A Safe, Efficient Algorithm for Regression Test Selection*

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 a necessary but costly maintenance activity aimed at demonstrating that code has not been adversely affected by changes. A selective approach to regression testing selects tests for a modified program from an existing test suite. We present a new technique for selective regression testing. Our algorithm constructs control dependence graphs for program versions, and uses these graphs to determine which tests from the existing test suite may exhibit changed behavior on the new version. Unlike most previous techniques for selective retest, our algorithm selects every test from the original test suite that might expose errors in the modified program, and does this without prior knowledge of program modifications. Our algorithm handles all language constructs and program modifications, and is easily automated.

1 Introduction

It has been estimated that software maintenance activities account for as much as two-thirds of the overall cost of software production[18]. One costly but necessary maintenance activity is regression testing, which is performed on modified programs to provide confidence that modifications are correct and have not adversely affected other portions of the program. An important difference between regression testing and development testing is that during regression testing we usually have an established suite of tests available for reuse. One regression testing strategy reruns all such tests, but this retest all approach may consume inordinate time and resources. An alternative, selective retest, chooses the tests from the old test suite that are deemed necessary to test the modified program. We run these tests, and then if necessary create new ones, possibly to meet some coverage criterion. However, as Leung and White[13] observe, this selective approach is beneficial only if the cost of selecting the test subset is less than the cost of running the tests we are able to omit.

Although previous research addresses the selective retest problem, many approaches are shaped by concerns about structural coverage criteria[1, 2, 5, 7, 8, 9, 15, 19, 20, 23]. These approaches identify changed or affected program components, such as statements or paths, and select tests that traverse these components. Coverage criteria are useful because they offer a way to measure test adequacy. However, at regression test time we also wish to convince ourselves that program functionality has not been adversely affected by modifications. To accomplish this second task using selective retest, we must determine not merely which existing tests help satisfy coverage criteria, but also which existing tests may exhibit changes in program behavior. Often, this second group of potentially revealing tests forms a superset of the coverage-satisfying tests. None of the coverage-based approaches cited above successfully selects all potentially revealing tests.

Some recent approaches to selective retest place less emphasis on coverage criteria[11, 12, 21]. One drawback of these approaches is that they require prior knowledge of all changes made to the original program. This knowledge must be obtained either by a programming environment that tracks changes as they are made, or by algorithms[11, 22] that determine corresponding program components. The assumption of a programming environment that tracks changes is unreasonable for large programs given current practice; the algorithms for determining corresponding components are unnecessarily costly for this application.

In this paper, we present a new technique for selective retest that is neither coverage-criteria based nor requires complete information on corresponding program components. Our algorithm constructs graphs representing control dependence for a program and

---

*This work was partially supported by NSF under Grant CCR-9109531 to Clemson University.
its modified version, and uses these graphs to identify tests that are potentially revealing and tests that cannot possibly expose errors. Our algorithm detects portions (regions) of code that differ in the two versions of the program, and selects for retest all tests that traverse these regions.

Our technique has several advantages over previous selective retest approaches. Unlike many previous techniques, our algorithm selects tests that may now traverse new statements, and tests that formerly traversed statements that have been deleted from the original program. Thus, our algorithm selects every test from the original test suite that can possibly expose errors in the modified program. Moreover, although our algorithm selects some tests that will not exhibit different behavior, it is more precise than most existing algorithms since it does not select tests that cannot traverse changed statements. Our algorithm is also simpler and more efficient than most existing algorithms, in part because it does not require a complete mapping of the “corresponding parts” of the original and modified programs. The algorithm is not difficult to implement, and thus allows automation of a substantial portion of the selective retest process. Finally, our algorithm is more general than many previous approaches: it is not restricted to a subset of language constructs, or limited to particular types of program modifications. The algorithm is easily extended to facilitate selective retest at the integration and system levels where its savings are even more dramatic.

In the next section, we provide background about control dependence information and discuss control dependence graphs. In section 3, we discuss issues relevant to regression testing in general and selective retest in particular. We detail our algorithm in section 4 where we focus first on intraprocedural testing, and then present a method for efficiently extending our approach to interprocedural testing. In section 5 we discuss related work, and in section 6 we present our conclusions.

2 Background

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 statement X may be control dependent on several statements in the program, but we can always identify immediate control dependencies for X. For example, in procedure A1 in Figure 1, statement S7 is control dependent on predicates P4 and P3, but immediately control dependent only on P4.

![Figure 1: Procedure A1 and its CDG.](image-url)

One way to encode control dependence information is with a control dependence graph (CDG)[4]. The CDG for procedure A1 is given in Figure 1. A CDG contains several types of nodes. Statement nodes, shown as ellipses, represent simple statements. Predicate nodes, depicted by squares, stand for conditional statements, and have one or two “labeled” edges exiting that represent possible control paths. Region nodes, represented by circles, summarize control dependencies for statements in the region; the entry node, labeled entry, can be thought of as a region. An exit node represents the program’s exit.

A CDG represents control dependencies explicitly. In Figure 1 directed edges denote immediate control dependencies, and the hierarchical structure of the CDG encodes control dependencies generally. For example, it is clear from the CDG that statement (node) S7 is control dependent on both P3-true and P4-false, but immediately control dependent on P4-false. Figure 1 also illustrates the use of region nodes. For example, nodes S6, S7, S8 and R1 are control dependent on P4-false; without region node R5, there would be four edges labeled “F” from node P4. Thus, R5 summarizes control dependence on P4-false.

The CDG of Figure 1 also illustrates two interesting control dependence relationships in A1 that occur due to the presence of the exit from the loop body in statement S5. First, the while loop headed by region R1 is control dependent on predicate P4, which controls execution of S5. This creates a cycle in the control dependence graph (R1,P3,R2,P4,R5,R1), Second, statements S9 and S10, which follow the while loop, are control dependent on P3 because an execution that traverses P3-False reaches S9 and S10, while an execution that takes P3-True may not reach them.
### 3 Issues in regression testing

A number of regression testing issues have been addressed, but most work addresses the following problem:

**PROBLEM 1.** Given program \( P \), its modified version \( P' \), and test set \( T \) used previously to test \( P \). Find a way, making use of \( T \), to gain sufficient confidence in the correctness of \( P' \).

Typical solutions to this problem consist of the following steps:

1. Identify the modifications made to \( P \), and obtain a mapping between code segments in \( P \) and \( P' \).
2. Using the results of step 1, select \( T' \subseteq T \), a set of tests that may reveal modification-related errors in \( P' \).
3. Run \( T' \) on \( P' \), establishing \( P' \)'s correctness with respect to \( T' \).
4. If necessary, create new tests for \( P' \). These may include new functional tests required by changes in specifications, and/or new structural tests required by applicable coverage criteria.
5. Create \( T'' \), a new test set/history for \( P' \).

Step 2 typically requires test history information that identifies, for each test, its input, output, and execution history — a list of statements or regions of code exercised by the test. For example, Figure 2 gives test history information for procedure A1, listing regions rather than statements in the execution history.

![Table](#)

<table>
<thead>
<tr>
<th>test number</th>
<th>input</th>
<th>output</th>
<th>execution history</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>empty file</td>
<td>0</td>
<td>entry, R1, R3</td>
</tr>
<tr>
<td>T2</td>
<td>-1</td>
<td>error</td>
<td>entry, R1, R2, R4</td>
</tr>
<tr>
<td>T3</td>
<td>1 2 3</td>
<td>2</td>
<td>entry, R1, R2, R5, R3</td>
</tr>
</tbody>
</table>

Figure 2: Test history information for procedure A1.

**Criterion 2 - Precision.** The “retest-all” strategy is a safe approach, but it is also imprecise. In addition to selecting all potentially revealing tests, it also selects tests that cannot possibly exhibit changed behavior. Ideally, a selective retest strategy should be precise: it should select only tests that will exhibit different behavior. However, it is undecidable whether an arbitrary test will exhibit changed behavior. Therefore we can seek, at best, to maximize precision.

**Criterion 3 - Efficiency.** A selective retest algorithm should be efficient: it should be automatable, and run quickly enough to be practical in the limited time typically allotted to regression testing. It should also record test histories in as little space as possible.

**Criterion 4 - Generality.** Finally, a selective retest algorithm must be general: it should be applicable to all languages and language constructs, effective on realistic programs, and capable of handling arbitrarily complex code modifications.

---

### Fragment F1

\[
\begin{align*}
\text{S1.} & \quad \text{if } P \text{ then} \\
\text{S2.} & \quad a := 2 \\
\text{S3.} & \quad \text{end}
\end{align*}
\]

### Fragment F1'

\[
\begin{align*}
\text{S1.} & \quad \text{if } P \text{ then} \\
\text{S2.} & \quad a := 2 \\
\text{S2a.} & \quad b := 3 \\
\text{S3.} & \quad \text{end}
\end{align*}
\]

Figure 3: Code showing addition of statements.
4 Our method for test selection

In this section we present our method for selective retest. The next subsection presents observations that motivate our approach. Section 4.2 presents our algorithm, and illustrates it on several examples. Section 4.3 addresses interprocedural testing.

4.1 Motivating observations

Our goal is to distinguish potentially revealing tests in a modified program from those that cannot exhibit altered behavior. Toward this goal we offer the following discussion (presented more formally in [17]).

In order for a particular test $T_i$, run originally on program $P_i$, to exhibit different behavior when run on program $P'$, the sequence of statements that $T_i$ traverses in $P'$ must differ from the sequence it traversed in $P$. If test $T_i$, run on $P'$, does not traverse any modified statements, does not traverse any new statements, does not miss any statements it traversed in $P$, and traverses all statements in the same order as it did on $P$, it cannot behave any differently in $P'$ than it did in $P$. For example, Figure 5 shows program fragments F4 and F4', and two tests used on F4. Statement S2 has been modified, yielding S2'. Of the two test cases, only T1 traverses S2'. T2 does not traverse S2' and thus, in the absence of other changes, cannot possibly exhibit different behavior, so it does not need to be rerun. A first task in selecting tests for retest is to distinguish tests that traverse changed code from those that do not. This statement may seem obvious, yet it has been overlooked in some previous techniques[1, 3]. These techniques suggest selection of all tests that traverse affected statements in $P'$; affected statements are program statements that make use of the results of, or are otherwise affected by, some code change. Affected code is quite different from changed code since code can be affected without having been modified, deleted, or added. It is possible for a test to traverse an affected statement without traversing any changed code. Such tests cannot exhibit differences in program behavior, and need not be rerun. For example, in Figure 5, S4 is an affected point, because the definition in modified statement S2' reaches S4. However, test case T2, which traverses S4, does not traverse any modified points, and cannot possibly exhibit different behavior.

Note that changes in a program are reflected by changes in its CDG. Deletion, addition, or modification of a statement in $P$ results in the addition, deletion, or modification of a node in $P$'s CDG. Complex changes to $P$, of course, result in substantial differences in $P$'s CDG. Note further that we can attach to each CDG region a list of tests known to enter that region (that is, tests known to execute statements nested under that region node), and we can keep test execution histories that list these regions (as in Figure 2). We now present the fundamental theorem on which our technique is based (for reasons of space, we omit the proof; interested readers are referred to [17]).

THEOREM 1. Given the CDG ($CDG$) of program $P$, the CDG ($CDG'$) of program $P'$, and test suite $T$ in which test execution histories list the regions in $P$ traversed by each test; the only tests in $T$ that can traverse different sequences of statements in $P'$ and $P$ are those attached to some region node $R$ in $P$, such that $R$ has a corresponding region node $R'$ in $P'$, and $R'$'s immediate children have been changed.
It follows from Theorem 1 that a simple approach to test selection is to select all tests that enter the parent region of any changed node. It is these tests, and only these tests, that may exhibit changes in program behavior. The problem with this simple approach is that it requires us to obtain a mapping of regions in CDG to regions in CDG', so that we compare all "corresponding" regions. Such a mapping is difficult to obtain in the presence of complex or multiple code changes. However, by traversing nodes in CDG properly we avoid this problem, and obtain an opportunity for optimization as well. Observe that once we have detected a difference among the children of region R in P, and selected the tests attached to R, we need not consider any of R's control dependence successors in CDG (unless they can be reached from some other region not control dependent on R). For example, if statement S6 in procedure A1 (Figure 1) is modified, and the condition in predicate P4 is changed to "n > 0", then we want to rerun tests attached to R5 and R2. However, the only tests that may reach R5 are those that reached R2. Therefore, once we have examined R2 and selected its tests, we need not proceed further down in CDG; the tests attached to R5 contributed along this chain of control dependencies have already been selected.

Thus, an efficient test selection algorithm need not locate all regions immediately enclosing changes; it need only search the CDG until some region R enclosing changes is found, and return the tests attached to R. In doing so, the algorithm automatically selects tests attached to regions control dependent on R. By traversing the CDG in this fashion, we obviate the need for complete information on code changes, or for complete information on the corresponding sections of code in two program versions. We need only check, at any node reached in the CDG, whether the children of that node in the new program differ from the children of the corresponding node in the old program. If so, we need not worry about identifying nested changes.

These observations lead us to the following efficient algorithm.

### 4.2 The test selection algorithm

Our algorithm SelectTests, given in Figure 6, takes a procedure P, its changed version P', and the test history for P, and returns T', the subset of tests from T that could possibly expose errors if run on P'. The algorithm first constructs the CDGs for P and P', and then calls procedure Compare with the entry nodes E and E' of the two CDGs.

Compare is a recursive procedure. Given any two CDG nodes N and N', Compare marks these nodes

```
algorithm SelectTests
input    Procedure P, changed version P',
         and test set T
output   test set T'
begin
   construct CDG and CDG', CDG's of P and P'
   let E and E' be entry nodes of CDG and CDG'
   T' = Compare( E, E' )
end

procedure Compare
input    N and N': nodes in CDG and CDG'
output   test set T'
begin
   mark N and N' "visited"
   if the children of N and N' differ
      return (all tests attached to N )
   else
      T' = NULL
      for each region or predicate child node of N
         not yet "visited" do
         find C', the corresponding child of N'
         T' = T' ∪ Compare( C, C' )
      end (* for *)
   end (* if *)
end
```

Figure 6: Procedure to calculate tests to reexecute.

"visited", and then determines whether the children of these nodes are equivalent. (We discuss methods for determining equivalence later in this section.) If any two children are not equivalent, a difference between P and P' has been encountered. In this case, the only tests of P that may have traversed the change in P' are those that traversed N in P. Thus, Compare returns all tests known to have traversed N. If, on the other hand, the children of N and N' are equivalent, Compare calls itself on all pairs of equivalent nonvisited predicate or region nodes that are children of N and N', and returns the union of the tests (if any) required to test changes under these children.

Figure 7 presents procedure A2, a changed version of procedure A1. In A2, statement S7 has (erroneously) been deleted, and statement S5a has been added. When called with A1 and A2, SelectTests constructs the CDGs for A1 and A2 (shown in Figures 1 and 7, respectively) and calls Compare with entry and entry'. Compare finds the children of these nodes equivalent, and invokes itself (invocation 2) on R1 and R1'. Recursive calls continue in this manner on nodes P3 and P3' (invocation 3), R2 and R2' (invocation 4), and P4 and P4' (invocation 5); in each case the children of the nodes are found equivalent. In invocation 6 (on R4 and R4'), Compare discovers nonequivalent children, and thus returns test T2, the only test attached to R4, to invocation 5. Next, Compare calls
itself with R5 and R5'. Compare discovers nonequivalent children again, and returns T3, the only test attached to R5. (Because R1 has already been visited, Compare does not examine it again.) Returning up the tree, {T2,T3} is passed back to invocation 4, and then to invocation 3. Here, Compare calls itself with R3 and R3', finds no differences, and returns a null set. Invocation 3 passes {T2,T3} up the tree to invocation 2, then to invocation 1, and finally to the main procedure. The resulting test set, {T2,T3}, contains all the tests that could possibly exhibit different behavior in A2. Had the deletion of S7 been the only change, only {T3} would have been returned. Had the addition of S5a been the only change, only {T2} would have been returned. Most other methods [1, 2, 5, 7, 8, 9, 15, 20] fail to identify T2 and/or T3 as necessary.

To see how SelectTests handles predicate changes, imagine what happens if line P4 in procedure A1 is also changed (erroneously) to “n>0”. (This change alters only the text associated with node P4 in program A2's CDG.) Called with procedures A1 and A2, SelectTests proceeds as in the previous example until it reaches R2 and R2'. Here it finds nonequivalent children, and returns {T2,T3}. Note that in this case, no analysis is needed on nodes under P4. No other methods for test case selection make use of the opportunities afforded by the nesting of control dependencies to reduce analysis in this fashion.

Finally, Figure 8 shows how SelectTests works when large structural changes to procedures are involved. In procedure A3, another variation of A1, error handling code has been added that changes the overall structure of the procedure. This change is detected on the first call to Compare, because Compare detects the differences between children of entry and entry'. Again, no further analysis is needed. SelectTests returns the entire test set T for retest. This is not inappropriate, since all tests in T traverse altered code. No existing techniques concerned with selecting safe test sets produce more precise results.

The running time of SelectTests is bounded by the time it takes to construct CDGs for P and P', plus the time it takes to traverse these CDGs. CDG construction is an O(n²) algorithm for procedures containing n statements [4]. Since the size of the CDG is O(n²), traversal is O(n³). Thus, SelectTests is an O(n³) algorithm.

Procedure Compare in SelectTests requires a method for determining when the children of two CDG nodes differ. A simple algorithm for doing this merely checks corresponding nodes for identical text contents. This method succeeds as long as CDGs are ordered.

This simple algorithm is inexpensive and easy to implement, but can be imprecise. Consider, for example, the program fragments shown in Figure 9, in which two unrelated assignment statements, S1 and S2, are swapped. The simple algorithm considers statement order significant, so it finds the children of entry and entry' different, and returns all the tests attached to entry. An algorithm that distinguishes between semantic and syntactic changes would note that the behavior of the nodes under entry and entry' is not, in fact, different, and would continue searching the CDGs for real differences.

Horwitz and Reps [10] outline an algorithm that is useful in this context, that increases the precision of the differenting in exchange for an increase in compu-
that the method reduces the number of tests that must be rerun on a single procedure. However, the examples also suggest that when procedures are small and uncomplicated, and their test sets are small, it may be more cost efficient simply to rerun all tests.

This objection is mitigated when we consider interprocedural testing. Whereas test sets for single procedures may be small, sets for groups of procedures and complete systems are much larger. In this context the savings achieved by our algorithm are multiplied. Moreover, in this context the problem of selective retest separates even more completely from the problems surrounding test coverage criteria. Structural coverage criteria, when used at all, are typically applied only at the intraprocedural level. Although integration and system-level criteria have been proposed[14], in practice large-system tests are functional, specification-based tests. It is precisely with these tests that we are most interested in running all that may exhibit different behavior, without regard to coverage considerations. And it is precisely with these tests that we can benefit from an automatable method that selects tests based on program structure, because we cannot in general determine the code that may be affected (and the tests that must be rerun) merely by referring to program specifications.

Where interprocedural testing is concerned, our method may be applied as follows.

**Step 1.** During the “off time” between releases, collect test histories for all tests in the test suite. Store test history information for the regression test period. This process may be automated and run in batch.

**Step 2.** At regression test time, identify changed procedures. This information can be provided by a configuration management system, or by using the textual differencing utilities provided with most systems. This step, too, may be fully automated.

**Step 3.** Run SelectTests on all modified procedures $P$ and their new versions $P'$. By proceeding in this fashion, we also ensure that all tests that may traverse new procedures, or that may have traversed procedures now deleted, will also be run, because in those cases calls to the procedures must have been added or deleted, and all tests that may possibly traverse or have traversed such calls will necessarily be selected. Note that the algorithm need only be run on the modified procedures. The entire step may be automated and run in batch.

---

**Figure 9: Program fragments and their partial CDGs.**

4.3 Interprocedural regression testing

Thus far we have considered our method in an intraprocedural context. Our examples demonstrate...
Step 4. Perform testing using the selected tests.

Step 5. Add any new tests deemed necessary to meet functional or structural coverage criteria for the integration or system tests, and run these tests.

Figure 10 illustrates the savings that are possible with this approach. The figure depicts an interactive system wherein a central driver routine controls access to banks of procedures that handle different classes of inputs. The figure suggests a potential modification to statement 121 in the driver (shown in the dotted box). A naive approach to interprocedural testing might assume that since the driver routine is modified, all 1387 tests that traverse that routine must be rerun. Applying SelectTests to the driver routine, however, indicates that only 45 of these tests traverse modified code in the procedure; a substantial reduction in workload.

5 Related work

In [16], we present a comprehensive review and evaluation of previous methods for selective retest. This section summarizes results demonstrated at length in that report, in order to contrast previous methods with ours.

Fischer, Raji, and Chruscicki[5] address the problem of selecting a minimal test set (from an existing test suite) that yields segment (basic block) coverage of all statements reachable from modified code; Hartmann and Robson[9] extend and implement Fischer’s approach. Although minimal test sets may provide structural coverage of changed code, they are not necessarily safe because they may omit potentially revealing tests. When tests have been created to satisfy a structural coverage criterion, minimization has value, but where functional tests are concerned we prefer not to discard potentially revealing tests.

Sherlund and Korel[19] present a method centered on types of modifications. Their method is also a minimization approach: for each affected point in a program, it selects some test that traverses a modification and reaches the affected point. Thus, some potentially revealing tests may be left unidentified. Moreover their method, unlike ours, demands prior knowledge of modifications. Finally, since their algorithm does not handle additions or deletions of predicate statements, or offer a precise approach to handling multiple modifications, it is less general than ours.

Yau and Kishimoto[23] present a selection method that depends on input partitions, and uses symbolic execution to determine tests that traverse modified blocks. Unlike ours, their method relies on knowledge of modifications, and is computationally expensive due to the use of symbolic execution. The authors suggest that the method is viable only for numerical programs that are not inordinately complex.

Dataflow-based regression testing methods have been advanced by Ostrand and Weyuker[15], Taha, Thebaut and Liu[20], and Harrold and Soffa[7, 8]. The greatest drawback of these methods is that unlike ours, they require prior knowledge of modifications, typically assumed to be derived through incremental dataflow analysis in a programming environment. Moreover, none of the methods is safe, because none account completely for the effects of deleted code. These methods may also require more extensive analysis than ours (particularly at the interprocedural level).

Benedusi, Cimilita, and De Carlini[2] suggest an approach based on path analysis. Like ours, their method has an advantage over others in not requiring prior knowledge of modifications. However, their method does not consider code deletion and thus is not safe. Moreover, since their method is based upon the use of path expressions (the length of which may be exponential in the size of a program) the algorithm is more expensive than ours.

Bates and Horwitz[1] present a method based on PDG adequacy criteria. Unlike ours, their method requires knowledge of modifications. Also, since their
method selects all tests that traverse affected statements, it may select tests that cannot possibly traverse new or modified code, an imprecision our method avoids. Moreover, their method is not safe because it does not handle deleted code. Finally, the method is more computationally expensive than ours, requiring both execution and regular slices on each statement.

Leung and White[12, 21] address regression test selection at the interprocedural level. Their approach places a “firewall” around affected modules. However, this approach is not safe; it considers only tests within and between modified and affected modules, thus missing tests (such as system tests) developed for larger enclosing sections of the program, that happen to traverse the modified code and may exhibit different behavior. Moreover, though the technique offers guidance in test selection, suggesting which units must be retested, no algorithm for selecting a subset of the tests for various units is presented.

Laski and Szermer[11] present an algorithm that determines the modified portions of a program by identifying new, deleted, and modified “clusters” of statements. They recommend selecting tests that traverse these clusters. Applied to regression testing, Laski and Szermer’s algorithm is both safe and general, but it is more expensive than ours. Their method uses an $O(n^3)$ algorithm (where $n$ is the number of program statements) to compute all clusters in a program using control dependence information, while we proceed directly from a CDG and when a modification is found, do not need to further analyze code beneath the enclosing region. Since we perform selective retest by traversing the CDG, our algorithm is $O(n^2)$. Moreover, Laski and Szermer’s algorithm identifies clusters that are larger than our regions. For example, given procedure A1 in Figure 1, their algorithm identifies only one cluster (composed of P3, P4, and S5-S10), whereas our algorithm is finer grained and identifies four regions within this cluster. It is not clear how larger clusters will affect the size of the test sets Laski and Szermer’s algorithm selects, but it seems likely that larger cluster size will lead to larger test set size.

6 Conclusion

In this paper, we have presented criteria for a selective retest algorithm, and an algorithm that meets these criteria. Our algorithm, SelectTests, is safe, because it selects every test that can possibly exhibit different behavior in a modified program, including tests that cover new or deleted code. Despite its conservatism, SelectTests is more precise than existing algorithms aimed at achieving safe solutions, partly because it does not base its selection on whether code is affected, looking instead for changed code. Our algorithm is faster and more space-efficient than comparable safe algorithms, and does not require information on code modifications. Finally, our method is general. It handles arbitrary programs, language constructs, and modifications, and facilitates retest at the integration and system levels.

Our technique is also useful in other contexts. Typically, the final step in regression testing is the update of test histories. Tests must be reexecuted, their new execution histories and outputs archived, and obsolete tests eliminated. Our algorithm facilitates this step: the tests it reselects are precisely those tests whose inputs, outputs, and histories may differ; no other tests need be considered.

Furthermore, our technique supports the use of certain structural coverage criteria. Typically, selective retest methods first choose tests to retest, run these tests, and finally determine the “components” of the coverage criteria that remain to be satisfied. By identifying regions of changed code, our algorithm also recognizes the sections of a program in which coverage may need to be reestablished. For example, if SelectTests determines that all (and only) the tests entering a particular region $R$ need to be run, then we know that if test set $T$ was statement- or branch- adequate on program $P$, it remains statement or branch adequate on all statements and branches not localized under $R$. We need only analyze coverage of statements and branches under $R$ in order to ascertain $T'$s adequacy on $P'$.

We are currently implementing our algorithm to perform experiments and obtain empirical evidence regarding its efficiency.

Several avenues for future research appear promising. Although SelectTests is more precise than algorithms interested in safety, greater precision is desirable. The fact that a test traverses changed code is not sufficient to indicate that it will exhibit changed behavior. For example, the test must reach some output statement, and that output statement must be somehow “affected” by the change that was traversed. Consider what happens, for example, if line 1 of procedure A1 (Figure 1) is modified. In this case, SelectTests selects every test in $T$ for retest. However, neither test T1 nor test T2 is germane in this case, because neither test traverses code that makes use of the variable defined in line 1. Several existing methods attempt to weed out such non-revealing tests, however no such method operates safely and without knowledge of modifications. Our future research will address
this problem. We also intend to further explore interprocedural applications, and support for adequacy criteria.

Acknowledgements

We wish to thank Brian Malloy for his many helpful suggestions. We also wish to thank the reviewers, who offered many useful comments that improved the presentation of this work.

References


