m_{\text{mod}}\), an accurate, updated matrix for \(P_{\text{mod}}\). The algorithm uses an external function, \(\text{instrument}(P)\), that instruments entities in the program by placing probes for monitoring.

```
ALGORITHM ReCOMPUTE\(\text{MATRIX}()\)
Input: \(P_{\text{orig}}, P_{\text{mod}}\): Original and modified versions, respectively
\(T\): set of test-cases run on \(P_{\text{orig}}\)
\(m\): coverage matrix \([E \times T]\) for \(P_{\text{orig}}\) when run with \(T\), where \(E\) is a set of entities in \(P_{\text{orig}}\)
Output: \(m_{\text{mod}}\): coverage matrix \([E \times T]\) for \(P_{\text{mod}}\)
Use: \(\text{instrument}(P):\) instruments program \(P\) for coverage
Declare: \(\text{entityMap}: \{(e, e'): e, e'\text{ entities in } P_{\text{orig}}, P_{\text{mod}}, \text{respectively}\}\)
\(T': \) set of test-cases
\(e, e': \) a entities in program

// Step 1
(1) \(m_{\text{mod}} = \text{create matrix for } P_{\text{mod}}\)
(2) \(m_{\text{mod}}[e', t] = 0 \text{ for all } e' \in P_{\text{mod}}, t \in T\)
(3) \((T', \text{entityMap}) = \text{MOD-DEJAVOO}(P_{\text{orig}}, P_{\text{mod}})\) // Step 2
(4) \(P_{\text{mod-inst}} = \text{instrument}(P_{\text{mod}})\) // Step 3
(5) foreach \(t \in T'\)
(6) \(\text{run} _{\text{Program}}(P_{\text{mod-inst}})\)
(7) endfor
(8) foreach \((e, e') \in \text{entityMap}\)
(9) foreach \(t \in T - T'\)
(10) \(m_{\text{mod}}[e', t] = m[e, t]\)
(11) endfor
(12) endfor
(13) return \(m_{\text{mod}}\)
```

Figure 4. ReCOMPUTE\(\text{MATRIX}\) algorithm.
RECOMPUTE_MATRIX consists of three main steps: (1) creating and initializing the coverage matrix, $m_{mod}$, for $P_{mod}$ (lines 1-2); (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); and (3) creating the instrumented version of the program, $P_{mod-inst}$, and running it with $T'$ to get coverage data for the affected parts of $P_{mod}$, and using the mappings stored in entityMap to transfer the coverage data for the unaffected parts of $P_{mod}$ (lines 4-12). RECOMPUTE_MATRIX then returns $m_{mod}$ (line 13).

In Step 1, the algorithm initializes $m_{mod}$, the matrix for $P_{mod}$, with size the number of entities in $P_{mod}$ (line 1). For example, when RECOMPUTE_MATRIX is run on $v_0$ and $v_1$, $m_{mod}$ is given size 12—the number of statements in $v_1$. Next, RECOMPUTE_MATRIX initializes all entries in this matrix with 0, indicating that none of the entities in $P_{mod}$ are covered.

In Step 2, RECOMPUTE_MATRIX 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 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 also stores mapping information about the nodes that it visits, in entityMap (line 3).

To illustrate, consider Figure 5, wherein the control-flow graphs of programs $v_0$ (Figure 1), $v_1$ (Figure 2), and $v_2$ (Figure 3) are shown. When MOD-DEJAVOO examines $v_0$ and $v_1$, it returns $\{t_3, t_4\}$ for $T'$ and $\{(s_1, s_1), (s_2, s_2), (s_3, s_3), (s_4, s_4), (s_5, s_5), (s_9, s_11), (s_{10}, s_{12})\}$ 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. The graphs in Figures 5 illustrate these sets by showing them in the modified program—the changes are shown in starred regions, the not-examined nodes are shown in clouded regions, and the remainder of the nodes are matched. For example, when MOD-DEJAVOO examines $v_0$ and $v_1$, the graph in the center of Figure 5 representing $v_1$ shows that $s_6$ is in the change set, nodes $s_7$-$s_{10}$ are in the not-examined set, and the rest of the nodes are matched. The starred region in $v_1$ depicts the change identified from $v_0$ to $v_1$; likewise, the starred region in $v_2$ depicts the change identified 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 $v_2$. 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 $v_2$.
of the nodes are in the match set.

In Step 3, \texttt{RECOMPUTE MATRIX} instruments the entities in \textit{P} to get \textit{P} so new coverage data can be collected for affected parts of \textit{P} (line 4). \textit{P} is then executed with \textit{T} (lines 5-7) to get coverage data for all entities with respect to \textit{T}.

Next \texttt{RECOMPUTE MATRIX} considers entities in the match set. Because of the technique used by \texttt{DEJAVA VO}, and thus by \texttt{MOD-DEJAVA VO}, for traversing the graphs, entities in the match set are on at least one path from the beginning of the program that does not go through the changes (i.e., these entities are not dominated by changes). Test cases in \textit{T} – \textit{T}’ that exercise these match-set entities will behave the same way they did when run on \textit{P}. This selection also guarantees that entities in the \textit{matched} set have the same coverage data on \textit{P} and \textit{P} when \texttt{RECOMPUTE MATRIX} uses \textit{entityMap}. \texttt{RECOMPUTE MATRIX} populates \textit{m}_{mod} for test cases in \textit{T} – \textit{T}’ by transferring the coverage data for \textit{P} to \textit{P} and thus the \textit{m}_{mod} using \textit{entityMap}.

Thus, \texttt{RECOMPUTE MATRIX} provides correct updated coverage data, given the safe RTS technique that MOD-DEJAVA VO uses.

4 Empirical Studies

To evaluate our technique, we developed a tool called \texttt{RECOVER} (Recomputing Coverage Data) that implements our algorithm, and used it to conduct two empirical studies. In this section, we first describe \texttt{RECOVER}, then present our empirical setup, and finally present the two studies.

4.1 \texttt{RECOVER}

Figure 7 shows the architecture of \texttt{RECOVER}, which consists of three components:

![Figure 7. Architecture of \texttt{RECOVER.}](image)
• **Mod-DejaVOO** implements the **Mod-DejaVOO** component of algorithm ReCompuTeMatrix. Mod-DejaVOO inputs program \( P_{\text{orig}} \) and its coverage matrix \( m \), modified version \( P_{\text{mod}} \), and test-suite \( T \). Mod-DejaVOO outputs the entity correspondence, \( \text{entityMap} \), between \( P_{\text{orig}} \) and \( P_{\text{mod}} \), along with \( T' \), the set test cases to run on \( P_{\text{mod}} \).

• **InStrumEntEr** inputs \( P_{\text{mod}} \) and adds probes to it to create \( P_{\text{mod-inst}} \) so that as \( P_{\text{mod-inst}} \) executes, it records coverage and populates the coverage matrix. InStrumEntEr outputs \( P_{\text{mod-inst}} \), and creates and outputs the matrix \( m_{\text{mod}} \) for program \( P_{\text{mod}} \).

• **ReCover Engine** inputs \( P_{\text{mod-inst}}, T, T', \text{entityMap} \), and \( m_{\text{mod}} \). ReCover Engine initializes each entry in coverage matrix \( m_{\text{mod}} \) with ‘0’. It populates the coverage information for \( T' \) by running the set of test cases in \( T' \) on \( P_{\text{mod-inst}} \). It populates the coverage information for \( T - T' \) using \( \text{entityMap} \) to transfer coverage data for the matched entities from \( m \) to \( m_{\text{mod}} \). ReCover Engine then outputs the updated coverage matrix \( m_{\text{mod}} \), which is the result of ReCover.

### 4.2 Subjects

For our studies, we used three subjects as shown in Table 2—Jakarta Regex, NanoXML, and JABA. Jakarta Regex, exp, is a Java regular-expression package. Jakarta Regex has 14 classes, 1000-2000 lines of code, depending on the version, and a test suite of size 148.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size in KLOC</th>
<th>Number of versions</th>
<th>Test-suite size</th>
</tr>
</thead>
<tbody>
<tr>
<td>JakartaRegex</td>
<td>1 - 2</td>
<td>4</td>
<td>148</td>
</tr>
<tr>
<td>NanoXML</td>
<td>3 - 4</td>
<td>33</td>
<td>214 - 216</td>
</tr>
<tr>
<td>JABA</td>
<td>70 - 80</td>
<td>309</td>
<td>707</td>
</tr>
</tbody>
</table>

**NanoXML**, an XML processor. NanoXML has six versions, and 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 ten versions, and \( v_5 \) has nine versions. Thus, we performed our studies on 33 versions of NanoXML. NanoXML has 3000-4000 lines of code depending on the version, and test suites of sizes ranging from 214 to 216.

**JABA** (Java Architecture for Bytecode Analysis)\(^3\) is an extensible API that supports research in program-analysis for Java and the development of program-analysis-based software-engineering tools for Java. JABA has 309 versions, with the size of the versions ranging from 50-80 KLOC, depending on the version, and a test suite of size 707.

### 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 Jakarta Regex, NanoXML, and JABA, as 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 \( m_{v_0} \), the coverage data for version \( v_0 \) of program \( P \). This coverage data is used for subsequent RTS activities. For estimated coverage data, we used JDiff [1] to estimate the coverage data for each version \( v_{i+1} \) using \( v_i \). For updated coverage data, we used our tool ReCover to calculate \( m_{v_{i+1}} \) of version \( v_{i+1} \) using \( m_{v_i} \) of version \( v_i \) on subsequent versions of \( P \). However, the updated coverage data that our technique computes is identical to that 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 \( P' \) and comparing this accurate coverage data with those obtained using ReCover. In all cases the coverage data were the same.

Table 3 shows the results of the study. The first column shows the versions on which DejaVOO was run. The next three columns show the number of test cases selected using DejaVOO, along with the number of false positives\(^3\) and false negatives\(^4\) when DejaVOO is applied to outdated coverage data. The next three columns show similar results when DejaVOO is applied to coverage data estimated using JDiff. The last column shows the number of test cases selected by DejaVOO using updated coverage data, which is the same coverage data as would be obtained by rerunning all test cases in the test suite. The table has three groups of rows that give the results for Jakarta Regex, DejaVOO, and JABA, respectively. For Jakarta Regex, the table shows the results of running DejaVOO on all pairs of versions. However, for NanoXML, and JABA, the table shows only a representative part of the results of running DejaVOO on all pairs of versions.

To understand the results, consider Table 3. The first row of each group in the table shows that when DejaVOO is run on the original version \( (v_0) \) of that subject 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 is accurate for \( v_0 \). However, when DejaVOO is run with subsequent changes, the results degrade. To see this, consider the first group in Table 3, which shows the results for Jakarta Regex. For both versions, DejaVOO selects all test cases (e.g., 148) using the outdated coverage data, many of which are false positives.\(^3\) For example, for \( v_1 \) and \( v_2 \), there are 133 false positives because only 15 test

cases are selected with accurate coverage data, and for \( v_2 \) to \( v_3 \), there are 145 false positives because only 3 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 \( v_1 \) to \( v_2 \)).

The results degrade and vary even more with larger programs. To see this, next consider the other two groups in Table 3. For \( \text{NanoXML} \) in the next group, there are many examples of false positives. For example, for \( v_2 \) and \( v_{2,1} \), 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 coverage data. Furthermore, for these versions of \( \text{NanoXML} \), 25 are false negatives for outdated coverage data, and two are false negatives for estimated coverage data. Whereas false positives are test cases that do not need to be rerun, false negatives are test cases that should have been selected, because they go through changes, but were not. These false negatives show the unsafety of using either outdated or estimated coverage data, and motivate the need to use updated coverage data.

For \( \text{JABA} \), shown in the third group of the table, the results also degrade and are often significant. For example, when \( \text{DEJAVOO} \) is run on \( v_8 \) and \( v_9 \), it selects 582 test cases using outdated coverage data and 431 test cases using estimated coverage using \( \text{JDiff} \), 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 31 are false positives for estimated coverage data. Furthermore, for these versions of \( \text{JABA} \), 121 are false negatives using outdated coverage data, and 72 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 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 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 three 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 then running the test cases in $T'$ on all modified versions of $P$. (3) selecting $T'$ and recording mappings using MOD-DEJAVOO, then updating coverage data for $T - T'$ using RECOVER, and finally running test cases in $T'$ on versions $v_0, v_1, ..., v_n$.

Figure 8 shows the results of the study. The horizontal axis represents each subject for retest-all, DEJAVOO, and RECOVER; the vertical axis represents the time for running the test cases on the subjects expressed as a percentage of the time for running tests using retest-all. For each subject, the chart shows three bars: the bar on the left represents the time for retest-all, which is 100% in all cases; the bar in the center represents sum of the times to run DEJAVOO and the selected test cases $T'$; the bar on the right represents the sum of the times to run the MOD-DEJAVOO, RECOVER ENGINE, and the selected test cases $T'$. The difference between the first and second bars shows the savings over retest all, and difference between second and third bars shows the overhead of our technique with respect to RTS using DEJAVOO. The chart clearly shows that our technique results in little overhead in the regression-testing time. To see the actual timings, consider Table 4, which shows the average timings for the RTS for all subjects using DEJAVOO, and MOD-DEJAVOO with RECOVER. For example JABA took on average 254.75 seconds for DEJAVOO and 262.13 seconds for MOD-DEJAVOO with RECOVER, with only 7.38 seconds per version (1.5%) overhead compared to DEJAVOO.

The results show that, in all cases, our technique provides a savings in regression-testing time. Furthermore, the increase in time required for RECOVER over that required for DEJAVOO is quite small—less than 1.5% in all cases—and supports our claim that the technique adds negligible overhead to regression-test selection.

### 5 Related work

No other existing 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.

Elbaum and colleagues showed the impact of software evolution on coverage data in their empirical studies [5]. 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 the Elbaum et al. study confirms their result on our subjects, and highlight the savings that can be achieved using our technique.

ECHELON [12] is a test-suite prioritization tool that uses BMAT [14](binary matching tool) to estimate coverage information 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 [1] 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 hammocks. The
6 Conclusion

In this paper, we have presented a technique that provides updated coverage data for a modified program without running all the 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. 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 two empirical studies on a set of subject programs of varying sizes, along with versions of those programs and test suites used to test them. The first study confirmed that regression-test selection using outdated and estimated coverage data caused the regression-test-selection algorithm to both select unnecessary test cases and omit important test cases. Thus, updated, accurate coverage data is required for effective regression-test selection. The updated coverage data that our technique computes is 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. The second study showed that the cost of our technique is quite small over the regression-test selection and compared to rerunning all test cases in the original test suite. Thus, our technique provides an efficient way to keep updated coverage data for use in software-maintenance tasks.

In future work, we will perform additional studies on real, industrial software, versions, and test suites. These studies will let us determine the practical implications of our technique. We will also investigate the use of the change information to create a selective instrumentation technique that is based on instrumenting only those statements impacted by the changes. Such instrumentation could provide savings in rerunning the selected test cases. Finally, we plan to integrate RECOVER into a selective-retest environment that includes test-case prioritization, test-suite augmentation, fault-localization, and visualization.

7 Acknowledgments

This work was supported in part by NSF awards CCF-0541049, CCF-0429117, and CCF-0306372 to Georgia Tech, by Tata Consultancy Services, Ltd. and by the State of Georgia to Georgia Tech under the Yamacraw Mission.

References