Selecting Tests and Identifying Test Coverage Requirements for Modified Software*

Gregg Rothermel and Mary Jean Harrold
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
{grother,harrold}@cs.clemson.edu

Abstract

Regression testing is performed on modified software to provide confidence that changed and affected portions of the code behave correctly. We present an approach to regression testing that handles two important tasks: selecting tests from the existing test suite that should be rerun, and identifying portions of the code that must be covered by tests. Both tasks are performed by traversing graphs for the program and its modified version. We first apply our technique to single procedures and then show how our technique is applied at the interprocedural level. Our approach has several advantages over previous work. First, our test selection technique is safe, selecting every test that may produce different output in the modified program. However, our selection technique chooses smaller test sets than other safe approaches. Second, our approach is the first safe approach to identify coverage requirements, and the first safe approach to do so interprocedurally. Third, our approach handles both structural and nonstructural program modifications, and processes multiple modifications with a single application of the algorithm. Finally, our approach can be automated, and is applicable to all programs for which program dependence graphs can be constructed.

1 Introduction

Regression testing is an expensive task performed on modified software to provide confidence that modified code behaves as intended, and that modifications have not inadvertently disrupted the behavior of unmodified code. One difference between regression testing and developmental testing is that at regression test time, we have access to existing test suites, and can reuse tests from those suites instead of generating new tests.

Regression testing techniques reuse test suites in different ways. Coverage techniques use structural coverage criteria to select existing tests that exercise changed or affected program components. Some coverage techniques[6, 9, 22] minimize the set of tests selected; others[3, 4, 7, 8, 15, 17, 23] select all tests that exercise changed or affected program components. In either case, coverage techniques identify tests that can be used to satisfy structural coverage criteria, but they may omit some tests that can cause the modified program to exhibit different output (and possibly expose faults). Moreover, many coverage techniques process each program change individually; these techniques may be impractical for real programs.

A second group of regression testing techniques[1, 14, 20], is composed of safe techniques, which select every test that will cause a modified program to produce different output than its original version. However, existing safe techniques do not sufficiently support the use of structural coverage criteria. Furthermore, these techniques have only been applied to individual procedures.

To address the deficiencies in current techniques, we present a regression testing technique that is both safe, and supports the use of structural coverage criteria. Our technique partitions an existing test suite into two subsets: tests that may cause the modified program to exhibit different output, and tests that cannot result in different output. Tests in the first subset may expose faults, so all of them are selected; tests in the other subset are not selected. Our technique also ana-

---

*This work was partially supported by a grant from Data General Corporation, and by NSF under Grants CCR-9100531 and CCR-9357811 to Clemson University.

1For a more precise description of the regression testing problem and a typical solution to that problem, see Section 2.

2We show this in [19] where we compare regression testing techniques.
lyzes the modified program and its original version to identify code affected by modifications; affected code must be exercised by tests to ensure adequate coverage of the modified program. We perform both test selection and identification of coverage requirements by traversing graphs for the original and modified versions of a program. When dealing with individual procedures, we use the program dependence graph[5]; for groups of procedures or entire programs, we use a variant of the system dependence graph[11].

Our approach has several advantages over previous approaches. First, our test selection technique selects every test that may produce different output in the modified program, and thus is safe. However, our technique selects smaller safe test sets than other safe techniques, and hence is more precise than those techniques. Second, although many previous approaches help the user identify test coverage requirements, ours is the first safe approach to do so, and the first safe approach to do so interprocedurally. Third, our approach handles both structural and nonstructural program modifications, and processes multiple modifications with a single application of the algorithm. Fourth, most test selection algorithms require prior knowledge of code modifications; that is, they require either a list of code changes, or a mapping of statements in the old program to statements in the new program. Our algorithm, however, requires no such prior knowledge; instead we perform analysis to locate code changes only as needed. Finally, our approach can be automated, and is applicable to all programs for which program dependence graphs can be constructed.

In the next section, we provide background information needed for the rest of the paper. Section 3 details our algorithm applied to individual procedures. In section 4, we show how our approach works interprocedurally. We discuss related work in section 6 and present conclusions in section 7.

2 Background

In this section, we briefly present the regression testing problem. Then, we overview the program dependence graph, the program representation that our algorithms use. Finally, we discuss the test execution information that our algorithms require.

Regression Testing

Given a program $P$, a modified version $P'$, and a test set $T$ used previously to test $P$, regression testing techniques attempt to make use of $T$ to gain sufficient confidence in the correctness of $P'$. These techniques usually consist of the following steps:

1. Identify the modifications that were made to $P$.
2. Select $T' \subseteq T$, the set of tests to reexecute on $P'$.
3. Retest $P'$ with $T'$, establishing $P'$'s correctness with respect to $T'$.
4. If necessary to satisfy some coverage criteria, create new tests for $P'$.
5. Create $T''$, a new test set for $P'$, and gather test execution information for $T''$.

In previous work[21], we described a framework for evaluating regression testing techniques. Using this framework, we classify a regression testing technique as safe if it selects all tests from $T$ that could possibly exhibit different output when run on $P'$. However, a safe approach may include many tests in $T'$ that cannot possibly exhibit different output in $P'$. Thus, we further classify a safe regression testing technique as precise if it avoids choosing tests that will not cause $P'$ to produce different output than $P$. Our framework also categorizes regression testing techniques in terms of their support for coverage criteria.

Program Dependence Graphs

A program dependence graph[5] (PDG) represents both control dependence and data dependence in a single graph. For statements $X$ and $Y$ in a program, if $X$ is control dependent on $Y$ then there must be at least two paths out of $Y$, where one path always causes $X$ to be executed and the other path may result in $X$ not being executed. A data dependence exists between statements $X$ and $Y$ in a program if $X$ defines a variable $v$, $Y$ uses $v$, and there is a path from $X$ to $Y$ in the program on which $v$ is not defined.5

Figure 1 shows procedure sqrt on the left and its partial PDG on the right (we omit some data dependence edges for readability); node labels in the PDG correspond to statement numbers in the procedure. For our discussion, we assume that a program has a single entry and a single exit. This implies that all procedures return to their callers; thus, we treat call sites as simple statements. Although we do not discuss it in this paper, with a simple modification in the construction of a PDG, we accommodate the possibility that procedures may exit before returning to their callers.

A PDG contains several types of nodes. Statement nodes, shown as ellipses in Figure 1, represent simple program statements such as assignment, input/output, or call statements. Circles represent region nodes, which summarize the control dependence conditions necessary to reach statements in the region. Predicate nodes, from which two edges may originate, are represented as squares. A hexagon represents the exit from the procedure.

Control dependencies are represented explicitly in the PDG by control dependence edges, shown as solid lines in Figure 1. For any control dependence edge $(A,B)$ in a PDG, $A$ is a control dependence predecessor (cd-predecessor) of $B$, and $B$ is a control dependence
successor (cd-successor) of A. Nodes in a PDG that have no cd-successors are leaf nodes; nodes with cd-successors are interior nodes. In Figure 1, S7 and S8 are both cd-successors of R3, and R3 is the cd-predecessor of both S7 and S8. Furthermore, S5 and S6 are leaf nodes and R3 is an interior node. Statements that have the same control dependencies are grouped together into the same region in a PDG. For example, statements S7 and S8 are both control dependent on P6-T. Without region R3, there would be two edges labeled “T” originating at P6.

Data dependencies between statements in a program are represented in a PDG by data dependence edges, shown in Figure 1 as dashed lines. For any data dependence edge \((A,B)\) in a PDG, \(A\) is a data dependence predecessor (dd-predecessor) of \(B\), and \(B\) is a data dependence successor (dd-successor) of \(A\). In the PDG in Figure 1, there are data dependence edges from S2, where \(eps\) is defined, to both S9 and P11, where \(eps\) is used. Thus, S9 and P11 are both dd-successors of S2, and S2 is a dd-predecessor of both S9 and P11. The graph shown in Figure 1 is a partial PDG since we omit the remainder of the data dependence edges for \(\text{sqrt}\).

Test Execution Information

In order to perform regression testing using an existing test suite, we must have access to test history information, which keeps data on previous executions of tests. For our purposes, test history information includes a region trace: a list of the predicate statements executed, and regions of code entered, by each test. Using the region trace, we associate, with each predicate and region node in the PDG, the set of tests in \(T\) that executed the predicate statement or entered the region of code during execution of the original program. We call this set a region history. For a predicate or region node \(R\), we use \(R\text{.history}\) to refer to \(R\)’s region history. Note that we can easily generate region traces from statement traces if necessary, without rerunning all tests on the original program.

Figure 1 contains a table that describes the test history information for procedure \(\text{sqrt}\). The table shows the region trace and input for each test. In the PDG in the figure, R3 has only T2 associated with it while P3 has all tests in the test suite associated with it. Thus, \(R3\text{.history} = \{T2\}\) and \(P3\text{.history} = \{T1,T2,T3,T4,T5\}\).

3 Intraprocedural Regression Testing

In Figure 2, we present procedure \texttt{RegTest}, the main procedure of our algorithm for selecting tests and identifying coverage requirements for intraprocedural regression testing. Given procedures \(P\) and \(P’\), and test set \(T\), \texttt{RegTest} first calls \texttt{ConstructPDG} to construct PDGs, \(G\) and \(G’\), for \(P\) and \(P’\), respectively. Next, \texttt{RegTest} calls \texttt{SelectTests}, which selects all tests in \(T\) that may cause \(P’\) to produce dif-
different output than $P$ and returns these tests in $T'$. SelectTests also returns Correspondence, which contains information about the correspondence between nodes in $G$ and $G'$ (described more fully in section 3.1). Finally, RegTest calls CoverageReqs, which uses the PDGs and Correspondence to determine components of $P'$ that should be covered by tests, and returns these components in $CL$. We discuss SelectTests and CoverageReqs in the next two sections. To use the information returned by RegTest in a regression testing effort, we first run the tests in $T'$, and verify that $P'$ runs correctly on those tests, and then if necessary we generate additional tests to ensure coverage of components in $CL$ not exercised by tests in $T'$.

In section 3.1, we show that SelectTests runs in time polynomial in the number of nodes in the PDG, the degree of the polynomial varying with the cost of the method used to calculate Correspondence. In section 3.2, we show that CoverageReqs runs in time quadratic in the size of the PDG. PDG construction algorithms are at worst quadratic in the number of statements in a program[5]. Since the size of the PDG is at worst quadratic in the number of statements in the program, RegTest runs in polynomial time in the size of the largest of the input procedures.

3.1 Selecting Regression Tests

To understand the basic principles underlying our test selection algorithm, suppose we are given procedure sqrt as shown in Figure 1, and a new version of that procedure, sqrt2, in which S14 is modified (erroneously) to $x1 = x3$. Figure 3 depicts the PDG for sqrt2. Let $G$ denote the PDG for sqrt (shown in Figure 1), and let $G'$ denote the PDG for sqrt2. For ease of reference, we use “primes” to distinguish nodes in $G'$ from nodes in $G$. PDGs $G$ and $G'$ are identical with exception of the nodes for statement S14 and its modified version, S14’ (shown inside a dotted box in Figure 3). Assume that there are no other changes to sqrt. To retest sqrt2, we must identify the tests in $T$ that will execute S14’; these are the only tests in $T$ that may cause sqrt2 to produce different output than sqrt as the result of code changes, and possibly expose errors. The only such tests are those that satisfy the control conditions for reaching S14’; these are just the tests that reach R6’, the region summarizing those control conditions. Since the control conditions summarized by R6’ are the same as those summarized by R6 in sqrt, the only tests that reach R6’ are those that reach R6. Thus in this case, a safe test selection method need only select tests in $T$ that reach R6; i.e., T2, T3, and T4.

Now suppose sqrt2 contains multiple code changes. In this case, it is still true that the only tests in $T$ that can execute S14’ are those that reach R6’ in $G'$. However, because there are other changes in sqrt2, we can no longer conclude that the tests reaching R6’ are exactly those that reached R6: some of those tests may take different execution paths due to other code changes. For example, if S14 is modified as detailed in the previous paragraph, and the conditional in line P13 is (erroneously) reversed, test T4 will not reach R6’, and tests T2, T3, and T5 will reach R6’. Thus, selecting the tests that reached R6 in G will not suffice. However, we can correctly conclude the following: the only tests that may now reach R6’ are those that either reached R6 in $G$, or that now reach some other changed code in sqrt2. It follows that in the presence of multiple changes, a safe way to select all tests that may execute changed code in sqrt2 is to find all tests that reach a region $R$ in $G$ such that $R$ corresponds to a region $R'$ in $G'$, and the cd-successors of $R$ and $R'$ differ. In the case of the example presented in this paragraph, this implies selecting all tests in R5.history and all tests in R6.history.

A naive method for selecting tests based on the preceding observations requires a complete mapping of region and predicate nodes in $G$ to region and predicate nodes in $G'$, and information on which region and predicate nodes in $G'$ enclosed changed nodes. Given such a mapping, the naive method simply selects all tests associated with region or predicate nodes that enclose changed nodes. However, such a mapping may be difficult and costly to obtain. We can improve on the
procedure \(\text{SelectTests}(G, G', T, \text{Correspondence}) : T'\)
\begin{align*}
\text{input} & \quad G, G' : \text{PDGs for } P \text{ and } P' \\
& \quad T : \text{a test set used previously to test } P \\
\text{output} & \quad T' : \text{the subset of } T \text{ selected for reuse} \\
\text{declare} & \quad \text{GetCorresp}(N, N') : \text{returns true if traversal is to continue in a region; false otherwise}
\end{align*}

1. begin SelectTests
2. foreach node \(n\) in \(G\) and \(G'\) do mark \(n\) “not visited”
3. \(T' \leftarrow \text{Compare}(E, E')\), where \(E, E'\) are entry nodes of \(G, G'\), respectively
4. end SelectTests
5. 
6. procedure \(\text{Compare}(N, N') : T'\)
7. begin Compare
8. Mark \(N\) and \(N'\) “visited”
9. if GetCorresp \((N, N')\) then
10. \(T' \leftarrow \emptyset\)
11. foreach new or modified cd-successor \(n\) of \(N'\) (or deleted cd-successor \(n\) of \(N\)) do
12. foreach dd-successor \(u\) of \(n\) in \(G'\) \((G)\) do
13. if \(U\) is “visited” then
14. Add to \(T'\) all tests in \(N\.\text{history}\) that are also in the \(U\.\text{history}\) of \(U'\)'s cd-predecessor
15. else mark \(U\) “affected” and attach \(N\.\text{history}\) to \(U\)'s cd-predecessor
16. if there are “affected” uses in any predicate, output, or control transfer statement control dependent on \(N\) or \(N'\) then
17. \(T' \leftarrow N\.\text{history}\)
18. else
19. foreach cd-successor \(n\) of \(N\) and \(N'\) that is a leaf node do mark \(n\) “visited”
20. foreach cd-successor \(C\) of \(N\) or \(N'\) marked “affected” do
21. Add to \(T'\) all tests in \(N\.\text{history}\) that are attached to \(C\)
22. foreach cd-successor \(C\) of \(N\) that is an interior node and is not “visited” do
23. \(T' \leftarrow T' \cup \text{Compare}(C, C')\), where \(C'\) is the corresponding cd-successor of \(N'\)
24. else \(T' \leftarrow N\.\text{history}\) /* processing cannot continue in \(N\) and \(N'\) */
25. return \(T'\)
26. end Compare

Figure 4: Procedure for selecting tests to rerun.

A naive method by instead traversing \(G\) and \(G'\) in preorder and, on reaching a region or predicate node whose cd-successors have changed, selecting all tests that reach that node. Having selected these tests, we need not proceed farther in our traversal; all tests reaching nodes farther in the traversal through this chain of control dependence edges have necessarily been selected. Thus, in the case of the example where \(S14\) and \(P13\) are modified, we traverse \(G\) and \(G'\) until we reach \(R5\), where we note the difference between \(P13\) and \(P13'\). At that point we select all tests in \(R5\.\text{history}\), and traverse no farther through nodes control dependent on \(R5\). We present a safe test-selection algorithm based on this approach in [20].

Unfortunately, the algorithm of [20] may be imprecise, admitting many tests that cannot exhibit different output in \(P\) and \(P'\). For example, if we modify line \(S2\) of procedure \(\sqrt{\text{p}}\) such that \(<\text{eps}\>\) is initialized to a different value, the algorithm of [20] selects all tests in \(T\), because all tests enter the region \((E)\) that encloses \(S2\). However, test \(T1\) never reaches a use of \(<\text{eps}\>\), and cannot cause the modified procedure to produce different output; thus \(T1\) need not be selected. Since we expect changes in initialization and assignment statements to occur somewhat frequently, this source of imprecision may have far-reaching effects. The test selection algorithm presented in this paper addresses this difficulty, and identifies a more precise \(T'\), by providing a means for excluding tests that execute changed definition statements, but do not reach uses of the changed definitions. Essentially, our new algorithm continues to use control dependence information to ensure selection of safe test sets, while relying on data dependence information to improve precision in test selection.

Figure 4 presents our algorithm for selecting regression tests. The algorithm is invoked by calling procedure \(\text{SelectTests}\) with PDGs \(G\) and \(G'\), and test set \(T\). \(\text{SelectTests}\) uses recursive procedure \(\text{Compare}\) to perform synchronous depth-first traversals of \(G\) and \(G'\), relying on a “visited” flag attached to each node to avoid revisiting nodes during the traversals. \(\text{SelectTests}\) first marks all PDG nodes “not visited”, and then initiates the graph traversal by calling \(\text{Compare}\) with the entry nodes of \(G\) and \(G'\).

\(\text{Compare}\) selects tests for a given pair of region or predicate nodes in \(G\) and \(G'\). Called with a pair of region or predicate nodes \(N\) and \(N'\), \(\text{Compare}\) first marks them “visited”, and then calls Boolean function \(\text{GetCorresp}(N, N')\). Given any such pair of nodes \(N\) and \(N'\), either their cd-successors have changed “sufficiently” to require rerunning of all tests reaching \(N\), or they have not. In the former case we select all tests through \(N\); in the latter case we continue the graph traversal through cd-successors of \(N\) and \(N'\).
procedure GetCorresp(R, R\'):boolean
input R, R\': region or predicate nodes in PDGs G and G'
output true if traversal is to continue beneath R and R\', false otherwise

1. begin
2. Attempt to match cd-successors of N and N', locating new, deleted, and modified nodes
3. if we cannot match cd-successors
4. return false
5. else
6. Record information on node correspondence, and on new, deleted, and modified nodes in Correspondence
7. if nodes associated with predicate, output, or control transfer statements are new, modified, or deleted then
8. return false
9. else if a cd-successor of N or N' is marked affected
10. return false
11. else
12. return true
13. end

Figure 5: The GetCorresp procedure.

The GetCorresp function has primary responsibility for interpreting the word “sufficiently”, and determining whether traversals should continue beneath N and N'. Essentially, this approach obviates the need for prior knowledge of modifications, instead locating changed code as it traverses the graph, and only as needed.

To understand Compare and GetCorresp more fully, it helps to consider once again the simpler versions of these routines, presented in [20], that we described above. In the simpler version of Compare, lines 11 through 21 of Figure 4 (all marked by asterisks) are omitted. The simpler version of GetCorresp returns true if any cd-successors of N' are new or modified, or if any cd-successors of N have been deleted; GetCorresp returns false otherwise. Given these simpler versions of the routines, when GetCorresp returns false, it signals that all tests that entered N may, on entering N', execute code that differs from that executed under N, and thereby produce different output. In this case, the predicate at line 9 of Compare evaluates to false, and line 24 is executed, selecting all tests through N'. Traversal through cd-successors of N and N' is not, in this case, required, because all tests reaching nodes beneath N and N' via the chain of control dependencies summarized by N have now been selected.

If, on the other hand, GetCorresp finds all nodes under N and N' equivalent, it returns true, signaling that the nodes immediately control dependent on N and N' will not cause tests to produce different output: in this case graph traversals must continue through cd-successors of N and N' not yet visited, as effected by lines 22 and 23 of Compare.

As mentioned above, the problem with the simpler versions of Compare and GetCorresp presented in [20] is that they select unnecessarily imprecise test sets. If the only code changed beneath N or N' is a definition statement, tests reaching N will only cause P' to produce different output if they also reach uses of the defined variable. The simpler versions of Compare and GetCorresp select all tests through N' whether or not those tests reach uses of the defined variable. We improve precision, then, by recording where a change merely involves a variable definition and when it does, marking the uses reached from this definition as “affected”. Instead of selecting all tests through such a definition, we select only the tests that reach both the definition and some use of the definition. This approach is complicated by the presence of multiple changes, because when P' contains multiple changes they may interact, and we cannot determine precisely which tests will reach which uses. However we can safely identify a superset of the tests that will reach uses, and select these, by adding lines 11 through 21 to Compare, and invoking a more intelligent version of GetCorresp.

Figure 5 presents a version of GetCorresp that achieves greater precision in test selection. This new version of GetCorresp attempts to establish a “mapping” between the cd-successors of N and N', recording that mapping in Correspondence (line 2). Correspondence is a pair of arrays that track each node in G and G'. Initially these arrays list all nodes as “not examined”. As GetCorresp determines mappings between nodes in G and G' it updates the arrays, recording nodes that are in P but not P' as “deleted”, nodes that are in P' but not P as “new”, nodes that exist in both P and P' but differ as “modified”, and nodes that are the same in both P and P' as “equivalent”. In the case of “modified” nodes, Correspondence also tracks which node in P corresponds to which node in P'.

If GetCorresp cannot establish a mapping, it returns false: all tests through P must be selected. If, however, GetCorresp can establish a mapping, it examines the mapped nodes. If nodes representing predicate, output, or control transfer statements are new, modified, or deleted, or if nodes are already marked “affected”, then GetCorresp returns false, indicating that all tests through N must be selected (lines 7-10). However, if neither of these conditions holds, GetCorresp returns true, enabling Compare to further consider the changes, and possibly avoid selecting all tests through N. GetCorresp can attempt to establish a mapping between cd-successors of N and N' in several ways. For example, GetCorresp can use a simple text differencing utility that compares the text string derived
from concatenating the code corresponding to the cd-succesors of $N$ to that derived from concatenating the code corresponding to the cd-succesors of $N'$. Alternatively, GetCorresp can use a method such as the SequenceMatching algorithm presented in [10] and [24]. If the mapping method is conservative and avoids “false positives” that identify changed components as unchanged, Compare will select safe test sets. More precise methods for establishing mapping yield more precise test selection, but at increased execution cost.

When GetCorresp returns true, either there are no changes beneath $N$ and $N'$, or the changes are not sufficient to force selection of all tests through $N$. In the latter case, Compare (line 11) considers each new or modified cd-successor $n$ of $N'$, and each deleted cd-successor $n$ of $N$. If $n$ contains a variable definition, Compare uses data dependence edges in $N$ and $N'$ to find nodes $U$ containing uses reached from $n$ (line 12). Some of these nodes may have already been visited in the traversal, and are marked “visited”. For any such $U$ marked “visited”, Compare selects the tests in $N.history \cap C.history$, where $C$ is the cd-predecessor of $U$ (lines 13-14), because all such tests exercise a changed definition and may reach the use at node $U$. If $U$ is not visited”, Compare marks $U$ “affected” and attaches the tests in $N.history$ to $C$ (line 15): these tests will either be selected when $C$ is visited, or selected due to their association with some region on which $C$ is control dependent. Note that whether $U$ is “visited” or not, if $P$ contains multiple changes, some of the tests in $N.history \cap C.history$ may no longer reach $n$ and/or $U$, but we cannot easily detect this, so we select all such tests to ensure safety. Note further that with multiple changes, some tests may now reach $n$ and $U$ that did not so originally; however these tests must first have executed a changed or affected predicate statement and would be selected by Compare in connection with that other changed statement.

At this point in Compare (after execution of lines 11-15), it is possible that due to a change in a definition in a cd-successor of $N$ or $N'$, a use that is also a cd-successor of $N$ or $N'$ is now marked “affected”. If Compare finds any such uses in nodes that are output, predicate, or control transfer statements, it returns tests in $N.history$ (lines 16-17). If there are no such “affected” uses, Compare marks “visited” all cd-succesors of $N$ and $N'$ that are leaf nodes. Now, if $N$ or $N'$ have “affected” cd-succesors, these nodes contain uses of variables reached by new or modified definitions, or reached by definitions in $P$ that are now deleted, and the tests that reached both the definitions and these uses must be selected. Thus, Compare selects all tests in $N.history$ that were attached to affected cd-succesors of $N$ and $N'$ that are interior nodes.

We now consider an example that illustrates the use of SelectTests. Figure 6 presents the code for procedure sqrt3 (a modified version of the sqrt procedure shown in Figure 1) and a partial PDG for sqrt3; changes to both sqrt and its partial PDG are shown in dotted boxes. In sqrt3 line $S8a$ is new, the conditional in line $P13$ has been (erroneously) changed, and the assignment in $S15$ has been (erroneously) changed. For simplicity, we show only one of the data dependence edges in the PDG.

When called with the PDGs of Figures 1 and 6, SelectTests first calls Compare with $E$ and $E'$. Compare marks $E$ and $E'$ “visited” and then calls GetCorresp. GetCorresp finds that all cd-succesors of $E$ and $E'$ are equivalent, and returns true, causing Compare to proceed to line 10. Compare finds no new, modified or deleted cd-succesors of $E$ or $E'$, and proceeds to line 16. Compare finds no affected uses in the cd-succesors of $E$ or $E'$, and thus calls itself on $P3$ and $P3'$ (line 23). This call leads to calls to Compare with $R1$ and $R1'$, then $R2$ and $R2'$, then $P6$ and $P6'$, with no differences detected between the programs, and no tests selected. Next, Compare invokes itself on $R3$ and $R3'$. GetCorresp determines that node pairs $(S7, S7')$ and $(S8, S8')$ are equivalent, and that $S8a$ is new. Since $S8a$ does not involve a predicate, output, or control transfer statement, and since there are no “affected” uses under $R3$ and $R3'$, GetCorresp returns true. Compare reaches line 11, and finds that $S8a$ is a new cd-successor of $R3'$, so Compare examines data dependence edges originating at $S8a$ to find reachable uses of $z3$ in sqrt3, and finds one in $S16'$ that is marked “visited”. The only test in both $R2.history$ and $R3.history$ is $T2$; this is a test that may both reach the changed definition in $S8a$ and reach the use of that definition in $S16'$, so Compare adds $T2$ to $T'$. Next, Compare continues its graph traversals and finds that cd-succesors of the pairs $(R4, R4')$ and $(P11, P11')$ are equivalent. When GetCorresp considers $R5$ and $R5'$, it finds that cd-successor $P13$ of $R5$ has been modified. Since $P13$ is a predicate, GetCorresp returns false, which causes Compare to select all tests in $R5.history$; i.e., $T2, ..., T5$. Compare does not further process the subgraph rooted at $R5$, because all tests executing statements in that subgraph have control dependencies summarized by $R5$, and have now been selected and added to $T'$.

Our example does not illustrate the use of the “affected” flag, because in the example, when Compare considers the use of $z3$ in $S16$, $S16$ has already been visited. If $S16$ had not already been visited, Compare would have marked $S16$ “affected”, and attached the tests in $R3.history$ to $S16$. Later in the traversal, if Compare reaches the cd-predecessor of $S16$, lines 20 and 21 ensure selection of tests. The “affected” flag thus supports selection of tests through uses not yet reached in the traversal.

With a small modification, SelectTests handles changes in variable, type, and constant declarations. We assume that each declaration or type definition used in or applicable to $P$ ($P'$) is recorded with the entry node in $G (G')$. When Compare visits PDG entry nodes, it determines the correspondences between such definitions.

Note that in this case, test $T2$ cannot reveal an error in the definition in $S8a$ because $T2$ executes another definition of $z3$ before it executes the use of $z3$ in $S16$. However, Compare would also select other tests not executing the intervening definition if any existed; in this case precision is sacrificed for the sake of safety. Imprecision of this sort can be eliminated if test history information tracks the definition and use associations satisfied by tests in that case we would not choose test $T2$ here.
procedure sqrt2 (real x): real
real x1, x2, x3, eps, errval
begin
S1 errval = 0.0
S2 eps = .001
P3 if (x <= 0.0) then
S4 output("illegal operand")
S5 return errval
else
P6 if (x < 1) then
S7 x1 = x
S8 x2 = x
S9 x1 = eps
S10 x2 = x
endif
P11 while (abs(x2-x1) >= 2.0*eps) do
S12 x3 = (x1+x2)/2.0
P13 if ((x1*x1-x1)*(x1*x1-x1) > 0) then
S14 x2 = x3
else
S15 x1 = x2
endif
endwhile
return x3
end

Figure 6: Procedure sqrt3 (a modified version of the sqrt procedure) and its partial PDG.

For any new, modified, or deleted definitions, uses of the variables defined are marked “affected” in G and G', just as uses reached from changed definition statements are marked. We also assume that “non-executable” initialization statements and constants are rendered as co-suc-cessors of entry nodes in the PDGs, and handled just like executable definitions.

SelectTests’s running time depends upon the algorithm used by GetCorresp; suitable lexical matching algorithms have bounds ranging from linear to quadratic in the number of nodes to be matched. The number of calls to GetCorresp is bounded by the lesser of the number of nodes in G and G'. Worst-case behavior of Compare depends on GetCorresp returning true, in which case Compare enters the then clause at line 10. At worst, GetCorresp returns true for every pair of nodes considered, causing lines 10-23 to be executed for every GetCorresp (and hence for every Compare) call. Lines 10-23 examine, at worst, each control dependence edge and each data dependence edge in G and G', however, once an edge e has been examined by these lines, it is never examined again on this or any subsequent entry to the then clause. Thus lines 10-23 do work, at worst, proportional to the number of edges in G and G', regardless of the number of times Compare is invoked. It follows that for PDGs of at most n nodes and n^2 edges, the upper bound on the running time of SelectTests ranges from O(n^3) to O(n^6), depending on the algorithm used by GetCorresp.

3.2 Identifying Test Coverage Requirements

SelectTests selects all tests in T that may produce different output in P' and P. However, after these tests have been run, there may still be parts of P that behave differently due to code modifications, and have not been tested. In order to measure test adequacy and ensure that appropriate program components are tested, many regression testing techniques use coverage criteria. However, by focusing on coverage such approaches often fail to select tests that may cause P' to produce different output than P. Moreover, in the presence of multiple or structural changes, these approaches often cannot determine the tests that cover a particular component without considering each code modification separately, and reanalyzing the program after each modification has been considered.

In contrast, our approach first selects a safe test set T*. We then use Correspondence to find definition-use pairs (du-pairs) that should be covered, to support dataflow testing criteria[18]. Using this approach, we need not consider each modification individually. Instead multiple modifications are handled in a single pass over the graphs. Moreover, by reusing the Correspondence structure constructed by SelectTests, we obviate the need to recompute correspondence information. Since our approach identifies

7 A definition-use pair is a pair of statements (S1, S2), such that S1 defines some variable v, S2 uses v, and there is a path in the program from S1 to S2 along which v is not redefined.
traversed by a forward slice from $S_8\alpha$.

We must also detect type 2 edges as we traverse $G'$. However, we cannot detect these edges precisely. To understand why, consider the $\texttt{sqrt3}$ program and its PDG depicted in Figure 7. Statement $P13'$ is data dependent on statement $S7'$, and thus the PDG for $\texttt{sqrt3}$ contains data edge $E1 = (S7', P13')$. Since node $S7'$ is equivalent to node $S7$, $E1$ is not a type 1 edge. Furthermore, since $S7'$ is not control or data dependent on any other new or modified node in the PDG, $E1$ is not a type 3 edge. However, since $P13'$ is modified, $E1$ is a type 2 edge. We can recognize $E1$ as a type 2 edge because we know that $P13'$ is modified. In contrast, consider data dependence edge $E2 = (S10, S15')$. Like $E1$, $E2$ is a type 2 edge, because $S15'$ is modified, but $E2$ is not a type 1 or type 3 edge. Given the results of SelectTests, however, we can not precisely identify $E2$ as a type 2 edge, because SelectTests never visits node $S15'$, but rather, ceases its traversal of the PDG upon reaching node $P13'$. To select type 2 edges, given the information provided by SelectTests, we must select all edges $(N1', N2')$ in $G'$ where (2a) $N2'$ is a new node, (2b) $N2'$ has a corresponding node $N2$ in $G$ that differs from $N2'$, or (2c) $N2'$ is flagged as “not examined”. Case (2c) is the source of the imprecision in our edge selection, but ensures selection of a safe superset of the type 2 edges.

To detect affected edges, algorithm RegTest calls CoverageTests, shown in Figure 8, which inputs $G$, $G'$, and Correspondence, and returns $\texttt{AdjDUPairSet}$, the set of du-pairs to cover. CoverageTests uses FindAffected and a “visited” flag to effect its traversal of $G'$. CoverageTests initially marks all nodes in $G'$ “not visited”, and then initiates the traversal of $G'$ by calling procedure FindAffected with entry node $E'$. FindAffected returns a set of data dependence edges $EL$, which is used to construct $\texttt{AdjDUPairSet}$.

FindAffected is called with a node $N'$ of $G'$. If $N'$ does not have an equivalent node in $G$ (line 10), then FindAffected performs a forward slice from $N'$, selecting all data dependence edges traversed by that slice and thereby selecting all edges of types 1 and 3 involving $N'$ (lines 11-13). If $N'$ has an equivalent node $N$ in $G$, then FindAffected considers each edge $(N', M')$ in $G'$, seeking type 2 edges (lines 14-19). If there is no node $M$ in $G$ corresponding to $M'$, or $M$ corresponds to $M'$ but $(N, M)$ does not exist, or $M$ corresponds to $M'$ but is not equivalent to $M'$, FindAffected selects $(N', M')$. After considering each edge, FindAffected invokes itself on all cd-successors of $N'$ (line 20).

To avoid computing a complete forward slice for each change, FindAffected uses information gathered during its traversal of the PDG in two ways. First, since no additional information is gathered on subsequent slices from a node $N$, when FindAffected slices forward it marks as “visited” each node it reaches (line 13). Whenever FindAffected reaches a “visited” node during this, or any subsequent slice, it does not continue the slice beyond that node (line 13). Second, if FindAffected finds a change under $N'$ and slices forward from $N'$, that slice necessarily reaches all cd-successors of $N'$, so there is no need to traverse the PDG further under $N'$. Thus, FindAffected only in-

---

Figure 7: A partial PDG for $\texttt{sqrt3}$, a modified version of $\texttt{sqrt}$.
procedure CoverageReqs \((G,G',\text{Correspondence})\) : AffDU Pair Set
input \(G, G' : \) PDGs for \(P\) and \(P'\), respectively
Correspondence : a correspondence of nodes in \(G\) to nodes in \(G'\)
output AffDU Pair Set : set of affected definition-use pairs in \(G'\)
1. begin CoverageReqs
2. For each node \(N'\) in \(G'\) do mark \(N'\) “not visited”
3. \(EL = \text{FindAffected}(E')\) /* For \(E'\) entry node of \(G'\) */
4. For each edge \((U,V)\) in \(EL\) do add definition-use pair \((U,V)\) to AffDU Pair Set
5. return AffDU Pair Set
6. end CoverageReqs

7. procedure FindAffected\((N')\) : \(EL\)
8. begin FindAffected
9. \(EL = \emptyset\)
10. if \(N'\) has no equivalent node \(N\) in \(G\) then
11. For each data dependence edge \((U',V')\) traversed during a forward slice on \(N'\) do
12. \(EL = EL \cup (U',V')\)
13. else if \(V'\) is “not visited” then mark \(V'\) “visited” and continue forward slice with \(V'\)
14. else /* \(N'\) has an equivalent node in \(G'\) */
15. For each data dependence edge \((N',M')\) do
16. if \(M'\) has no corresponding node in \(G\) then \(EL = EL \cup (N',M')\)
17. else
18. Let \(M\) be the node corresponding to \(M'\) in \(G\)
19. if there is no edge \((N,M)\) in \(G\), or \(M\) differs from \(M'\), then \(EL = EL \cup (N',M')\)
20. For each cd-successor \(C\) of \(N'\) “not visited” do \(EL = EL \cup \text{FindAffected}(C)\)
21. return \(EL\)
22. end

Figure 8: Procedure for identifying definition-use pairs to cover.

vokes itself on cd-successors of \(N'\) at line 20, in the else clause, where \(N'\) and \(N\) have been found equivalent.

To see how FindAffected works, consider sqrt3, and its partial PDG depicted in Figure 7. Let \(G\) be the PDG for sqrt, and \(G'\) be the PDG for sqrt3. Suppose that SelectTests determines that each node \(N'\) in \(G'\) is “equivalent” to node \(N\) in \(G\), except for nodes \(S8a, P13', R6', R7', S14', \) and \(S15'\). Furthermore, suppose \(S8a\) is identified as “new”, and \(P13'\) is flagged as “modified”. Nodes \(R6', R7', S14', \) and \(S15'\) remain “not examined”. This information is passed in Correspondence to CoverageReqs.

CoverageReqs marks all nodes in \(G'\) “not visited”, and then calls FindAffected with \(E'\). Since \(E'\) corresponds to \(E\), FindAffected finds the else clause at line 14, and examines edges originating at \(E'\). When considering edge \((E',P13')\), FindAffected discovers that \(P13'\) has corresponding node \(P13\) in \(G\), and that edge \((E,P13)\) exists in \(G\), but that \(P13'\) is modified; thus edge \((E',P13)\) is of type 2, so FindAffected adds \((E',P13')\) to edge list \(EL\). Other data dependence edges from \(E'\) \((E',P6')\) and \((E',S7')\), not shown) are not selected. FindAffected next visits (in turn) nodes \(S1', S2', P3', R1', S4', S5', P6', R3', S7', \) and \(S8'\), and selects type 2 edge \((S7',P13')\). It then visits node \(S8a\), which has no corresponding node in \(G\). Thus FindAffected enters the then clause at line 11. FindAffected slices forward from \(S8a\); this slice visits (and marks “visited”) nodes \(S12', S16', X', P13', R6', S14', P11', R3', R7', \) and \(S15'\), and adds data dependence edges \((S8a,S12'), (S8a,S16'), (S12',S12'), (S12',S16'), (S14',P11'), (S15',S12'), (S15',P11'), (S12',S14'), \) and \((S12',S15')\) to \(EL\). When the slice is complete, FindAffected continues visiting nodes, adding type 2 edge \((S9',P13')\) to \(EL\), Since all cd-successors of \(R5'\) are marked “visited”, FindAffected returns. A total of thirteen data dependence edges are returned to CoverageReqs, which translates them into definition-use pairs, and returns them to RegTest.

CoverageReqs, due to its method of marking nodes, visits each PDG edge at most twice; once during the then clause of lines 11-13, and once during the else clause at lines 15-19. CoverageReqs also visits each node at most once during its walk of the PDG, and once for every edge that enters the node. Thus, CoverageReqs runs in time linear in the size of the PDG.

4 Interprocedural Regression Testing

Our intraprocedural regression testing algorithm both reduces the number of tests we must rerun and identifies the du-pairs we must cover to retest a modified program. However, selective retest algorithms are cost effective only if the time we save by omitting tests exceeds the time we spend finding tests we can omit[16]. For intraprocedural regression testing, it may often be more cost effective to rerun all tests, and cover all du-pairs in the modified procedure.

When we consider interprocedural testing this objection is mitigated. Interprocedural testing addresses
an entire program, or a group of procedures. When we maintain large programs, code modifications are often restricted to a small subset of the program's procedures. In this situation the savings resulting from careful test selection and identification of a reduced set of affected program components can outweigh the costs of selection and identification.

To apply RegTest interprocedurally, we use a graph based on the system dependence graph (SDG)[12]. We build an SDG by constructing PDGs for each procedure in a program, and adding special nodes and edges that represent parameters, global variables, and interprocedural data dependencies. For illustration, see Figure 9, where we present a partial SDG for a postfix calculator program, calculator. The figure gives the code for the main procedure of calculator, and the portion of the SDG for the program that corresponds to the main procedure and the sqrt procedure.

The nodes in an SDG are the same as those in a PDG, with the exception of parameter nodes. Parameter nodes are placed beneath each call and entry node; each formal parameter represented by "formal in" and "formal out" nodes, and each actual parameter represented by "actual in" and "actual out" nodes. Function return values are similar, but they require only the two "out" nodes. Global variables are treated like parameters; nodes similar to those for parameters are created at each call site for each global variable.

The edges in our SDG are as follows (refer to Figure 9 for illustration):

1. **Control dependence edges** are the same in SDGs as in PDGs.
2. **Intraprocedural data dependence edges** are the same in SDGs as in PDGs, except that since the definitions and uses of variables that were assumed at call, entry, and exit sites are now rendered as individual parameter nodes, edges formerly connected to call, entry and exit sites are now connected to parameter nodes instead.
3. **Parameter control dependence edges** depict the control dependence of parameter nodes on call or entry nodes.
4. **Parameter-in edges** connect actual-in parameter nodes to their corresponding formal-in parameter nodes, representing the passage of data into called procedures.
5. **Parameter-out edges** connect formal-out parameter nodes to their corresponding actual-out parameter nodes, representing the return of values to calling procedures. There is a parameter-out edge from formal-out node X to actual-out node Y if and only if the value of X is passed back into Y.
6. **Call edges** connect each call node to the entry node of the called procedure.
7. **Interprocedural data dependence edges** summarize interprocedural data dependencies; these edges do not appear in the SDG of [12]. These edges are the same as intraprocedural data dependence edges, with the restriction that the endpoints of the edge must be in different procedures.

8. **Transitive dependence edges** connect actual-in parameter nodes to actual-out parameter nodes. There is a transitive interprocedural dependence edge from actual-in node X to actual-out node Y if and only if the value of the variable at Y is transitively dependent upon the value of the variable at X, by some chain of control and data dependencies occurring within the called procedures.

Horwitz, Reps, and Binkley[12] present an algorithm for performing forward slices on SDGs that operates in two passes. The first pass slices along edges of types 1, 2, 3, 5, and 8, and the second pass slices along edges of types 1, 2, 3, 4, 6, and 8. This two-pass approach is necessary if the computed slice is to preserve the calling context of called procedures; without the two-pass approach a slice can reach procedures not accessible through a legitimate sequence of procedure calls and returns.

With a few modifications, RegTest functions on SDGs, thus selecting tests and returning coverage information for entire programs. The required changes are as follows:

- In all procedures, when we refer to data dependence edges or dd-successors/predecessors, understand this to involve only edges of types 2 and 7.
- In all procedures, when we refer to control dependence edges or cd-successors/predecessors, understand this to include only edges of types 1 and 6.
- Procedures FindAffected and CoverageReqs are replaced by procedures FindAffected2 and CoverageReqs2, shown in Figure 10. Note that FindAffected2 differs from FindAffected only by addition of line 5. In CoverageReqs2, lines 12-26 replace lines 11-13 of CoverageReqs. Both pairs of procedures could be replaced by single, parameterized versions; we depict the different versions to facilitate discussion.

RegTest and SelectTests function the same on SDGs as on PDGs, except that RegTest handles entire programs and builds SDGs rather than PDGs. The operations performed by CoverageReqs2, however, differ from those performed by CoverageReqs in two ways. First, lines 13-26 of FindAffected2 support the two-pass forward slicing algorithm required on SDGs in lieu of the one-pass algorithm required for PDGs. Whereas the one-pass slice had only to mark nodes “visited” in order to avoid slicing through nodes a second time, the two-pass slice must distinguish the passes in which nodes are visited, thus each node has two flags attached: “pass1-visited” and “pass2-visited”. The first pass of the two-pass slice does not slice through nodes already marked “pass1-visited” (line 16), and the second pass does not slice through nodes already marked “pass2-visited” (line 23). As each pass encounters nodes, it sets the appropriate “visited” flag to true (lines 19 and
main()
int token
real num,num1,num2
begin
    token = gettoken(num)
while (token <> DONE)
    case token of
        NUMBER:
            push(num)
        ADD:
            num1 = pop()
            num2 = pop()
            push(num1+num2)
        MULT:
            num1 = pop()
            num2 = sqrt(num1)
            push(num2)
        endcase
    endwhile
end

Figure 9: The main procedure of calculator (lower left), the partial SDG for calculator (upper left), and modified portions of the SDG corresponding to calculator' (right).
procedure CoverageReqs2
input G, G' : PDGs for P and P', respectively
ComponentMapping : a partial mapping from nodes in G to nodes in G'
output AffDUPairSet : set of affected definition-use pairs in G'
1. begin CoverageReqs2
2. foreach node N' in G' do mark N' “not visited”
3. EL = FindAffected2(E') /* for E' entry node of G' */
4. foreach edge (U, V) in EL do add-du-pair (U, V) to AffDUPairSet
5. foreach edge (U, V) in GL where both V and U are “pass1-” or “pass2-visited” do add-du-pair (U, V) to AffDUPairSet
6. return AffDUPairSet
7. end CoverageReqs2

8. procedure FindAffected2(N') : EL
9. begin FindAffected2
10. EL = φ
11. if N' has no equivalent node N in G then
12. foreach type 7 edge (N', M') do GL = GL ∪ (N', M')
13. /* perform pass 1 of the forward slice */
14. foreach type 2 edge (U', V') traversed during pass 1 of a forward slice on N' do
15. if U' and V' are not parameter nodes then EL = EL ∪ (U', V')
16. if V' in “pass1-visited” then do not slice beyond V'
17. else if V' is not “pass2-visited” then
18. foreach edge (V', W') of type 7 do GL = GL ∪ (V', W')
19. Mark V' “pass1-visited”
20. /* perform pass 2 of the forward slice */
21. foreach type 2 edge (U', V') traversed during pass 2 of a forward slice on N' do
22. if U' and V' are not parameter nodes then EL = EL ∪ (U', V')
23. if V' in “pass2-visited” then do not slice beyond V'
24. else if V' is not “pass1-visited” then
25. foreach edge (V', W') of type 7 do GL = GL ∪ (V', W')
26. Mark V' “pass2-visited”
27. else /* N' has an equivalent node in G */
28. foreach data dependence edge (N', M') do
29. if M' has no corresponding node in G then EL = EL ∪ (N', M')
30. else
31. Let M be the node corresponding to M' in G
32. if there is no edge (N, M) in G, or M differs from M', then EL = EL ∪ (N, M')
33. foreach cd-successor C of N' “not visited” do EL = EL ∪ FindAffected2(C)
34. return EL
35. end FindAffected2

Figure 10: Procedure for computing affected du-pairs for interprocedural regression testing.

As the passes traverse type 2 edges, if those edges do not have parameter nodes as source or sink, those edges are considered “affected” and added to EL.

The second change to CoverageReqs is also motivated by differences in the forward slicing algorithm. The interprocedural slicing algorithm of [12] does not traverse interprocedural data dependence (type 7) edges. In order to ensure selection of affected interprocedural data dependence edges, FindAffected keeps track of all such edges leaving nodes visited during the forward slice, by adding them to set GL (lines 12, 18, and 25). When control returns to CoverageReqs, CoverageReqs selects every such edge (U, V) such that both U and V were visited during some forward slice (line 5). This results in selection of some edges that are not affected, but ensures that a superset of the affected interprocedural data dependence edges is selected.

We now consider an example that illustrates the way RegTest works interprocedurally. Suppose test set T for calculator contains five tests (T1', T2', T3', T4', and T5'), that exercise the sqrt function with the same values as the five tests presented in Figure 1. Suppose further that T contains a number of other tests that do not cause the sqrt module to be invoked. Now, suppose a new version of calculator, calculator2, is created by removing the definition of eps in statement S2 of sqrt, and placing global declaration eps=.0001 at the top level of the main procedure. When RegTest is invoked, it first creates the SDG (G') for calculator2. Figure 9 shows portions of G' that differ as a result of this change, and the two interprocedural data dependence edges created as a result of the change.

When SelectTests is invoked for this example, Compare finds new node S50' under E', and since S50' is a declaration, marks nodes S9' and P11' “affected,” attaching every test that entered the program to R4' and R2'. SelectTests then visits portions

9 In an actual implementation, rather than attaching tests to these nodes, we simply attach a pointer to the set of tests in E.history.
of the SDGs beneath E and E', and finds no differences under any regions until traversal of the PDG for sqrt3 begins. There, when Compare visits nodes R2 and R2', GetCorresp finds P11' “affected,” and causes Compare to select all tests attached to R2 that are also in R2.history; i.e., T2', T3', T4', and T5'. In this example, these are the only tests necessary to test the change. All tests that do not pass through sqrt are exempt from rerunning, together with those that enter sqrt but do not reach R2'; i.e., T1'. Since the program as a whole is likely to have a large number of test cases that do not enter sqrt and that are not selected, this yields substantial savings in test execution time.

Next, CoverageReqs2 is invoked, and it calls FindAffected with E. FindAffected finds no edges to select at E, and calls itself with node S50'. S50' has no corresponding node in G, so CoverageReqs2 enters the then clause at line 12, and adds interprocedural edges (S50', S9') and (S50', P11') to GL (line 12). The next step in the algorithm is the two-pass slice. Edges (S50', P11') and (S50', S9') are type 7 edges and therefore not sliced over, but edge (S50', actual eps in) is of type 2, so the slice proceeds over this edge. The first pass of the slice does not traverse parameter-in edges, and thus from the actual eps in node reaches the actual eps out node and then proceeds no farther because there are no edges out of that node. The second pass of the slice follows edges (S50', actual eps in), (actual eps in, formal eps in), and (formal eps in, S9), slicing forward from there along control and data dependence edges, and adding edges (S9, P13), (S14, S12), (S12, S16), (S16, X), (S12, P13), (S14, P13), (S15, S12), (S15, P11), (S12, S14), (S12, S15), (S9, P13), (S50, P13) to EL. Each node visited is marked “pass2-visited”. On slicing forward from formal eps in along edge (formal eps in, P11), FindAffected sees that P11 has already been “pass2-visited”, so does not proceed farther. When the slice has been completed, FindAffected continues its traversal. Note that this traversal subsequently reaches several nodes that were marked “visited” during the slice on S50; FindAffected does not invoke itself on these nodes. No other changes are found, and no other edges are added to CL. When control returns to CoverageReqs, CoverageReqs translates the edges on EL into du-pairs and adds them to AffDUPairSet, and then checks the list of potential interprocedural data dependence edges, GL. The two edges on this list are translated into du-pairs and added to AffDUPairSet. Finally, CoverageReqs returns AffDUPairSet.

The bounds on the running times of SelectTests and CoverageReqs established in relation to PDGs remain the same when these algorithms are applied to SDGs. Since the size of the SDG is polynomial in a number of parameters (comprised principally of PDG size and numbers of calls, parameters, and variables)[12], RegTest remains polynomial when applied to SDGs.

5 Related Work

In [19] and [21], we present a framework for evaluating regression test selection techniques, and a comprehensive comparison of previous techniques. In this section, we examine those techniques that are most closely related to our work.

In previous work[20], we presented a safe algorithm that uses the program dependence graph to identify those tests in an existing test suite to rerun; that algorithm handles multiple changes to the procedure. Our algorithm RegTest differs from that work in several ways. First, although our previous method was safe, it was unnecessarily imprecise, selecting all existing tests when a simple change was encountered at the top level of the program. RegTest avoids selecting unnecessary tests in that fashion; it postpones selection of a test through a simple assignment statement until a statement that uses the value computed at the assignment is reached. Second, our previous method did not identify those parts of the program that should be covered. In contrast, RegTest performs an analysis on the program to identify affected du-pairs for coverage. Finally, while the previous method offered no special assistance for test selection or identification of affected components at the interprocedural level, RegTest provides an algorithmic solution to both of these tasks.

Bates and Horwitz[3] present a coverage approach to regression testing, based on PDG adequacy criteria. Their method requires prior knowledge of code modifications, and processes these modifications one at a time to determine which statements must be covered, and which existing tests can be reused to test the modifications. In contrast, RegTest requires no prior information about program modifications, and handles multiple modifications in one application. Bates and Horwitz's method selects all tests that traverse affected statements; as we show in [21], this allows their method to select tests that cannot possibly traverse new or modified code, an imprecision RegTest avoids. Moreover, Bates and Horwitz's method is not safe, because it can omit some tests that executed statements now deleted from P; tests that may produce different output in P than in P'. Finally, their technique has only been applied to single procedures and thus, as presented, is not applicable interprocedurally.

Agrawal, Horgan, Krauser and London[1] introduce a family of slice-based regression test selection methods. Their methods require prior knowledge of program changes and process them one at a time. When restricted to non-structural changes to a program’s control flow graph, all but one of their methods are safe. However, when this restriction is relaxed, their methods all miss tests that can cause different program output, and thus are not safe. Moreover, their methods offer no assistance with coverage criteria. Finally, they describe their techniques only for single procedures. In contrast to these slice-based methods, RegTest is safe, offers assistance with coverage, is applicable interprocedurally, and does not require prior knowledge of program changes.

Laski and Szemer[14] present an algorithm that determines modified portions of a program by identify-
ing new, deleted, and modified “clusters” of statements. They recommend selecting tests that traverse these clusters. Applied to regression testing, Laski and Szemer’s method is safe, but it identifies clusters that are larger than the regions used by RegTest, and this causes their method to be less precise than ours. Laski and Szemer suggest that coverage criteria could be applied to these clusters, but offer no algorithm that would do so. Finally, Laski and Szemer’s method is only presented for single procedures, whereas our method applies interprocedurally.

6 Conclusion

In this paper, we have presented a new algorithm for regression testing that performs two important tasks. First, it selects those tests from an existing test suite that may exhibit different output in the modified program, and second, it identifies parts of the program that are affected by modifications and should be covered. The algorithm handles multiple changes, does not require prior knowledge of the location of changes, and works on both single procedures and entire programs.

We have implemented the intraprocedural version of the algorithm presented in [20]. Our experiments with that version indicate that it is efficient and easy to implement. We are currently extending the implementation to handle precision improvements, identification of test coverage requirements, and interprocedural regression testing. We also plan to implement our algorithm for C++.

Acknowledgements

We wish to thank the anonymous reviewers of this paper for their thorough readings of the paper, and their thoughtful suggestions.

References


