

A Safe, Efficient Regression Test Selection Technique

Gregg Rothermel	Mary Jean Harrold
Department of Computer Science	Department of Computer and Information Science
Oregon State University	The Ohio State University
Dearborn Hall 307-A	395 Drees Lab, 2015 Neil Avenue
Corvallis, OR	Columbus, OH 43210-1277
grother@cs.orst.edu	harrold@cis.ohio-state.edu

Abstract

Regression testing is an expensive but necessary maintenance activity performed on modified software to provide confidence that changes are correct and do not adversely affect other portions of the software. A regression test selection technique chooses, from an existing test set, tests that are deemed necessary to validate modified software. We present a new technique for regression test selection. Our algorithms construct control flow graphs for a procedure or program and its modified version, and use these graphs to select tests that execute changed code from the original test suite. We prove that under certain conditions, the set of tests our technique selects includes every test from the original test suite that can expose faults in the modified procedure or program. Under these conditions our algorithms are *safe*. Moreover, although our algorithms may select some tests that cannot expose faults, they are at least as precise as other safe regression test selection algorithms. Unlike many other regression test selection algorithms, our algorithms handle all language constructs and all types of program modifications. We have implemented our algorithms; initial empirical studies indicate that our technique can significantly reduce the cost of regression testing modified software.

Keywords: software maintenance, regression testing, selective retest, regression test selection.

1 Introduction

Software maintenance activities can account for as much as two-thirds of the overall cost of software production [40, 47]. One necessary maintenance activity, *regression testing*, is performed on modified software to provide confidence that the software behaves correctly and that modifications have not adversely impacted the software's quality. Regression testing is expensive; it can account for as much as one-half of the cost of software maintenance [5, 29].

An important difference between regression testing and development testing is that during regression testing, an established suite of tests may be available for reuse. One regression testing strategy reruns all such tests, but this *retest all* approach may consume inordinate time and resources. *Selective retest techniques*, in contrast, attempt to reduce the time required to retest a modified program by selectively reusing tests and selectively retesting the modified program. These techniques address two problems: (1) the problem of selecting tests from an existing test suite and (2) the problem of determining where additional tests may be required. Both of these problems are important; however, this paper addresses the first problem – the *regression test selection problem*.

This paper presents a new regression test selection technique. Our algorithms construct control flow graphs for a procedure or program and its modified version, and use these graphs to select tests that execute changed code from the original test suite. Our technique has several advantages over other regression test

selection techniques. Unlike many techniques, our algorithms select tests that may now execute new or modified statements, and tests that formerly executed statements that have been deleted from the original program. We prove that under certain conditions the algorithms are *safe*: that is, they select *every* test from the original test suite that can expose faults in the modified program. Moreover, although the algorithms may select some tests that cannot reveal faults, they are more precise than other safe algorithms because they select fewer such tests than those algorithms. Our algorithms automate an important portion of the regression testing process, and operate more efficiently than most other regression test selection algorithms. Finally, our algorithms are more general than most other techniques. They handle regression test selection for single procedures and for groups of interacting procedures. They also handle all language constructs and all types of program modifications for procedural languages.

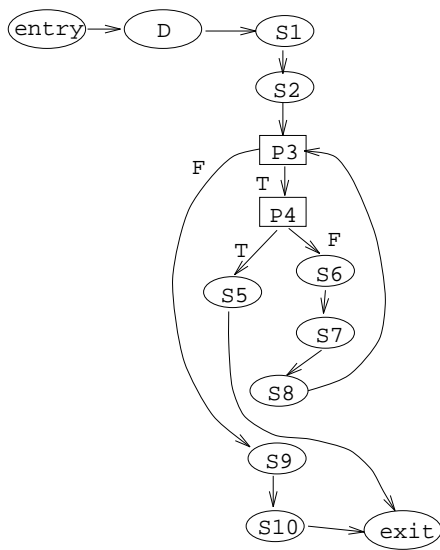
We have implemented our algorithms, and conducted empirical studies on several subject programs and modified versions. The results suggest that in practice, the algorithms can significantly reduce the cost of regression testing a modified program.

2 Background

The following notation is used throughout the rest of this paper. Let P be a procedure or program, P' be a modified version of P , and S and S' be the specifications for P and P' , respectively. $P(i)$ refers to the output of P on input i , $P'(i)$ refers to the output of P' on input i , $S(i)$ refers to the specified output for P on input i , and $S'(i)$ refers to the specified output for P' on input i . Let T be a set of tests (a *test suite*) created to test P . A *test* is a 3-tuple, $\langle \text{identifier}, \text{input}, \text{output} \rangle$, in which *identifier* identifies the test, *input* is the input for that execution of the program, and *output* is the specified output, $S(\text{input})$, for this input. For simplicity, the sequel refers to a test $\langle t, i, S(i) \rangle$ by its identifier t , and refers to the outputs $P(i)$ and $S(i)$ of test t for input i as $P(t)$ and $S(t)$, respectively.

2.1 Control Flow Graphs

A *control flow graph* (CFG) for procedure P contains a node for each simple or conditional statement in P ; edges between nodes represent the flow of control between statements. Figure 1 shows procedure `avg` and its CFG. In the figure, *statement nodes*, shown as ellipses, represent simple statements. *Predicate nodes*, shown as rectangles, stand for conditional statements. Labeled edges (*branches*) leaving predicate nodes represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in P to which they correspond. The figure uses statement numbers as node labels; however, the actual code of the associated statements could also serve as labels. Case statements can be represented in CFGs as nested if-else statements; in this case, every CFG node has either one unlabeled out edge or two out edges labeled “T” and “F”. Declarations and nonexecutable initialization statements can be represented collectively as a single node labelled “D”, associated with this node in the order in which they are encountered by the compiler. (Section 3.1.4 discusses other methods for handling case statements, declarations, and other types of nonexecutable initialization statements.) A unique *entry node* and a unique *exit node* represent entry to and exit from P , respectively. The CFG for



```

Procedure avg
S1. count = 0
S2. fread(fileptr,n)
P3. while (not EOF) do
P4.   if (n<0)
S5.     return(error)
      else
S6.       numarray[count] = n
S7.       count++
      endif
S8.   fread(fileptr,n)
      endwhile
S9. avg = calcavg(numarray,count)
S10. return(avg)
  
```

Figure 1: Procedure avg and its CFG.

a procedure P has size, and can be constructed in time, linear in the number of simple and conditional statements in P [2].

Code Instrumentation

Let P be a program with CFG G . P can be instrumented such that when the instrumented version of P is executed with test t , it records a *branch trace* that consists of the branches taken during this execution. This branch trace information can be used to determine which edges in G were traversed when t was executed: an edge (n_1, n_2) in G is *traversed* by test t if and only if, when P is executed with t , the statements associated with n_1 and n_2 are executed sequentially at least once during the execution. The information thus gathered is called an *edge trace* for t on P . An edge trace for t on P has size linear in the number of edges in G , and can be represented by a bit vector.

Given test suite T for P , a *test history* for P with respect to T is constructed by gathering edge trace information for each test in T , and representing it such that for each edge (n_1, n_2) in G , the test history records the tests that traverse (n_1, n_2) . This representation requires $O(e|T|)$ bits, where e is the number of edges in G , and $|T|$ is the number of tests in T . For CFGs of the form defined above, e is no greater than twice the number of nodes in G ; thus, e is linear in the size of P . Figure 2 gives a table that reports test information and the corresponding test history for program avg of Figure 1.

For convenience, assume the existence of function $\text{TestsOnEdge}(n_1, n_2)$, that returns a bit vector v of size $|T|$ bits such that the k th bit in v is set if and only if test k in T traversed edge (n_1, n_2) in G .

TEST INFORMATION

test	input	output	edges traversed
t1	empty file	0	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,S9), (S9,S10),(S10,exit)
t2	-1	error	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S5),(S5,exit)
t3	1 2 3	2	(entry,D),(D,S1), (S1,S2),(S2,P3),(P3,P4), (P4,S6),(S6,S7),(S7,S8) (S8,P3),(P3,S9),(S9,S10), (S10,exit)

TEST HISTORY

edge	TestsOnEdge(edge)
(entry,D)	111
(D,S1)	111
(S1,S2)	111
(S2,P3)	111
(P3,P4)	011
(P3,S9)	101
(P4,S5)	010
(P4,S6)	001
(S5,exit)	010
(S6,S7)	001
(S7,S8)	001
(S8,P3)	001
(S9,S10)	101
(S10,exit)	101

Figure 2: Test information and test history for procedure `avg`.

2.2 Regression Testing

Research on regression testing spans a wide variety of topics, including test environments and automation [10, 12, 24, 25, 58], capture-playback mechanisms [33], test suite management [16, 23, 33, 51, 56], program size reduction [7], and regression testability [29]. Most recent research on regression testing, however, concerns *selective retest techniques* [1, 4, 6, 8, 11, 13, 14, 15, 18, 19, 20, 21, 22, 23, 27, 28, 30, 31, 39, 41, 43, 44, 45, 49, 50, 51, 52, 54, 55, 57].

Selective retest techniques reduce the cost of regression testing by reusing existing tests, and identifying portions of the modified program or its specification that should be tested. Selective retest techniques differ from the *retest-all* technique, which runs all tests in the existing test suite. Leung and White [32] show that a selective retest technique is more economical than the retest-all technique only if the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that the selective retest technique omits.

A typical selective retest technique proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P' .
2. Test P' with T' , establishing P' 's correctness with respect to T' .
3. If necessary, create T'' , a set of new functional or structural tests for P' .
4. Test P' with T'' , establishing P' 's correctness with respect to T'' .
5. Create T''' , a new test suite and test history for P' , from T , T' , and T'' .

In performing these steps, a selective retest technique addresses several problems. Step 1 involves the *regression test selection problem*: the problem of selecting a subset T' of T with which to test P' . This problem includes the subproblem of identifying tests in T that are *obsolete* for P' . Test t is obsolete for program P' if and only if t specifies an input to P' that, according to S' , is invalid for P' , or t specifies an

invalid input-output relation for P' . Step 3 addresses the *coverage identification problem*: the problem of identifying portions of P' or S' that require additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing tests and checking test results for correctness. Step 5 addresses the *test suite maintenance problem*: the problem of updating and storing test information. Although each of these problems is significant, we restrict our attention to the regression test selection problem. We further restrict our attention to code-based regression test selection techniques, which rely on analysis of P and P' to select tests.

There are two distinguishable phases of regression testing: a *preliminary phase* and a *critical phase*. The preliminary phase of regression testing begins after the release of some version of the software; during this phase, programmers enhance and correct the software. When corrections are complete, the *critical phase* of regression testing begins; during this phase regression testing is the dominating activity, and its time is limited by the deadline for product release. It is in the critical phase that cost minimization is most important for regression testing. Regression test selection techniques can exploit these phases. For example, a technique that requires test history and program analysis information during the critical phase can achieve a lower critical phase cost by gathering that information during the preliminary phase.

There are various ways in which this two-phase process may fit into the overall software maintenance process. A *big bang* process performs all modifications, and when these are complete proceeds with regression testing. An *incremental* process performs regression testing at intervals throughout the maintenance life cycle, with each testing session aimed at the product in its current state of evolution. Preliminary phases are typically shorter for the incremental model than for the big bang model; however, for both models, both phases exist and can be exploited.

3 Regression test selection algorithms

For reasons that will become clear, our goal is to identify all nonobsolete tests in T that execute changed code with respect to P and P' . In other words, we want to identify tests in T that (1) execute code that is new or modified for P' or (2) executed code in P that is no longer present in P' . To capture the notion of these tests more formally, we define an execution trace $ET(P(t))$ for t on P to consist of the sequence of statements in P that are executed when P is executed with t . Two execution traces $ET(P(t))$ and $ET(P'(t))$ are *equivalent* if they have the same lengths and if, when their elements are compared from first to last, the text representing the pairs of corresponding elements is lexicographically equivalent. Two text strings are *lexicographically equivalent* if their text (ignoring extra white space characters when not contained in character constants) is identical. Test t is *modification-traversing for P and P'* (or simply, t is *modification-traversing*) if and only if $ET(P(t))$ and $ET(P'(t))$ are nonequivalent.

To identify the modification-traversing tests in T we must identify the nonobsolete tests in T that have nonequivalent execution traces in P and P' . Assume henceforth that T contains no obsolete tests, either because it contained none initially or because we have removed them. (If we cannot effectively determine test obsolescence we cannot effectively judge test correctness. Thus, this assumption is necessary in order to reuse tests at all, whether selectively or not.) In addition, assume that for each test $t \in T$, P terminated and

produced its specified output when we executed it with t . Our task is to partition T as nearly as possible into tests that have different execution traces in P and P' , and tests that do not.

The next section presents our intraprocedural test selection algorithm – an algorithm that operates on individual procedures. Section 3.2 presents our interprocedural test selection algorithm – an algorithm that operates on entire programs or subsystems.

3.1 Intraprocedural test selection

There is a unique mapping between an execution trace for a program and the nodes of the CFG that represents that program. This mapping is obtained by replacing each statement in the execution trace by its corresponding CFG node, or equivalently, by the label of that node. This mapping yields a *traversal trace*: given CFG G for P , and test t with execution trace $ET(P(t))$, the traversal trace for t on G is $TR(P(t))$. If N is a node in traversal trace $TR(P(t))$, the *traversal trace prefix* for $TR(P(t))$ with respect to N is the portion of $TR(P(t))$ that begins with the first node in the trace and ends at N .

Assume that CFG nodes are labelled by the text of the statements to which they correspond. Given two traversal traces, a *pairwise comparison* of the traces compares the labels on the first nodes in each trace to each other, then compares the labels on the second nodes in each trace to each other, and so forth. If a test t has nonequivalent execution traces in P and P' , a pairwise comparison of the traversal traces for t in P and P' reaches a first pair of nodes N and N' whose labels are not lexicographically equivalent. In this case we say that the two traversal traces are nonequivalent, but that the traversal trace prefixes of those traces are equivalent up to and not including N and N' .

Suppose t has nonequivalent execution traces $ET(P(t))$ and $ET(P'(t))$ in P and P' , and let N and N' be the *first* pair of nodes found to be not lexicographically equivalent during a pairwise comparison of the corresponding traversal traces, $TR(P(t))$ and $TR(P'(t))$. The traversal trace prefixes of $TR(P(t))$ and $TR(P'(t))$ with respect to N and N' , respectively, are equivalent up to, but not including, N and N' . In other words, if t is modification-traversing for P and P' , there is some pair of nodes N and N' in G and G' , the CFGs for P and P' , respectively, such that N and N' have labels that are not lexicographically equivalent, and N and N' are endpoints of traversal trace prefixes that are equivalent up to, but not including, N and N' . To find tests that are modification-traversing for P and P' , we can synchronously traverse CFG paths that begin with the entry nodes of G and G' , looking for pairs of nodes N and N' whose labels are not lexicographically equivalent. When traversal of CFG paths finds such a pair, we use `TestsOnEdge` to select all tests known to have reached N .

3.1.1 The basic test selection algorithm

Figure 3 presents `SelectTests`, our intraprocedural regression test selection algorithm. `SelectTests` takes a procedure P , its modified version P' , and the test suite T for P , and returns T' , a set that contains tests that are modification-traversing for P and P' . `SelectTests` first initializes T' to ϕ , and then constructs CFGs G (with entry node E) and G' (with entry node E') for P and P' , respectively. Next, the algorithm calls `Compare` with E and E' . `Compare` ultimately places tests that are modification-traversing for P and P'

```

algorithm   SelectTests( $P, P', T$ ): $T'$ 
input       $P, P'$ : base and modified versions of a procedure
               $T$ : a test set used to test  $P$ 
output      $T'$ : the subset of  $T$  selected for use in regression testing  $P'$ 

1. begin
2.    $T' = \phi$ 
3.   construct  $G$  and  $G'$ , CFGs for  $P$  and  $P'$ , with entry nodes  $E$  and  $E'$ 
4.   Compare(  $E, E'$  )
5.   return  $T'$ 
6. end

procedure   Compare( $N, N'$ )
input       $N$  and  $N'$ : nodes in  $G$  and  $G'$ 

7. begin
8.   mark  $N$  “ $N'$ -visited”
9.   for each successor  $C$  of  $N$  in  $G$  do
10.     $L$  = the label on edge  $(N, C)$  or  $\epsilon$  if  $(N, C)$  is unlabeled
11.     $C'$  = the node in  $G'$  such that  $(N', C')$  has label  $L$ 
12.    if  $C$  is not marked “ $C'$ -visited”
13.      if  $\neg$  LEquivalent(  $C, C'$  )
14.         $T' = T' \cup$  TestsOnEdge( $(N, C)$ )
15.      else
16.        Compare(  $C, C'$  )
17.      endif
18.    endif
19.  endfor
20. end

```

Figure 3: Algorithm for intraprocedural test selection.

into T' . SelectTests returns these tests.¹

Compare is called with pairs of nodes N and N' , from G and G' , respectively, that are reached simultaneously during the algorithm’s comparisons of traversal trace prefixes. Given two such nodes N and N' , Compare determines whether N and N' have successors whose labels differ along pairs of identically labeled edges. If N and N' have successors whose labels differ along some pair of identically labeled edges, tests that traverse the edges are modification-traversing due to changes in the code associated with those successors. In this case Compare selects those tests. If N and N' have successors whose labels are the same along a pair of identically labeled edges, Compare continues along the edges in G and G' by invoking itself on those successors.

Lines 7-20 of Figure 3 describe Compare’s actions more precisely. When Compare is called with CFG nodes N and N' , Compare first marks node N “ N' -visited” (line 8). After Compare has been called once with N and N' it does not need to consider them again – this marking step lets Compare avoid revisiting pairs of nodes. Next, in the for loop of lines 9-19, Compare considers each control flow successor of N . For each successor C , Compare locates the label L on the edge from N to C , then seeks the node C' in G' such that (N', C') has label L ; if (N, C) is unlabeled ϵ is used for the edge label. Next, Compare considers C and

¹An earlier version of this algorithm [43] was based on control dependence graphs for P and P' . The possibility of performing that algorithm on CFGs was suggested by Weibao Wu (personal communication). The two approaches select the same test sets, but the CFG-based approach is more efficient, and easier to implement, than the earlier approach.

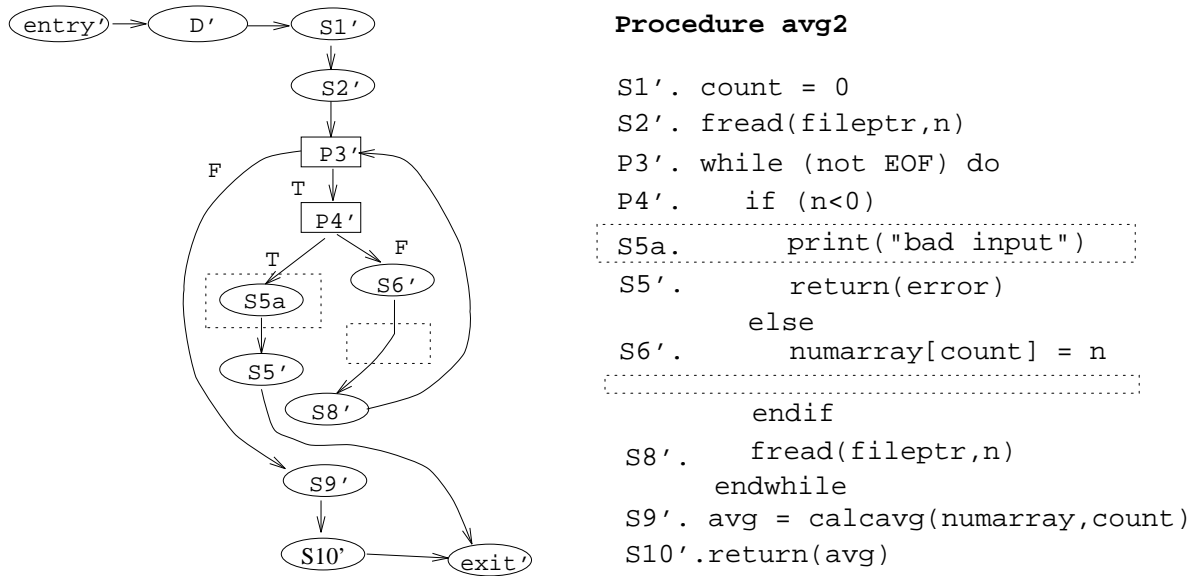


Figure 4: Procedure avg2 and its CFG.

C' . If C is marked “ C' -visited”, `Compare` has already been called with C and C' , so `Compare` does not take any action with C and C' . If C is not marked “ C' -visited”, `Compare` calls `LEquivalent` with C and C' . The `LEquivalent` function takes a pair of nodes N and N' , and determines whether the statements S and S' associated with N and N' are lexicographically equivalent. If `LEquivalent`(C, C') is false, then tests that traverse edge (N, C) are modification-traversing for P and P' ; `Compare` uses `TestsOnEdge` to identify these tests, and adds them to T' . If `LEquivalent`(C, C') is true, `Compare` invokes itself on C and C' to continue the graph traversals beyond these nodes.

We next consider several examples that illustrate how `SelectTests` works. Figure 4 presents procedure `avg2` and the CFG for `avg2`; `avg2` is a modified version of procedure `avg`, shown in Figure 1. In `avg2`, statement `S7` has erroneously been deleted and statement `S5a` has been added. When called with `avg` and `avg2`, and with test suite T (shown in Figure 2), `SelectTests` initializes T' to ϕ , constructs the CFGs for the two procedures, and calls `Compare` with `entry` and `entry'`. `Compare` marks `entry` “`entry'`-visited”, and then considers the successor of `entry`, D . `Compare` finds that D' is the corresponding successor of `entry'`, and because D is not marked “ D' -visited”, calls `LEquivalent` with D and D' . Because D and D' have the same labels (the declaration statements associated with the two nodes have not changed from `avg` to `avg2`), `LEquivalent` returns true, and `Compare` invokes itself on D and D' (invocation 2). Recursive calls continue in this manner on nodes $S1$ and $S1'$ (invocation 3), $S2$ and $S2'$ (invocation 4), and $P3$ and $P3'$ (invocation 5); in each case the successors of the nodes have lexicographically equivalent labels. On invocation 5, `Compare` must consider two successors of $P3$: $P4$ and $S9$. When `Compare` considers $S9$, it calls itself with $S10$ and $S10'$, and subsequently with `exit` and `exit'`, and selects no tests. When `Compare` considers $P4$ and $P4'$, it first seeks a true child of $P4'$ to compare with $S5$; it finds $S5a$ and calls `LEquivalent` with $S5$ and $S5a$. The

statement associated with $S5$ and the statement associated with $S5a$ are not lexicographically equivalent, so `LEquivalent` returns false; `Compare` uses `TestsOnEdge($P4, S5$)` to locate the set of tests ($\{t2\}$) that reach $S5$ in `avg` and adds these tests to T' . When `Compare` seeks a false successor of $P4'$, it finds $S6'$, and calls `LEquivalent` with $S6$ and $S6'$; `LEquivalent` returns true for these nodes, so `Compare` invokes itself on $S6$ and $S6'$. `Compare` finds the labels on the successors of these nodes, $S7$ and $S8'$, not lexicographically equivalent, and adds to T' the set of tests that traverse edge $(S7, S8)$; T' is now $\{t2, t3\}$. At this point the algorithm has compared all traversal trace prefixes either up to modifications or up to the exit node, so no further traversal is necessary; recursive `Compare` calls return to the main program, and the algorithm returns set $T' = \{t2, t3\}$, in which both tests are modification-traversing. Many other regression test selection techniques [4, 6, 14, 18, 20, 23, 39, 51] omit $t2$ or $t3$.

If, for `avg` and `avg2`, the deletion of $S7$ had been the only change, `SelectTests` would have returned only $\{t3\}$. If the addition of $S5a$ had been the only change, `SelectTests` would have returned only $\{t2\}$. In this latter case, `Compare` would eventually invoke itself with $S8$ and $S8'$, but find the successor of $S8$, $P3$, already marked “ $P3'$ -visited”; thus, `Compare` would not reinvoke itself with $P3$ and $P3'$.

To see how `SelectTests` handles changes in predicate statements, consider the result when line $P4$ in procedure `avg` is also changed (erroneously), to “ $n > 0$ ”. This change alters only the text associated with node $P4'$ in `avg2`'s CFG. In this case, when called with the CFGs for `avg` and `avg2`, `SelectTests` proceeds as in the previous example until it reaches $P3$ and $P3'$. Here it finds that successors $P4$ and $P4'$ have labels that are not lexicographically equivalent and selects $\{t2, t3\}$. The procedure does not need to analyze successors of $P4$ and $P4'$.

To see how `SelectTests` handles large-scale structural changes, consider the result when new error handling code is inserted into `avg`, such that the procedure checks `fileptr` as its first action, and executes the rest of its statements only if `fileptr` is not NULL. `SelectTests` detects this change on the second invocation of `Compare`, when `Compare` detects the differences between successors of D and D' . The procedure does not need to analyze successors of D and D' : it returns the entire test set T because all tests in T are modification-traversing.

To understand why, at line 12, `SelectTests` marks C “ C' -visited” rather than just “visited”, consider Figure 5. The figure contains procedure `twovisits` (far left), a modified version of that procedure, `twovisits'` (far right), and the CFGs for the two procedures (next to their respective procedures, with declaration nodes omitted). The two versions produce identical output for all values of x other than “0”. When $x = 0$, the execution trace for `twovisits` is $\langle \text{entry}, P1, S2, S3, S4, \text{exit} \rangle$, and the procedure prints “1”. For the same input, the execution trace for `twovisits'` is $\langle \text{entry}', P1', S2', S5', \text{exit}' \rangle$, and the procedure prints “2”.

When `SelectTests` runs on the CFGs for `twovisits` and `twovisits'`, it considers `entry` and `entry'` first, and then invokes itself on $P1$ and $P1'$. Suppose `SelectTests` next invokes itself on $S3$ and $S3'$ (our algorithm is not required to visit the successors of a pair of nodes in any particular order; if it were, we could reverse the contents of the `if` and `else` clauses in this example, and still make the point that we are about to make). `SelectTests` marks $S3$ “ $S3'$ -visited”, then continues with invocations on $S4$ and $S4'$, and `exit` and `exit'`, selecting no tests, because tests that take these paths in the two versions are not modification-

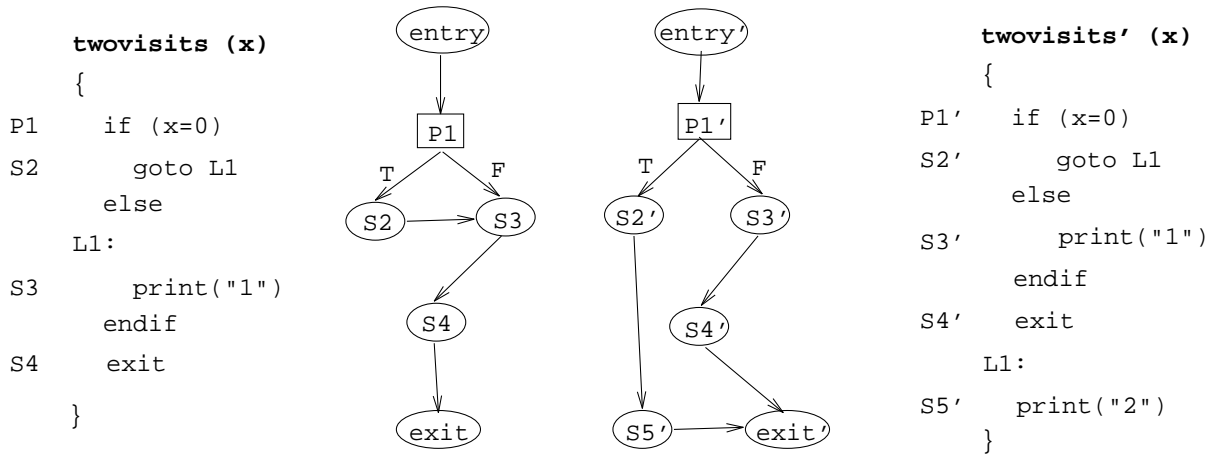


Figure 5: Procedures `twovisits` and `twovisits'`, and their CFGs.

traversing. Now, `SelectTests` resumes its consideration of successors of $P1$ and $P1'$ and invokes itself on $S2$ and $S2'$. `SelectTests` marks $S2$ “ $S2'$ -visited” and considers the successors of $S2$. $S2$ has only one successor, $S3$; the corresponding successor of $S2'$ is $S5'$. Here is the point we wish to make. If on visiting $S3$, `SelectTests` had simply marked $S3$ “visited”, and now seeing that $S3$ had been visited declined to visit it again, `SelectTests` would not compare $S3$ and $S5'$ and would not detect the need to select tests through $S3$. However, as the algorithm is written, `SelectTests` sees that $S3$ is only marked $S3'$ -visited, not $S5'$ -visited, and thus, proceeds to compare $S3$ and $S5'$, and select the necessary tests.

In certain cases, it is possible for `SelectTests` to select tests that are *not* modification-traversing for P and P' . This is not surprising, because the problem of precisely identifying these tests in general is PSPACE-hard; thus unless $P=NP$, no efficient algorithm will always identify precisely the tests that are modification-traversing for P and P' [42]. Figure 6, which illustrates this possibility, depicts a C function, `pathological`, and a modified version of that function, `pathological'`, with the CFGs for the versions (declaration nodes omitted). Each `while` construct in the versions first increments the value of x , and then tests the incremented value, to determine whether to enter or exit its loop. Suppose test suite T for `pathological` contains tests $t1$ and $t2$ that use input values “0” and “-2”, respectively, to exercise the function. Figure 7 shows the inputs, outputs, and traversal traces that result when `pathological` and `pathological'` are run on these tests. When `pathological` is run on tests $t1$ and $t2$, it outputs “1” for both. When `pathological'` is run on test $t1$, it also outputs “1”; however, when `pathological'` is run on test $t2$, it outputs “3”. Tests $t1$ and $t2$ both traverse edge $(P1, S4)$ in the CFG for `pathological`.

Consider the actions of `SelectTests`, invoked on `pathological` and `pathological'`. Called with `entry` and `entry'`, `Compare` invokes itself with $P1$ and $P1'$, then with $P2$ and $P2'$. When invoked with $P2$ and $P2'$, `Compare` considers their successors, $P1$ and $P3'$, respectively, finds their labels lexicographically equivalent, and invokes itself with them. `Compare` finds the labels of their successors, $S4$ and $P4'$, not lexicographically equivalent, and selects the tests on edge $(P1, S4)$, that is, tests $t1$ and $t2$.

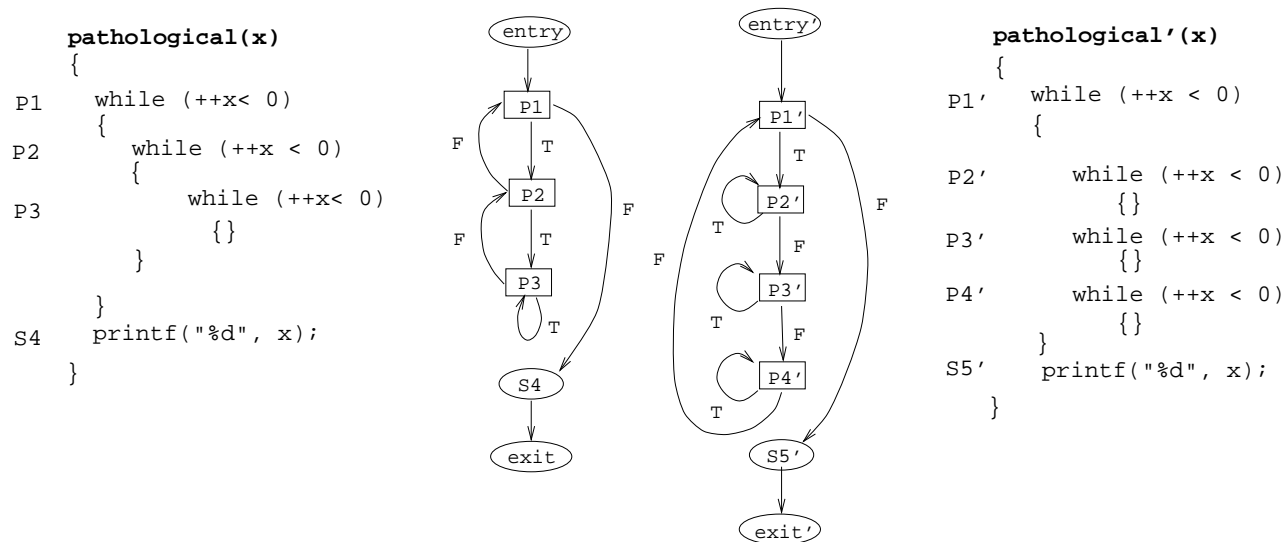


Figure 6: Procedures `pathological` and `pathological'`, and their CFGs.

test	input	output	traversal traces (showing node labels) for procedure <code>pathological</code>
t1	0	1	entry, while (++x < 0), printf("%d",x);, exit
t2	-2	1	entry, while (++x < 0), while (++x < 0), while (++x < 0), printf("%d",x);, exit

test	input	output	traversal traces (showing node labels) for procedure <code>pathological'</code>
t1	0	1	entry, while (++x < 0), printf("%d",x);, exit
t2	-2	3	entry, while (++x < 0), while (++x < 0), while (++x < 0), while (++x < 0), while (++x < 0), printf("%d",x);, exit

Figure 7: Test suites and traversal traces for `pathological` and `pathological'`.

The problem with this test selection is that, whereas *t2* is modification-traversing for the two versions of `pathological`, *t1* is *not* modification-traversing for the two versions. Figure 7 shows the traversal traces, which correspond to the execution traces, for the tests on the two versions. Test *t1* has equivalent execution traces for the two versions, whereas test *t2* does not: the traces for *t2* differ in their fifth elements. Thus, *t1* is *not* modification-traversing for the two versions, and `SelectTests` chooses it unnecessarily.

3.1.2 Correctness of the algorithm

Controlled regression testing is the practice of testing P' under conditions equivalent to those that were used to test P . Controlled regression testing applies the scientific method to regression testing: to determine whether code modifications cause errors, test the new code, holding all other factors that might affect program behavior constant. The importance of controlled regression testing is stated well by Beizer [5], who writes: “It must be possible to precisely recreate the entire test situation or else it may be impossible to resolve some of the nastiest configuration dependent bugs that show up in the field.” Controlled regression testing is further discussed in Reference [46].

For the purpose of regression test selection, we want to identify all tests $t \in T$ that reveal faults in P' – the *fault-revealing* tests. An algorithm that selects every fault-revealing test in T is *safe*. There is no effective procedure that, in general, precisely identifies the fault-revealing tests in T [42]. However, under controlled regression testing, the modification-traversing tests are a superset of the fault-revealing tests [42]. Thus, for controlled regression testing, a regression test selection algorithm that selects all modification-traversing tests is safe. This result is significant, because it supports the following theorem:

Theorem 1: `SelectTests` is safe for controlled regression testing.

Proof: We outline the logic of the proof here. See Reference [42] (pages 77-81) for details.

Let N and C be nodes in G , and let N' and C' be nodes in G' , such that (N, C) is an edge in G labeled L , and (N', C') is an edge in G' also labeled L . N and N' are *similar* for C and C' if and only if C and C' have lexicographically equivalent labels. N and N' are *comparable* if and only if there are traversal trace prefixes ending in N and N' that are similar every step of the way.

The proof initially shows that if test t is modification-traversing for P and P' , then there are nodes N and N' in G and G' , respectively, such that N and N' are comparable, and such that there exists some pair of identically labeled edges (N, C) and (N', C') in G and G' , respectively, where $t \in \text{TestsOnEdge}(N, C)$, and N and N' are not similar for C and C' . The proof then shows that (1) `SelectTests` calls `Compare(N, N')` with every pair of comparable nodes N and N' in G and G' , and that, (2) when called with N and N' , if N and N' are not similar for some pair of nodes C and C' such that (N, C) and (N', C') are on identically labeled edges, `SelectTests` selects all tests on edge (N, C) .

Thus, if t is modification-traversing for P and P' , `SelectTests` selects t . Because the modification-traversing tests form a superset of the fault-revealing tests for controlled regression testing, `SelectTests` is safe for controlled regression testing.

We are also interested in how well `SelectTests` does at omitting tests that cannot reveal faults. For controlled regression testing, tests that are not modification-traversing cannot be fault-revealing; ideally, all such tests should be omitted. The previous section showed, however, that there are cases in which `SelectTests` selects tests that are not modification-traversing. Theorem 2 identifies a necessary condition for `SelectTests` to select such tests: the *multiply-visited-node condition*. The multiply-visited-node condition holds for P and P' if and only if `SelectTests`, run on P and P' , marks some node in P “ X -visited” for more than one node X in the CFG for P' . The theorem is as follows:

Theorem 2: Given procedure P , modified version P' , and test suite T for P , if the multiply-visited-node condition does not hold for P and P' , and `SelectTests` selects test set T' for P' , then every test in T' is modification-traversing for P and P' .

Proof: We outline the logic of the proof here. See Reference [42] (pages 84-85) for details.

The proof proceeds by showing (1) that if `SelectTests` selects test t , then there exists a pair of nodes N and N' in G and G' such that N and N' are comparable, and for some pair of identically labelled edges (N, C) and (N', C') in G and G' where $t \in \text{TestsOnEdge}(N, C)$, N and N' are not similar for C and C' . The proof then assumes that given P and P' for which the multiply-visited-node condition does not hold, `SelectTests` selects some test t that is not modification-traversing, and using step (1), shows that this assumption leads to a contradiction. Thus t must be modification-traversing.

Theorem 2 provides a way to characterize the class of programs and modified versions for which `SelectTests` selects tests that are not modification-traversing. The theorem is significant for two reasons. First, procedures like `pathological` and `pathological'`, which cause the multiply-visited-node condition to hold, are atypical. With some work, additional examples can be constructed; however, all examples located to date have been contrived, and do not represent programs that appear in practice. Second, in empirical studies with “real” programs, we have never found a case in which the multiply-visited-node condition held. Thus, current evidence suggests that despite the existence of examples like `pathological` and `pathological'`, in practice, `SelectTests` selects exactly the tests in T that are modification-traversing for P and P' .

A final theorem is as follows:

Theorem 3: `SelectTests` terminates.

Proof: We outline the logic of the proof here. See reference [42] (page 81) for details.

The proof proceeds by showing that (1) the number of recursive calls made to `Compare` is bounded, and (2) the work required by a call to `Compare` is bounded. Part (1) follows from the fact that every call to `Compare` marks some node N in G N' -visited for some node N' in G' , where the maximum number of node pairs in G and G' is bounded. Part (2) follows from consideration of the boundedness of each statement inside the `Compare` function.

3.1.3 Complexity of `SelectTests`

The running time of `SelectTests` is bounded by the time required to construct CFGs G and G' for P and P' , respectively, plus the number and cost of calls to `Compare`. Let n be the number of statements in P , and n' the number of statements in P' . CFG construction is an $O(n)$ operation [2]. An upper bound on the number of calls to `Compare` is obtained by assuming that `Compare` can be called with each pair of nodes N and N' in G and G' , respectively. Under this assumption, the overall cost of `SelectTests` is $O(n + n' + m(nn'))$, where m is the cost of a call to `Compare`.

Each call to `Compare` results in an examination of at most two edges (“T” and “F”) at line 9, and thus, two calls to `LEquivalent` (line 13). Depending on the results of the `LEquivalent` operation, the call to `Compare` results in either a set union operation (line 14) or an examination of at most two successors of N (line 16). The set union task, implemented as a bit vector operation, has a worst-case cost proportional to the number of tests in T . The `LEquivalent` procedure has a cost that is linear in the number of characters in the statements compared; for practical purposes this size is bounded by a constant (the maximum line length present in the procedure). Thus, m in the above equation is bounded by $k|T|$ for some constant k .

It follows that given a pair of procedures for which CFGs G and G' contain n and n' nodes, respectively, and given a test suite of $|T|$ tests, if `Compare` is called for each pair of nodes (N, N') ($N \in G$ and $N' \in G'$), the running time of `SelectTests` is $O(|T|nn')$.

The assumption that `Compare` may be called for each pair of nodes N and N' from G and G' applies only to procedures P and P' for which the multiply-visited-node condition holds. Program `pathological` of Figure 6 illustrates a case in which that condition holds: in that example, for graphs of 6 and 7 nodes, respectively, the algorithm makes 16 calls to `Compare`. When the multiply-visited-node condition does not hold, however, `Compare` is called at most $\min\{n, n'\}$ times. In these cases, which include all cases observed in practice, `SelectTests` runs in time $O(|T|(\min\{n, n'\}))$.

3.1.4 Improvements to the basic algorithm

The preceding sections presented a simple version of `SelectTests`. There are several ways in which to increase the efficiency or precision of that algorithm [42]. This section discusses two improvements.

Handling variable and type declarations

A change in a variable or type declaration may render a test fault-revealing, even though that test executes no changed program statements other than the declaration. For example, in a Fortran program, changing the type of a variable from “`Real*16`” to “`Real*8`” can cause the program to fail even in the absence of direct alterations to executable code. Our algorithm associates variable and type declarations with “declaration” nodes in CFGs, in the order in which the compiler encounters the declarations, and attaches every test that enters a procedure to the edge that enters the procedure’s declaration node. This approach can be extended to place declaration nodes elsewhere in CFGs (for example, for languages like C that allow declarations in blocks). A similar approach uses a separate declaration node for each variable or type declaration. Using these approaches, our algorithms detect differences between variable and type declarations, and flag tests that may be affected by these differences as modification-traversing.

These approaches have drawbacks: a new, modified, or deleted variable or type declaration may unnecessarily force selection of all tests. If a declaration of variable v changes, the only tests that can be fault-revealing due to such a change (for controlled regression testing) are tests that reach some statement in the executable portion of the procedure or program that contains a reference to the memory associated with v . If a declaration of type τ changes, the only tests that can be fault-revealing due to such a change (for controlled regression testing) are tests that reach some statement in the executable portion of the procedure or program that contains a reference to the memory associated with some variable whose type is based on τ . Our test selection algorithms can be modified to reduce imprecise test selection at declaration changes. One approach requires the algorithm to identify and keep a list of affected variables – variables whose declarations have changed or whose declarations are dependent on changed type definitions. The `LEquivalent` procedure uses this list to detect occurrences of affected variables, and report statements that contain references to the memory locations associated with those variables as modified. The modified algorithm postpones test selection until it locates statements that contain affected references; it then selects only tests that reach those statements.

An alternative representation for case statements

Instead of representing case statements as a series of nested `if-else` statements, we can represent them as a set of choices incident on a single predicate node that has a labeled out edge for each case. This representation yields more precise test selection than the nested `if-else` representation. With minor modifications, `Compare` handles this representation of switches.

A principal difference in this representation is that for switch predicates, the set of labeled out edges may vary from P to P' if cases are added or removed. If a case is added to a switch, then all tests in T that took the default case edge in P can conceivably take the branch to the new case in P' ; thus, `Compare` looks for new edges, and if it detects them, it adds tests that formerly traversed the default edge to T' . If a case is removed from a switch, `Compare` detects the missing labeled edge in G' , and selects all tests that traversed the edge in G .

3.2 Interprocedural test selection

The foregoing examples demonstrate that our regression test selection technique can reduce the number of tests that must be run for single procedures. However, the examples also suggest that when procedures are small and uncomplicated, and their test sets are small, it may be more cost effective to run all tests. This objection is mitigated for interprocedural testing. Test sets for subsystems and complete programs are typically much larger than test sets for single procedures. In this context, the savings that can result from selective retest increases. This section shows how to extend our test selection technique to function interprocedurally – on entire programs or subsystems.

Assume that we can obtain a mapping of procedure names in P to procedure names in P' , that records, for each procedure $\mathcal{P} \in P$, the name \mathcal{P}' of its counterpart in P' . A simple but naive approach to interprocedural regression test selection executes `SelectTests` on every pair of procedures $(\mathcal{P}, \mathcal{P}')$ where \mathcal{P}' is the counterpart of \mathcal{P} . With this simple approach, if a procedure $\mathcal{P} \in P$ is no longer present in P' , tests that used to enter \mathcal{P} are selected at former call sites to \mathcal{P} . Similarly, if a procedure \mathcal{P}' is inserted into P' , tests that enter \mathcal{P}' are selected at the call sites to \mathcal{P}' . Notice that the number of times \mathcal{P} is called from within P is immaterial: trace information reports precisely which tests reach which edges in \mathcal{P} , and a single traversal of \mathcal{P} and \mathcal{P}' suffices to find tests that are modification-traversing for \mathcal{P} and \mathcal{P}' .

By running `SelectTests` on all pairs of corresponding procedures in P and P' , we obtain a test set T' that is safe for controlled regression testing, but fail to take advantage of several opportunities for improvements in efficiency. To describe these opportunities, we refer to the CFGs for the procedures in a program `sys`, shown in Figure 8. Program `sys` contains four procedures: `main` is the entry point to the program, and `A`, `B`, and `C` are invoked when the program runs. Notice that `A` is recursive.

Suppose statement `S1` in `sys` is modified, creating a new version of `sys`, `sys'`. In this case, all tests in T are modification-traversing for `sys` and `sys'`, and are selected when `SelectTests`, called with `main` and `main'`, reaches `S1` in `main`. In this case, there is no need to compare procedures `A`, `B`, and `C` to their counterparts in `sys'`: tests that could reach those procedures and become modification-traversing within them have already been selected. The naive algorithm does unnecessary work for this example.

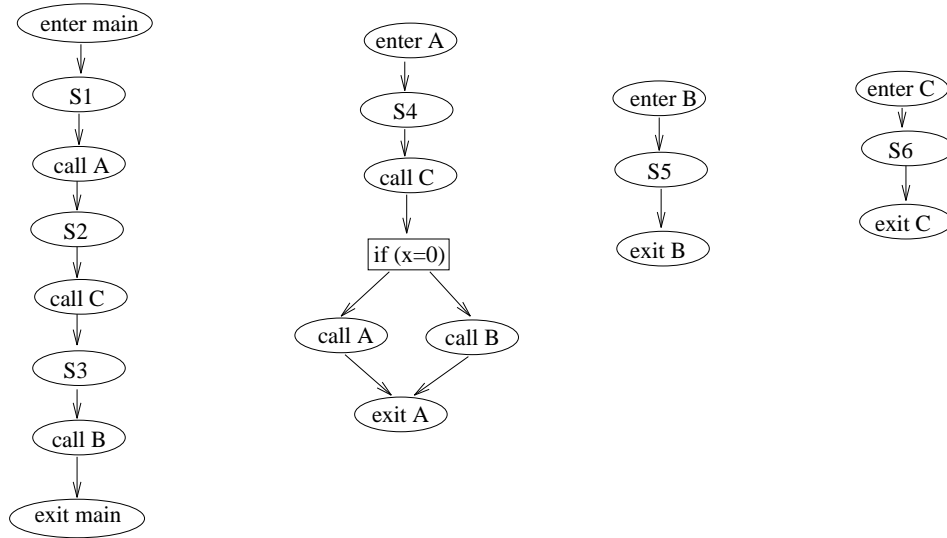


Figure 8: Program `sys`, that illustrates interprocedural test selection effects.

Alternatively, suppose statement $S6$ (and no other statement) in `sys` is modified. In this case, because every test of `sys` that enters `main` reaches the call to `A` in `main`, then reaches the call to `C` in `A`, every test of `sys` becomes modification-traversing in `C` and `C'`. In this case, there is no need to compare `B` with its counterpart in `sys'`, and no need to traverse portions of the CFG for `main` beyond the call to `A`, or portions of the CFG for `A` beyond the call to `C`. Here, too, the naive algorithm does unnecessary work.

These observations motivate an algorithm for interprocedural test selection by CFG traversal that begins by examining the entry procedures for P and P' . When the algorithm reaches call nodes, it immediately enters the graphs for the called procedures if their entry nodes have not previously been visited. When the algorithm completes its processing of a procedure, it records whether tests that enter the procedure can exit it without becoming modification-traversing. On subsequent encounters with calls to such procedures, the algorithm continues its traversal beyond the call nodes only if tests can pass through the procedures without becoming modification-traversing.

3.2.1 The basic interprocedural test selection algorithm

Figure 9 gives algorithm `SelectInterTests`, which selects tests for subsystems or programs. The algorithm uses procedures `SelectTests2` and `Compare2`, which are similar to the intraprocedural test selection procedures `SelectTests` and `Compare`, respectively. The algorithm also keeps data structure *proctable*, that records the name of each procedure encountered in the traversal of the graph for P in a *name* field, and keeps a *status* flag for each procedure that, if defined, can have value “visited” or “selectsall”.

`SelectInterTests` first initializes T' and *proctable* to ϕ , and invokes `SelectTests2` on the entry procedures, P_E and P'_E , of the two programs. Like `SelectTests`, `SelectTests2` takes two procedures P and P' as input, and locates tests that are modification-traversing for those procedures. However, `SelectTests2`

```

algorithm   SelectInterTests(  $\mathcal{P}, \mathcal{P}', P_E, P'_E, T$  ) :  $T'$ 
input       $\mathcal{P}, \mathcal{P}'$ : base and modified versions of a program or subsystem
               $P_E, P'_E$ : entry procedures to  $\mathcal{P}$  and  $\mathcal{P}'$ 
               $T$ : a test set used previously to test  $\mathcal{P}$ 
output     $T'$ : the subset of  $T$  selected for use in regression testing  $\mathcal{P}'$ 
data     proactable: contains fields name and status

1.  begin
2.      $T' = \phi$ 
3.     proactable =  $\phi$ 
4.     SelectTests2(  $P_E, P'_E$  )
5.     return  $T'$ 
6.  end

algorithm   SelectTests2(  $P, P'$  )
input       $P, P'$ : base and modified versions of a procedure

7.  begin
8.     add  $P$  to proactable, setting its status to “visited”
9.     construct  $G$  and  $G'$ , CFGs for  $P$  and  $P'$ , with entry nodes  $E$  and  $E'$ 
10.    Compare2(  $E, E'$  )
11.    if the exit node in  $G$  is not marked “ $S$ -visited” for some node  $S$  in  $G'$ 
12.        set the status flag for  $P$  to “selectsall”
13.    endif
14.  end

procedure   Compare2(  $N, N'$  )
input       $N$  and  $N'$ : nodes in  $G$  and  $G'$ 

15. begin
16.    mark  $N$  “ $N'$ -visited”
17.    for each successor  $C$  of  $N$  in  $G$  do
18.         $L$  = the label on edge (  $N, C$  ) or  $\epsilon$  if the edge is unlabeled
19.         $C'$  = the node in  $G'$  such that (  $N', C'$  ) has label  $L$ 
20.        if  $C$  is not marked “ $C'$ -visited”
21.            if  $\neg$  LEquivalent(  $C, C'$  )
22.                 $T' = T' \cup$  TestsOnEdge( (  $N, C$  ) )
23.            else
24.                for each procedure  $O$  called in  $C$  do
25.                    if  $O \notin$  proactable or status for  $O$  is not “visited” or “selectsall”
26.                        SelectTests2(  $O, O'$  )
27.                    endif
28.                endfor
29.                if any procedures called in  $C$  do not have status flag “selectsall”
30.                    Compare2(  $C, C'$  )
31.                endif
32.            endif
33.        endif
34.    endfor
35.  end

```

Figure 9: Algorithm for interprocedural test selection.

begins by inserting P into *proactable*, and setting the *status* flag for P to “visited”, to indicate that a traversal of P has begun. The procedure then creates CFGs G and G' for P and P' , respectively, and calls Compare2 with the entry nodes of those CFGs. When control returns from Compare2 to SelectTests2, SelectTests2 determines whether the exit node of P was reached during the traversal of Compare2. If not, tests that enter P become modification-traversing on every path through P ; thus, there is no point in visiting nodes beyond

calls to P . To note this fact, `SelectTests2` sets the *status* flag for P in *proctable* to “selectsall”.

`Compare2` is similar to `Compare`, except that when `Compare2` finds that two nodes C and C' have lexicographically equivalent labels, before it invokes itself on those nodes it determines whether C contains any calls. If C contains calls, it may be appropriate to invoke `SelectTests2` on the called procedures. (It is not necessary to check C' for calls: at this point in the algorithm, C and C' have already been compared and found to have lexicographically equivalent labels; thus, C and C' are equivalent in terms of calls.) `Compare` examines the *status* flag for each procedure O called in C ; if *status* is “visited” or “selectsall”, then O has been (or in the former case may, in the case of recursive calls, be in the process of being) traversed, and there is no need to invoke `SelectTests2` on O and O' again. Finally, if any procedure called in C has its *status* flag set to “selectsall”, then all tests through that procedure (and thus all tests through C) have been selected, and there is no need to compare successors of C and C' .

In this fashion, `SelectInterTests` processes pairs of procedures from base and modified programs. By beginning with entry procedures, and processing called procedures only when it reaches calls to those procedures, the algorithm avoids analyzing procedures when calls to those procedures occur only subsequent to code changes. Furthermore, the algorithm avoids traversing portions of graphs that lie beyond calls to procedures through which all tests are modification-traversing.

Unlike the naive algorithm that processes every pair of procedures in P and P' , `SelectInterTests` requires no mapping between procedure names in P and P' , provided P and P' compile and link. If procedure `foo` in P is deleted from P' , statements in P that contain calls to `foo` must be changed for P' . In this case, when `Compare2` reaches a node in G that corresponds to a statement in which `foo` is called, `Compare2` selects all tests that reach the node, and does not invoke `SelectTests2` on `foo`. Thus, it is not possible for `Compare2`, in line 26, to be unable to find a counterpart for `foo` in P' . The cases where `foo` is not present in P but is added to P' , and where `foo` is renamed for P' , are handled similarly.

The improvements to the basic intraprocedural algorithm discussed in Section 3.1.4 also apply to this interprocedural algorithm.

The following example illustrates the use of `SelectInterTests`. Suppose program `sys` of Figure 8 is changed to `sys'` by modification of the code associated with node $S5$ in procedure `B`. (We do not show the graphs for the modified program; to discuss the example we distinguish nodes in the CFG for `sys` from nodes in the CFG for `sys'` by adding primes to them.) Initially, `SelectInterTests` calls `SelectTests2` with `main` and `main'`. `SelectTests2` adds `main` to *proctable* with *status* “visited”, creates the CFGs for the two procedures, then invokes `Compare2` with the entry nodes of those graphs. `Compare2` begins traversing the graphs, and on reaching nodes *call A* and *call A'*, because A is not listed in *proctable*, invokes `SelectTests2` on A and A' . `SelectTests2` adds A to *proctable* with *status* “visited”, then builds the CFGs for the two procedures and begins to traverse them. On reaching calls to C , the algorithm adds C to *proctable* with *status* “visited”, makes CFGs for C and C' , and begins traversing them. The algorithm finds no calls or differences in C and C' and thus, when the call to `Compare` with the entry nodes of C and C' terminates, the algorithm marks the exit node of C “*exit'*-visited”. This marking means that tests entering C and C' pass through unaffected, so `SelectTests2` does not set the *status* flag for C to “selectsall”.

On returning from the call to `SelectTests2` with `C` and `C'`, `Compare` resumes at line 29, finds that the *status* flag is not set, and continues traversing `A` and `A'` by invoking itself with the *call C* node and its counterpart in the graph for `sys'`. The traversal eventually reaches the call to `A` in the `if` predicate: here `Compare2` finds that `A` has *status* “visited” and does not reinvoke `SelectTests2` on `A`, thus handling the recursion. On reaching the call to `B` in the `else` clause of the predicate, `Compare2` invokes `SelectTests2` with `B` and `B'`. This invocation ultimately identifies the differences in `B` and `B'`, and selects tests that reach the modified code. Furthermore, because the code difference prevents `Compare2` from reaching the *exit B* node, when the call to `Compare` with the entry nodes of `B` and `B'` returns, `SelectTests2` sets the *Status* flag for `B` to “selectsall”: all tests that enter `B` are modification-traversing.

On returning from the call to `SelectTests2` for `A` and `A'`, `Compare2` resumes, at line 29, with the *call A* node in `main` and its corresponding node in `main'`. The algorithm traverses the graphs through node `S3` and its counterpart in `main'`. Here, when it examines successors of the nodes, `Compare` notes that the *status* flag for `B` is set to “selectsall”, and thus, does not reinvoke `SelectTests2` on `B` and `B'`. Furthermore, `Compare2` sees that all procedures called in the *call B* node have *status* “selectsall”, and thus does not further traverse the graph for `main`.

3.2.2 Complexity of `SelectInterTests`

`SelectTests` and `SelectInterTests` are of comparable complexity. To see this, suppose P contains p procedures and n statements, suppose c of these n statements contain procedure calls, and suppose P' contains n' statements. Assume that the number of procedure calls in a single statement, and the length of a statement, are bounded by constants k_1 and k_2 , respectively. An upper bound on the running time of `SelectInterTests` is obtained by considering the case where P and P' are identical: in this case the algorithm builds and walks CFGs for every procedure in P and its corresponding procedure in P' .

In this worst case, regardless of the value of p , the time required to build CFGs for all procedures in P is $O(n)$, and the time required to build the CFGs for all procedures in P' is $O(n')$. An upper bound on the number of calls to `Compare2` is obtained by assuming that every node in P must be compared to every node in P' , as might happen if P and P' contain single procedures; in this case, the number of `Compare2` calls is $O(nn')$. Excluding, for the moment, the cost of the *proctable* lookups in lines 24-31, a call to `Compare2` requires the same amount of work as a call to `Compare`, namely, $k_2|T|$. Regardless of the number of calls to `Compare`, lines 24-31, over the course of a complete execution of `SelectInterTests`, require at most k_1c table lookups on a table that contains at most p entries; using a naive table lookup algorithm, the lines require $O(cp)$ string comparisons, where the time for each comparison is bounded by k_2 . Thus, the time required by `SelectInterTests` in the worst case is $O(n + n' + |T|nn' + cp)$, where cp is bounded above by k_1n^2 . In practice, however, we expect a lower bound on execution time. When the multiply-visited-node condition does not hold, the expression $|T|nn'$ becomes $|T|(\min\{n, n'\})$. Furthermore, by using an efficient hashing scheme to implement *proctable*, we can reduce the $O(cp)$ table lookup time to (expected time) $O(c)$, which is $O(n)$.

4 Evaluations of the Algorithms

This section evaluates our algorithms and compares them to other regression test selection techniques. Section 4.1 presents an analytical evaluation and comparison; Section 4.2 presents empirical results.

4.1 Analytical Evaluation and Comparison

Although some regression test selection techniques select tests based on information collected from program specifications [31, 52], most techniques, including ours, select tests based on information about the code of the program and the modified version [1, 4, 6, 8, 11, 13, 14, 15, 18, 23, 27, 28, 31, 39, 43, 45, 50, 51, 54, 57]. These code-based techniques pursue various goals. Coverage techniques emphasize the use of structural coverage criteria; they attempt to locate program components, such as statements or definition-use pairs, that have been modified or may be affected by modifications, and select tests from T that exercise those components. Minimization techniques work like coverage techniques, but select minimal sets of tests through modified or affected program components. Safe techniques, in contrast, emphasize selection of tests from T that can reveal faults in a modified program.

To provide a mechanism for evaluating and comparing regression test selection techniques, we developed an analysis framework that consists of four categories: inclusiveness, precision, efficiency, and generality. *Inclusiveness* measures the extent to which a technique selects tests from T that reveal faults in a modified program; a 100% inclusive technique is safe. *Precision* measures the extent to which a technique omits tests in T that cannot reveal faults in a modified program. *Efficiency* measures the space and time requirements of a technique, focusing on critical phase costs. *Generality* measures the ability of a technique to function in a practical and sufficiently wide range of situations. We have used our framework to compare and evaluate all code-based regression test selection techniques that we have found descriptions of in the literature. We have also used our framework to evaluate our technique and compare it to other techniques. Reference [46] presents this framework and evaluation in detail; this section summarizes results reported in that work.

Inclusiveness.

Our test selection algorithms are safe for controlled regression testing. Only three other techniques [11, 23, 27] can make this claim. These three techniques each depend for their safety upon the same assumptions on which our algorithms depend. Thus, with existing regression test selection techniques, safe test selection is possible only for controlled regression testing.

Precision.

Our test selection algorithms are not 100% precise. However, because the problem of precisely identifying the tests that are fault-revealing for a program and its modified version is undecidable, we know that we cannot have an algorithm that is both safe and 100% precise.

Nevertheless, our algorithms are the most precise safe algorithms currently available. For cases where the multiply-visited-node condition does not hold (we believe this includes all practical cases), our technique selects *exactly* the modification-traversing tests, whereas other safe techniques select the modification-traversing

tests, along with tests that are not modification-traversing. In cases where the multiply-visited-node condition does hold, we can prove that `SelectTests` and `SelectInterTests` are more precise than two of the other three safe test selection techniques [11, 23], and we have strong evidence to suggest that our algorithms are more precise than the third safe technique [27].

Efficiency.

As discussed previously, our algorithms run in time $O(|T|nn')$ for procedures or programs of n and n' statements, and test set size $|T|$. This is an improvement over the efficiency of two of the other safe techniques [23, 27]. Moreover, we expect our algorithms to run in time $O(|T|(\min\{n, n'\}))$ in practice – a bound comparable to the worst-case run time of the third safe technique [11]. Our algorithms are also as efficient as, if not more efficient than, existing non-safe algorithms. Our algorithms are fully automatable. Furthermore, much of the work required by our technique, such as construction of CFGs for P and collection of test history information, can be completed during the preliminary regression testing phase. Unlike most other algorithms, and all safe algorithms, our algorithms do not require prior computation of a mapping between components of programs or procedures and their modified versions; instead, they locate changed code as they proceed, and in the presence of significant changes avoid unnecessary comparison.

Generality.

Our algorithms apply to procedural languages generally, because we can obtain the required graphs and test history information for all such languages. Unlike many other techniques, our technique supports both intraprocedural and interprocedural test selection. Also unlike several techniques, our technique handles all types of program modifications, and handles multiple modifications in a single application of the algorithms.

4.2 Empirical Evaluation

Researchers wishing to experiment with software testing techniques face several difficulties – among them the problem of locating suitable experimental subjects. The subjects for testing experimentation include both software and test suites; for regression testing experimentation, multiple versions of the software are also required. Obtaining such subjects is a non-trivial task. Free software, often in multiple versions, is readily accessible, but free software is not typically equipped with test suites. Commercial software vendors, who are more likely to maintain established test suites, are often reluctant to release their source code and test suites to researchers. Even when suitable experimental subjects are available, prototype testing tools may not be robust enough to operate on those subjects, and the time required to ensure adequate robustness may be prohibitive.

Given adequate experimental subjects and sufficiently robust prototypes, we may still question the generalizability of experimental results derived using those subjects and prototypes. Experimental results obtained in the medical sciences generalize due to the fact that a carefully chosen subset of a population of subjects typically represents a fair (i.e., normally distributed) cross-section of that population. As Weyuker states, however, when the subject population is the universe of software systems, we do not know what it means to

<i>Program</i>	<i>Procedures</i>	<i>LOC</i>	<i>Nodes</i>	<i>Edges</i>	<i>Versions</i>	<i>Tests</i>	<i>Description</i>
replace	21	512	383	432	32	5542	pattern replacement
usl.123	20	472	303	364	7	4056	lexical analyzer
totinfo	16	440	249	271	23	1054	information measure
usl.128	21	399	355	409	10	4071	lexical analyzer
schedule2	16	301	219	243	10	2680	priority scheduler
schedule1	18	292	219	232	9	2650	priority scheduler
tcas	8	141	89	87	41	1578	altitude separation

Table 1: The 7 subject programs used for Study 1.

select a fair cross-section of that population, nor do we know what it means to select a fair cross-section of the universe of modified versions or test suites for software [53]. Weyuker concludes that software engineers typically perform “empirical studies” rather than experiments. She insists, however, that such studies offer insight, and are valuable tools in understanding the topic studied. We agree with Weyuker; hence, this section outlines the results of empirical studies.

To empirically evaluate our regression test selection technique, we implemented the `SelectTests` and `SelectInterTests` algorithms as tools, which we call “DejaVu1” and “DejaVu2”, respectively. Our implementations select tests for programs written in C. We implemented tools and conducted empirical studies on a Sun Microsystems SPARCstation 10 with 128MB of virtual memory.²

4.2.1 Study 1: Intraprocedural and Interprocedural Test Selection

Our first study investigated the efficacy of `DejaVu1` and `DejaVu2` on a set of small, but non-trivial, real subject programs. The primary objective of this study was to empirically investigate the extent to which our algorithms could reduce the cost of regression testing at the intraprocedural and interprocedural levels.

Subjects

Hutchins, *et al.* [26] report the results of an experiment on the effectiveness of dataflow- and controlflow-based test adequacy criteria. To conduct their study, the authors obtained seven C programs, that ranged in size from 141 to 512 lines of code, and contained between 8 and 21 procedures. They constructed 132 versions of these programs, and created large *test pools* for the programs. The authors made these programs, versions, and test suites available to us. We refer to their experiment as the “Siemens study”, and to the experimental programs as the “Siemens programs”. Table 1 describes the Siemens programs.³

To study our intraprocedural test selection algorithms, we considered each procedure in the Siemens programs that had been modified for one or more versions of a program. Table 2 lists these procedures.

Because the Siemens study addressed error detection capabilities, the study employed faulty versions of base programs. For our purposes, we shall consider these faulty versions as ill-fated attempts to create

²SPARCstation is a trademark of Sun Microsystems, Inc.

³There are a few differences between the numbers reported in Table 1 and the numbers reported in Reference [26]. Hutchins *et al.* report 39 versions of `tcas`; their distribution to us contained 41. Also, the numbers of tests in the test pools we obtained from their distribution differed slightly from the numbers they reported: in the two most extreme cases, for example, we found 16 more tests (`tcas`), and 36 fewer tests (`usl.123`). These difference amount to less than 1% of total test pool sizes and do not affect the results of this study.

<i>Procedure</i>	<i>LOC</i>	<i>Versions</i>	<i>Tests</i>	<i>Procedure</i>	<i>LOC</i>	<i>Versions</i>	<i>Tests</i>
1. infotbl	86	7	963	22. locate	20	1	1745
2. gettoken	77	6	4070	23. gser	19	4	297
3. main	71	8	1054	24. isnumconstant	17	1	4056
4. schedule1main	66	1	2650	25. isstrconstant	16	2	4056
5. omatch	58	5	4177	26. noncrossingbdescend	16	4	876
6. makepat	58	2	5520	27. noncrossingbclimb	16	4	876
7. gettoken2	51	4	4056	28. finish	15	1	2310
8. dodash	37	8	2891	29. findnth	13	2	1831
9. gcf	29	3	668	30. change	12	1	4658
10. getcommand	28	3	2649	31. putend	12	1	2642
11. numericcase	27	1	1322	32. upgradeprio	11	1	1867
12. upgradeprocessprio	25	5	1820	33. getline	9	2	4658
13. getprocess	25	2	2649	34. enqueue	9	10	2642
14. esc	24	3	5214	35. initialize	7	8	1578
15. lgamma	24	1	729	36. inpatset	6	1	4177
16. altseptest	24	13	1578	37. inset2	5	7	1367
17. istokenend	23	3	3938	38. ownbelowthreat	4	3	604
18. newjob	23	1	2642	39. ownabovethreat	4	2	582
19. subline	23	3	4177	40. inhibitbiasedclimb	3	10	886
20. getcc1	21	1	2891	41. alim	3	2	564
21. unblockprocess	20	2	2028				

Table 2: The 41 subject procedures used for Study 1. We number the procedures to facilitate subsequent references to them.

modified versions of the base programs. The use of faulty versions also lets us make observations about error detection during regression testing.

Hutchins *et al.* [26] describe the process used by the Siemens researchers to construct test suites and faulty program versions – we paraphrase that description here. The Siemens researchers created faulty versions of base programs by manually seeding faults into those programs. Most faults involve single line changes; a few involve multiple changes. The researchers required that the faults be neither too easy nor too difficult to detect (a requirement that was quantified by insisting that each fault be detectable by at least 3, and at most 350, tests in the test pool), and that the faults model “realistic” faults. Ten people performed the fault seeding, working for the most part without knowledge of each other’s work.

The Siemens researchers created test pools “according to good testing practices, based on the tester’s understanding of the program’s functionality and knowledge of ...the code.” The researchers initially generated tests using the *category partition method* and the Siemens TSL (Test Specification Language) tool [3, 38]; they then added additional tests to the test suites to ensure that each coverage unit (statement, edge, and du-pair) in the base program and versions was exercised by at least 30 tests.

The Siemens subjects present some disadvantages for our study, because they employ only faulty modifications, use constructed faults rather than real ones, and use only faults that yield meaningful detection rates. However, the Siemens subjects also have considerable advantages. The fact that the Siemens researchers made the subjects available to us is an obvious advantage. Also, the seeded faults in the programs do model real faults. Furthermore, the source code for the base programs and versions is standard C, amenable to

analysis and instrumentation by our prototype tools. Finally, the Siemens subjects have previously served as a basis for published empirical results.

Empirical Procedure

To obtain our empirical results, we initially used an analysis tool [17] on the base programs and modified versions to create control flow graphs for those versions. We then ran a code instrumentation tool to generate instrumented versions of the base programs. For each base program, we ran all tests for that program on the instrumented version of the program, and collected test history information for those tests. We then ran `Dejavu1`, our implementation of `SelectTests`, on each procedure from a Siemens program that had been modified in one or more modified versions, with each modified version of the base version of that procedure. We also ran `Dejavu2`, our implementation of `SelectInterTests`, on each Siemens base program, with each modified version of that base program. Execution timings were obtained during off-peak hours on a restricted machine; our testing processes were the only user processes active on the machine. We repeated each experiment five times for each (base program, modified program) or (base procedure, modified procedure) pair, and averaged our results over these runs; all timings reported for this study list these average results. In our experimentation, we used controlled regression testing.

Results

Figure 10 shows the test selection results for our intraprocedural test selection tool, `DejaVu1`, in Study 1. The graph shows, for each of the 41 base versions of the Siemens procedures, the percentage of tests selected by `DejaVu1`, on average, over the set of modified versions of that base procedure. The graph shows that for this study, intraprocedural test selection reduced the size of selected test sets in some cases, but the overall savings were not dramatic. In fact, for 21 of the 41 subject procedures, `DejaVu1` always selected 100% of the tests for modified versions of the procedures. `DejaVu1` reduced test sets by more than 50% on average in only five cases. These results are discussed in greater detail later in this section.

Figure 11 shows the test selection results for our interprocedural test selection tool, `DejaVu2`, in Study 1. The graph shows, for each of the 7 base versions of the Siemens programs, the percentage of tests selected by `DejaVu2`, on average, over the set of modified versions of that base program. The average test set selected by `DejaVu2` for a modified version was 55.6% as large as the test set required by the retest all approach. In other words, `DejaVu2` averaged a savings in test set size of 44.4%. Over the various base programs, the test sets selected by `DejaVu2` ranged from 43.3% (on `replace`) to 93.6% (on `schedule2`) of the size of the total test sets for those programs.

The fact that our algorithms reduce the number of tests required to retest modified programs does not by itself indicate the worth of the algorithms. If ten hours of analysis are required to save one hour of testing, this might not be of benefit – unless the ten hours are fully automated, the hour saved is an hour of human time, and we can spare the ten hours. We would like to show that the time saved in not having to run tests exceeds the time spent analyzing programs and selecting test suites. Toward this end, Figure 12 shows some timings. For each of the seven base programs, the figure shows three columns: (1) the average time required

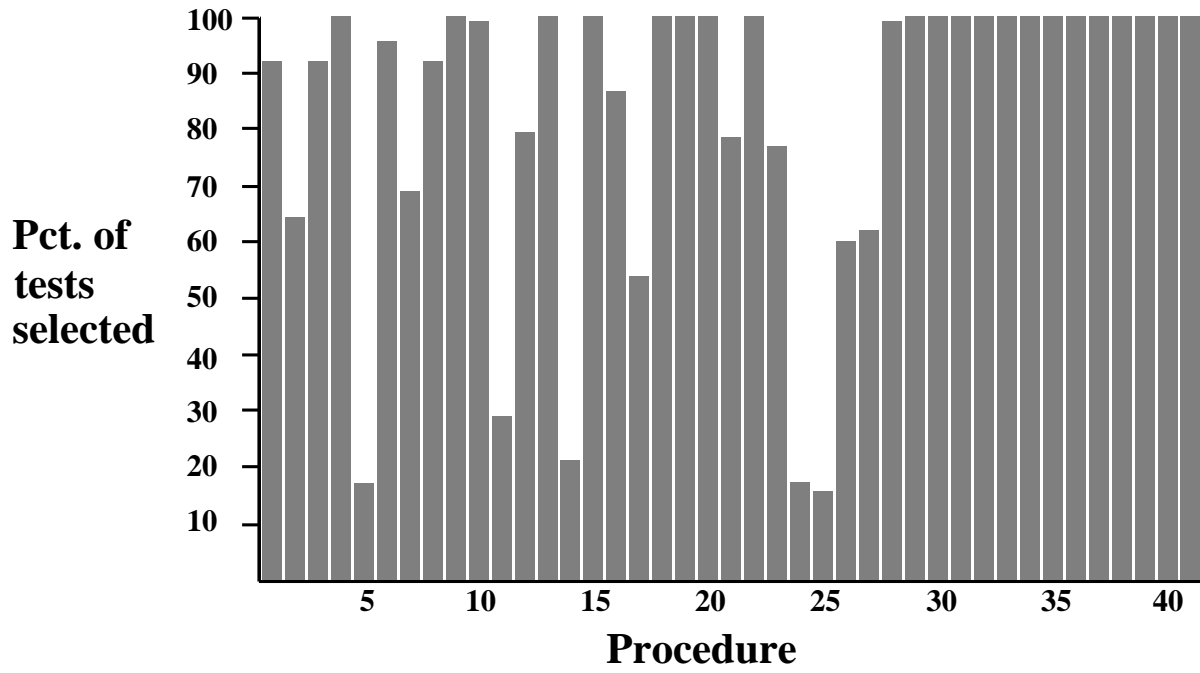


Figure 10: Intraprocedural test selection for Study 1.

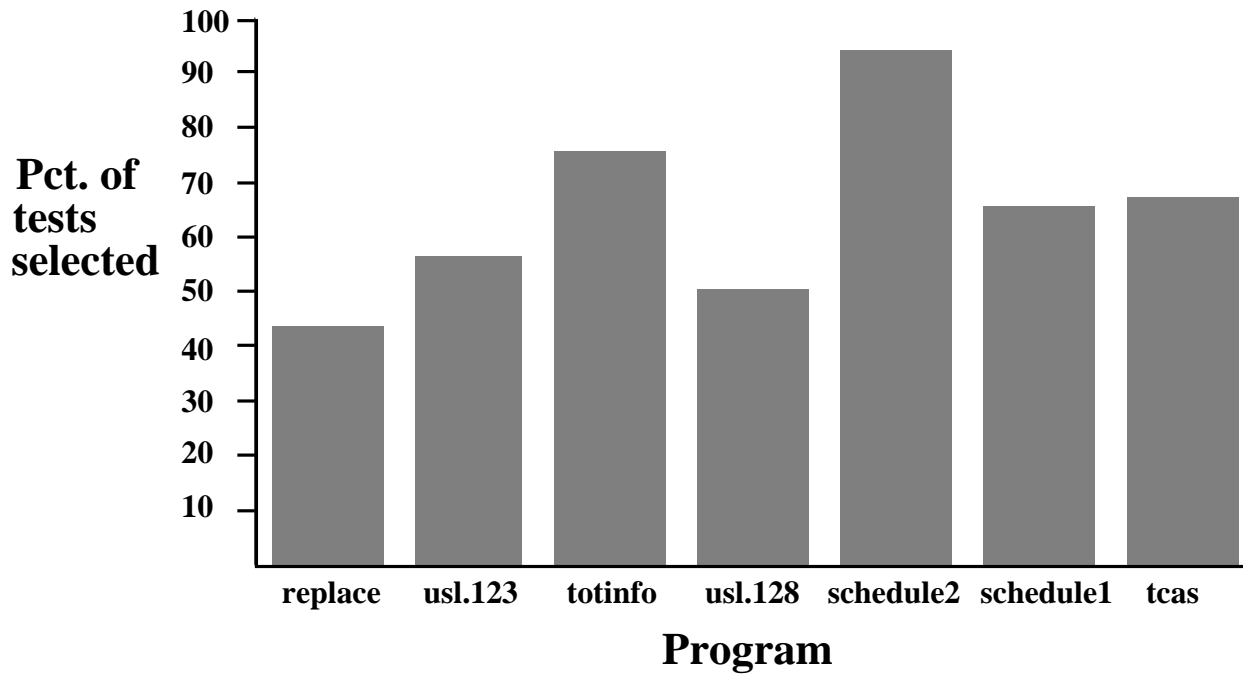


Figure 11: Interprocedural test selection for Study 1.

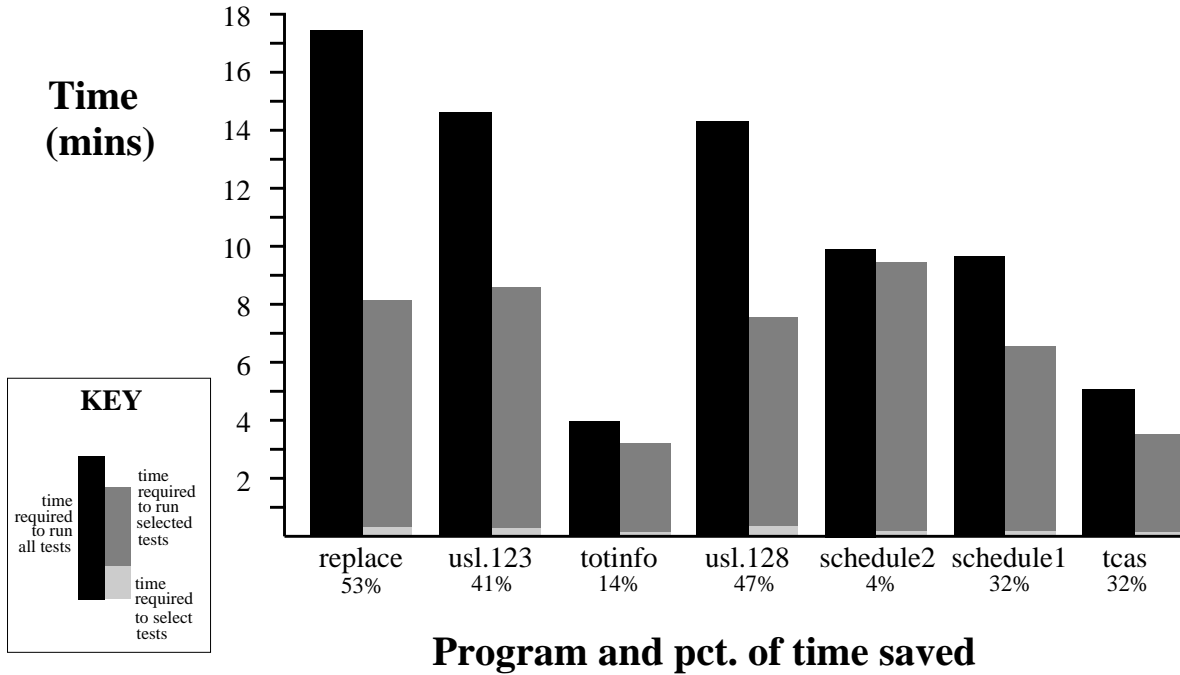


Figure 12: Interprocedural timings for Study 1.

to run all tests on the modified version of the program (darkest column); (2) the time required to perform analysis, on average, of the base and modified versions of the program (lightest column); and (3) the time required, on average, to run the selected tests on the modified version of the program. Because the goal is to compare the time required to run all tests to the time required to select and run a subset of the tests, columns (2) and (3) are “stacked” and placed alongside column (1). Times are shown in minutes. Under each program name, the percentage of total time saved by using test selection is displayed. Note that in this study, both the execution of tests and the validation of test results were fully automated.

As the figure indicates, the cost of our algorithms in terms of time is negligible – it never exceeds 22 seconds. These measurements include the cost of building CFGs for both the base and modified program version; however, the CFG for the base version could have been computed and stored, like test history information, during the preliminary phase of testing, further reducing the critical period cost of the algorithm. The figure also shows that in all cases, DeJaVu2 produced a savings in overall regression testing time. In the worst case, for `schedule2`, DeJaVu2 saved only 28 seconds, or 4%, of total effort. In the best case, for `replace`, DeJaVu2 saved 9 minutes and 17 seconds, or 53%, of total effort.

Savings of a few minutes and seconds, such as those achieved in this study, may be unimportant. In practice, however, regression testing can require hours, days, or even weeks of effort, and much of this effort may be human-intensive. If results such as those demonstrated by this study scale up, a savings of even 10% may matter, and a savings of 50% may be a big win. In fact, we conjecture that the savings obtainable from DeJaVu2 increase, on average, as larger programs are used as subjects.

<i>Program</i>	<i>Procedures</i>	<i>LOC</i>	<i>Nodes</i>	<i>Edges</i>	<i>Versions</i>	<i>Tests</i>	<i>Description</i>
player	766	49316	35000	41000	5	1035	transaction manager

Table 3: The subject program used for Study 2.

<i>Version</i>	<i>Functions Modified</i>	<i>Lines of Code Changed</i>
1	3	114
2	2	55
3	11	726
4	11	62
5	42	221

Table 4: The 5 modified versions of `player`.

4.2.2 Study 2: Interprocedural Test Selection on a Larger Scale

The primary objective of our second study was to empirically investigate our conjecture that test selection may offer greater benefits for larger programs than for small programs, by applying our technique to a larger subject.

Subjects

For our second study, we obtained a program, `player`, that is one of the subsystems of an internet-based game called `Empire`. The `player` executable is essentially a transaction manager; its main routine consists of initialization code, followed by a five-statement event loop that waits for receipt of a user command, and upon receiving one, calls a routine that processes the command (possibly invoking many more routines to do so), then waits to receive the next command. The loop, and the program, terminate when a user issues a quit command. Since its initial encoding in 1986, `Empire` has been rewritten many times; many modified versions of the program have been created. Most of these versions involve modifications to `player`.

For our study, we located a base version of `player` for which five distinct modified versions were available. Table 3 presents some statistics about the base version. As the table indicates, the version contains 766 C functions and 49,316 lines of code, excluding blank lines and lines that contain only comments. The CFGs for the functions contained approximately 35,000 nodes and 41,000 edges in total (these numbers are approximate for reasons explained later). Table 4 describes the versions of `player` that we used for our study.

There were no test suites available for `player`. To construct a realistic test suite, we used the `Empire` information files, which describe the commands that are recognized by the `player` executable and discuss parameters and special side-effects for each command. We treated the information files as informal specifications; for each command, we used its information file to construct versions of the command that exercise all parameters and special features, and test erroneous parameters and conditions. This process yielded a test suite of 1035 functional tests. We believe that this test suite is typical of the sorts of functional test suites designed in practice for large software systems.

The `player` program is a suitable subject for several reasons. First, the program is part of an existing software system that has a long history of maintenance at the hands of numerous coders; in this respect,

the system is similar to many existing commercial software systems. Second, as a transaction manager, the `player` program is representative of a large class of software systems that receive and process interactive user commands. (Other examples of systems in this class include database management systems, operating systems, menu-driven systems, and computer-aided drafting systems, to name just a few.) Third, we were able to locate several real modified versions of one base version of the program. Fourth, although the absence of established test suites for the program was a disadvantage, the user documentation provided a code-independent means for generating functional tests in a realistic fashion. Finally, although not huge, the program is not trivial.

Empirical Procedure

Due to limitations in our prototype analysis tools, we could analyze only 85% of the procedures in the `player` program; thus, we could not instrument, or run our `DejaVu` implementations on, 15% of the procedures. However, we were able to simulate the test selection effects of `DejaVu` on `player`. Our simulation determines exactly the numbers of tests selected and omitted by our algorithm in practice; we were also able to determine exactly the time required to run all tests, or all selected tests. We could not obtain precise results of the time required to build CFGs for the versions and perform test selection on those graphs. Instead, we estimated those times: we determined the time required to build CFGs and run `DejaVu1` on 85% of the code, and multiplied those times by 1.176 to determine the time required for 100% of the code.

Results

Figure 13 shows the test selection results for our interprocedural test selection algorithm for the modified versions of `player`. The graph shows, for each of the five modified versions, the percentage of tests that our algorithm selects. As the results indicate, on average over the five versions, our algorithm selects 4.8% of the tests in the existing test suite. In other words, on average, the algorithm reduces the number of tests that must be run by over 95%.

Figure 14 shows timings for this study. Like the graph in Figure 12, for each of the five modified versions, the graph shows three columns: (1) the time required to run all tests on the modified version of the program (darkest column); (2) the estimated time required to perform analysis of the base and modified versions of the program (lightest column); and (3) the time required to run the selected tests on the modified version of the program. Times are shown in hours. The graph shows that in all cases, our algorithm produced a savings in overall regression testing time. This savings ranged from 4 hours and 39 minutes (82% of total effort) to 5 hours and 37 minutes (93% of total effort).

Our estimate of analysis time projects a cost of at most 25 minutes; however, this estimate computes the cost of building CFGs for every procedure in both the base and modified program version during the critical period, and walking all of those CFGs completely. In practice, CFGs for procedures in the base version could be built during the preliminary phase, and CFGs for procedures in modified versions could be built on demand, lowering the analysis cost. Furthermore, test timings consider only the cost of running the tests, because we were not able to automate the validation of results for these tests. In practice, the time required to run tests would be much larger than the times shown, and the resulting savings would increase.

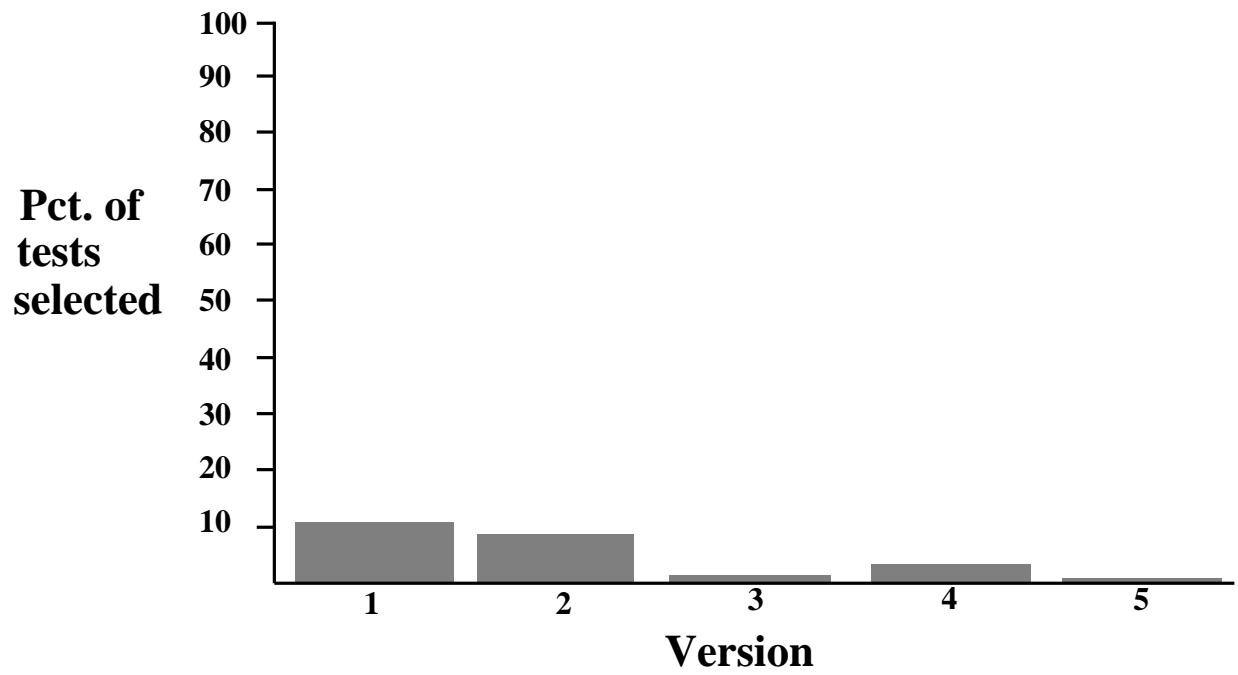


Figure 13: Test selection statistics for Study 2.

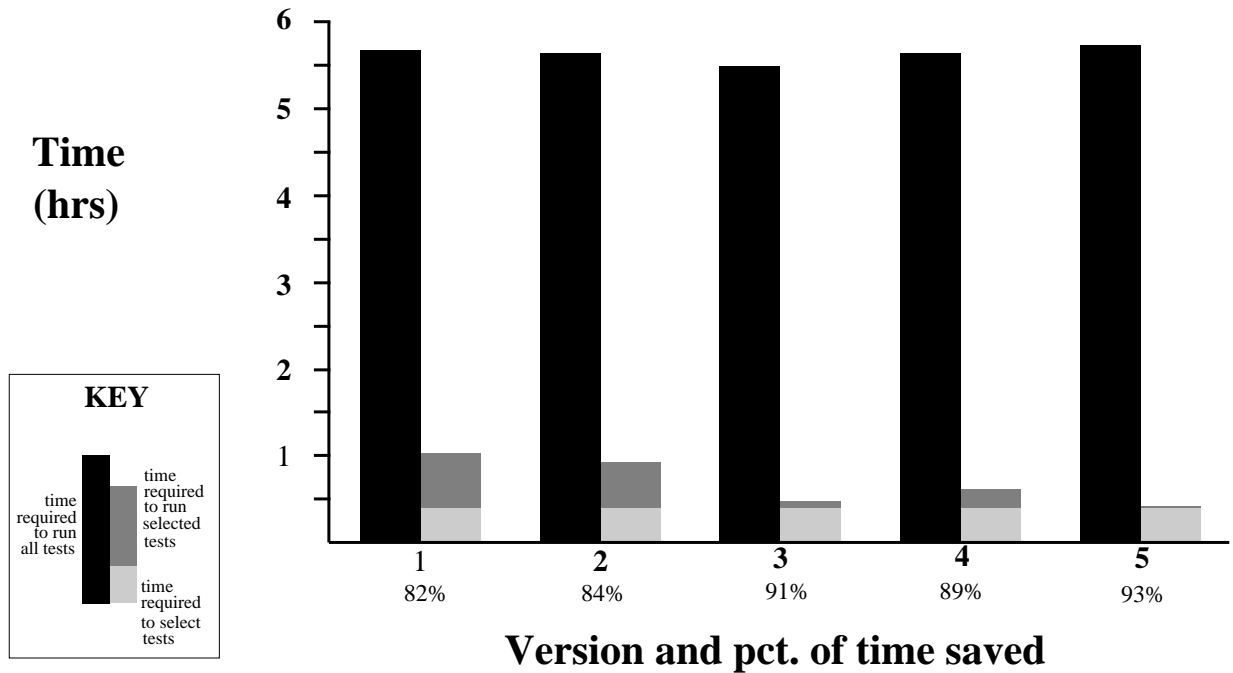


Figure 14: Timings for Study 2.

4.2.3 Additional discoveries and discussion

Our studies yielded several additional discoveries.

First, in Section 3 we saw that our algorithms can select tests that are not modification-traversing for P and P' , but only when the multiply-visited-node condition holds. In our experiments, we never encountered a case where that condition held. These results support our belief that in practice, our algorithms will not select non-modification-traversing tests.

Second, although we have reported results as averages over sets of modified programs, it is interesting to examine the behavior of our algorithms for individual cases.⁴ Consider, for example, the results for program `replace`. The test pool for `replace` contains 5542 tests, of which DeJaVu2 selects, on average, 2399 (43.3%). However, over the 32 modified versions of `replace`, the selected test sets ranged in size from 52 to 5542 tests, with no size range predominant; the standard deviation in the sizes of the selected test sets was 1611.5. In contrast, for `schedule2`, DeJaVu2 selects, on average, 2508 (93.6%) of the program’s pool of 2680 tests, with a standard deviation of 253.5. On eight of the ten modified versions of `schedule2`, DeJaVu2 selects at least 90% of the existing tests.

Given this range of variance, we would like to identify the factors that influence the success of test selection. This could help us determine when test selection is likely to be successful, and also support conclusions about ways in which to build programs and test suites that are “regression testable.” On examining our experimental subjects and test suites, we determined that the effectiveness of test selection was influenced by three factors: the structure of P , the location of modifications in P' , and the extent of the code coverage achieved by tests in T . Although these factors can interact, they may also operate independently.

Finally, as an interesting side-effect, our first study provides data about the fault-revealing capabilities of regression test selection techniques. Study 1 involved faulty modified program versions, for which a very small percentage of tests are fault-revealing. For example, only 302 of the 5542 tests for version 26 of `replace` are fault-revealing. DeJaVu2 finds that 1012 of the 5542 tests are modification-traversing and selects them. A *minimization* test selection technique that selects one test from the set of tests that cover modified code has only a 29.8% chance of selecting one of the 302 fault-revealing tests. Next, consider version 19 of `replace`. Although 4658 of the 5542 tests of `replace` are modification-traversing for this version, only 3 of these tests are fault-revealing for the version. A minimization test selection technique that selects only one of the 4658 tests that cover this modification has only a .064% chance of selecting a test that exposes the fault. In either case, DeJaVu2 guarantees that the fault is exposed. Study 1 contains many other comparable cases.

It would not be fair, on the basis of Study 1 alone, to draw general conclusions about the relative fault detection abilities of minimization and safe test selection techniques, because the Siemens study deliberately restricted modifications to those that contained faults that were neither too easy nor too difficult to detect. Nevertheless, the Siemens study did employ faults that are representative of real faults, so we expect that cases such as the two discussed above can arise in practice. Thus, these results give us good reason to question the efficacy of minimization test selection techniques where fault detection is concerned.

⁴Reference [42] lists results for all programs and modified versions individually.

4.2.4 Summary of empirical results, and limitations of the studies

The major conclusions derived from our empirical studies can be summarized as follows:

- Our algorithms can reduce the time required to regression test modified software, even when the cost of the analysis performed to select tests is considered.
- Interprocedural test selection can offer greater savings than intraprocedural test selection.
- Regression test selection algorithms can yield greater savings when applied to large, complex programs than when applied to small, simple programs.
- There exist programs, modified versions, and test suites for which test selection offers little in the way of savings.
- The factors that affect the effectiveness of test selection techniques include the structure of programs, the nature of the modifications made to those programs, and the type of coverage attained by tests.
- Our belief that in practice, programs contain no multiply-visited-nodes, remains plausible.
- Minimization techniques for test selection can be much less effective than safe techniques for revealing faults.

Our studies have the following limitations:

- We have studied only a small sample of the universe of possible programs, modified programs, and test suites. We cannot claim that this sample is normally distributed. We believe, however, that the subjects of our studies are representative of significant classes of programs that occur in practice.
- Both of our studies required some constructed artifacts: Study 1 used constructed modified versions and tests, and Study 2 used constructed tests. In both cases, however, efforts were made to ensure that constructed artifacts were representative of real counterparts.
- Due to limitations in our program analysis tools, our second study required an estimation of analysis times. However, we believe that our estimates understate the analysis time required by our technique.

5 Conclusions and Future Work

This work is important for two reasons. The first reason is economic. The cost of software maintenance dominates the overall cost of software [9, 34, 35]. Moreover, the cost of maintenance, measured in terms of the percentage of software budget spent on maintenance, is increasing [5, 37, 48]. Because regression testing constitutes a significant percentage of maintenance costs [5, 9, 29], improvements in regression testing processes can significantly lower the overall cost of software.

The second reason for the importance of this work involves software quality. Regression testing is an important method both for building confidence in modified software and for increasing its reliability. At

present, however, regression testing is often overlooked or inadequately performed: either the testing of new features, or the revalidation of old features, or both, are sacrificed [5]. In one survey of 118 software development organizations, only 12 percent of these organizations were found to have mechanisms for assuring some level of adequacy in their regression testing [36]. Without adequate regression testing, the quality and reliability of a software system decrease over the system's lifetime. Practical, effective selective retest techniques promote software quality.

There are several promising directions for future work in this area. First, while the empirical results reported in this paper are encouraging, they are also preliminary; further empirical studies would be useful. Second, our work has focused on the problem of selecting tests from an existing test suite. An equally important problem is that of ensuring that code modified or affected by modifications is adequately tested. Future work should consider the extension of this technique to help identify the need for new tests. Third, our research has revealed that the size of the test sets our algorithms select may vary significantly as a function of program structure, type of modifications, and test suite design. Further research could investigate correlations between these three factors and the related issues of regression testability and test suite design. Fourth, our technique, and all existing safe regression test selection techniques, are safe only for controlled regression testing. Further research could investigate ways to make controlled regression testing possible in situations where it is difficult to attain. Finally, it is not the case that our algorithms function only for controlled regression testing: it is simply that like all other regression test selection algorithms, they are not safe in the absence of controlled regression testing. When we cannot employ controlled regression testing, and cannot guarantee safety, our algorithms may still select useful test suites. When the testing budget is limited, and we *must* choose a subset of T , modification-traversing tests such as those selected by our algorithm may be better candidates for execution than non-modification-traversing tests. Empirical studies could investigate this possibility further.

Acknowledgements

S.S. Ravi made several helpful suggestions, especially in regard to the proofs and complexity analyses. We also thank the anonymous reviewers for contributions that substantially improved the presentation of the work. This work was partially supported by a grant from Microsoft, Inc., and by the National Science Foundation under Grant CCR-9357811 to Clemson University and The Ohio State University. A preliminary version of this work appeared in Reference [43].

References

- [1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 348–357. IEEE Computer Society Press, September 1993.
- [2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [3] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the Third Symposium on Software Testing, Analysis, and Verification*, pages 210–218. ACM Press, December 1989.

- [4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*. ACM Press, January 1993.
- [5] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.
- [6] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 352–361, Los Alamitos, CA, October 1988.
- [7] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50. IEEE Computer Society Press, November 1992.
- [8] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the Conference on Software Maintenance - 1995*. IEEE Computer Society Press, October 1995.
- [9] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, December 1976.
- [10] P.A. Brown and D. Hoffman. The application of module regression testing at TRIUMF. *Nuclear Instruments and Methods in Physics Research, Section A*, A293(1-2):377–381, August 1990.
- [11] Y.F. Chen, D.S. Rosenblum, and K.P. Vo. TestTube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222. IEEE Computer Society Press, May 1994.
- [12] T. Dogsa and I. Rozman. CAMOTE - computer aided module testing and design environment. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 404–408. IEEE Computer Society Press, October 1988.
- [13] K.F. Fischer. A test case selection method for the validation of software maintenance modifications. In *Proceedings of COMPSAC '77*, pages 421–426. IEEE Computer Society Press, November 1977.
- [14] K.F. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6. IEEE Computer Society Press, November 1981.
- [15] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 299–308. IEEE Computer Society Press, November 1992.
- [16] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.
- [17] M.J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: a system for the development of program-analysis-based tools. In *Proceedings of the 33rd Annual Southeast Conference*, pages 110–119. ACM Press, March 1995.
- [18] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 362–367. IEEE Computer Society Press, October 1988.
- [19] M.J. Harrold and M.L. Soffa. An incremental data flow testing tool. In *Proceedings of the Sixth International Conference on Testing Computer Software*. Frontier Technologies, May 1989.
- [20] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167. ACM Press, December 1989.
- [21] J. Hartmann and D.J. Robson. Revalidation during the software maintenance phase. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 70–79. IEEE Computer Society Press, October 1989.
- [22] J. Hartmann and D.J. Robson. RETEST - development of a selective revalidation prototype environment for use in software maintenance. In *Proceedings of the Twenty-Third Hawaii International Conference on System Sciences*, pages 92–101, January 1990.
- [23] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.
- [24] D. Hoffman. A CASE study in module testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 100–105. IEEE Computer Society Press, October 1989.
- [25] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the Third Workshop on Software Testing, Analysis, and Verification*, pages 97–102. ACM Press, December 1989.
- [26] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200. IEEE Computer Society Press, May 1994.

- [27] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 282–290. IEEE Computer Society Press, November 1992.
- [28] J.A.N. Lee and X. He. A Methodology for Test Selection. *The Journal of Systems and Software*, 13(1):177–185, September 1990.
- [29] H.K.N. Leung and L. White. Insights Into Regression Testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 60–69. IEEE Computer Society Press, October 1989.
- [30] H.K.N. Leung and L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance: Research and Practice*, 2:209–222, December 1990.
- [31] H.K.N. Leung and L.J. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance - 1990*, pages 290–300. IEEE Computer Society Press, November 1990.
- [32] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance - 1991*, pages 201–208. IEEE Computer Society Press, October 1991.
- [33] R. Lewis, D.W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *ESEC '89. 2nd European Software Engineering Conference Proceedings*, pages 487–496. Springer-Verlag, September 1989.
- [34] B.P. Lientz and E.B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Applications Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA, 1980.
- [35] B.P. Lientz, E.B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.
- [36] F. Martinig. Software testing: Poor consideration. *Testing Techniques Newsletter*, October 1996.
- [37] J. T. Nosek and P. Palvia. Software maintenance management: Changes in the last decade. *Journal of Software Maintenance Research and Practice*, 2:157–174, 1990.
- [38] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6), June 1988.
- [39] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247. Lawrence and Craig, September 1988.
- [40] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, NY, 1987.
- [41] D. S. Rosenblum and E. J. Weyuker. Predicting the cost-effectiveness of regression testing strategies. In *Proceedings of the ACM SIGSOFT '96 Fourth Symposium on the Foundations of Software Engineering*. ACM Press, October 1996.
- [42] G. Rothermel. Efficient, effective regression testing using safe test selection techniques. Technical Report 96-101, Clemson University, January 1996.
- [43] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 358–367. IEEE Computer Society Press, September 1993.
- [44] G. Rothermel and M.J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 14–25. IEEE Computer Society Press, September 1994.
- [45] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 94)*. ACM Press, August 1994.
- [46] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.
- [47] S. Schach. *Software Engineering*. Aksen Associates, Boston, MA, 1992.
- [48] D. Sharon. Meeting the challenge of software maintenance. *IEEE Software*, 13(1):122–125, January 1996.
- [49] B. Sherlund and B. Korel. Modification oriented software testing. In *Conference Proceedings: Quality Week 1991*, pages 1–17. Software Research, Inc., 1991.
- [50] B. Sherlund and B. Korel. Logical modification oriented software testing. In *Proceedings: Twelfth International Conference on Testing Computer Software*. Frontier Technologies, June 1995.

- [51] A.B. Taha, S.M. Thebaut, and S.S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 527–534. IEEE Computer Society Press, September 1989.
- [52] A. von Mayrhauser, R.T. Mraz, and J. Walls. Domain based regression testing. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 26–35. IEEE Computer Society Press, September 1994.
- [53] E.J. Weyuker. Empirical techniques for assessing testing strategies,. (Panel discussion at the International Symposium on Software Testing and Analysis), August 1994.
- [54] L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 262–270. IEEE Computer Society Press, November 1992.
- [55] L.J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 338–347. IEEE Computer Society Press, September 1993.
- [56] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference on Software Engineering*, pages 41–50. IEEE Computer Society Press, April 1995.
- [57] S.S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *COMP-SAC '87: The Eleventh Annual International Computer Software and Applications Conference*, pages 272–277. IEEE Computer Society Press, October 1987.
- [58] J. Ziegler, J.M. Grasso, and L.G. Burgermeister. An Ada based real-time closed-loop integration and regression test tool. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 81–90. IEEE Computer Society Press, October 1989.