A Safe, Efficient Regression Test Selection Technique1

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. Most regression test selection techniques depend on a particular test adequacy criterion or require prior knowledge of where code has been modified. We present a new technique for regression test selection that is neither adequacy-based, nor requires prior knowledge of modifications. Our algorithms construct control flow graphs for a procedure or program and its modified version, and use these graphs to select tests, from the original test set, that execute changed code. We prove that under certain conditions, the set of tests our algorithms select includes every test, from the original test suite, that can expose faults in the modified procedure or program. Thus, under these conditions, the 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.

1 Introduction

Software maintenance activities may account for as much as two-thirds of the overall cost of software production[38, 43]. 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[30].

An important difference between regression testing and development testing is that during regression testing, we may have an established suite of tests available for reuse. One regression testing strategy reruns all such tests, but this retest all approach may consume inordinate time and resources. 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.

Although many selective retest techniques have been proposed, most of these techniques emphasize the use of structural coverage criteria[4, 7, 9, 13, 14, 15, 18, 19, 20, 21, 23, 22, 29, 37, 44, 45, 54]. These

1A preliminary version of this work appeared in Reference [40]. This work was partially supported by a grant from Microsoft, Inc., and by NSF under Grant CCR-9357811 to Clemson University and Ohio State University.

techniques identify changed or affected program components, such as statements or paths, and attempt to select tests that exercise these components. Coverage criteria are useful because they offer a way to measure test adequacy. However, another goal of regression testing is to provide confidence that program functionality has not been adversely affected by modifications. In practice, most test suites contain tests designed to exercise functionality, rather than tests designed to cover components; this is particularly true for test suites designed for integration and system testing. Where selection of these tests from an existing test suite is concerned, we are often more concerned with finding tests that reveal faults in the modified program, than we are with finding tests that cover modified components. The coverage-based selective retest techniques cited above can omit such \fault-revealing" tests in significant situations[42].

Some selective retest techniques place less emphasis on coverage criteria[1, 11, 28, 32, 31, 48, 49]. One drawback of these techniques is that they require prior knowledge of where the modified program has been changed. This knowledge must be obtained either by a programming environment that tracks changes as they are made, or by algorithms[28, 52] that determine corresponding program components. The assumption of a programming environment that tracks changes may be unreasonable given current practice. The algorithms for determining corresponding components may be unnecessarily costly for this application.

Selective retest 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, in this paper, we address the first problem: the regression test selection problem. In this paper, we present a new technique for regression test selection that is neither coverage-criteria based nor requires complete information on

corresponding program components. Our algorithms construct control
flow graphs for a procedure or program and its modified version, and
use these graphs to select tests from the original test suite that
execute changed code.

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, our technique selects
every test, from the original test suite, that can expose faults in
the modified program. Thus, under those conditions, our algorithms are
safe. Moreover, although our algorithms may select some tests that
cannot reveal faults, they are more precise than other safe
algorithms, because they select fewer tests that cannot execute
changed code. Our algorithms are also more efficient than most
existing algorithms. The algorithms are easy to implement, and they
automate an important portion of the regression testing process.
Finally, our algorithms are more general than most previous
techniques. They handle both interprocedural and intraprocedural
regression test selection. Also, they 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. Our results suggest
that in practice, our algorithms can significantly reduce the cost of
regression testing a modified program.
In the next section, we provide background information. We present our
test selection algorithms in Section 3, and we analyze and evaluate
our algorithms in Section 4. Section 5 presents our conclusions.

2 Background

We adopt the following notation throughout the rest of this paper. Let
P be a procedure or program, let P be a modified version of P , and
let S and S0 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 S0(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, <identifier, input, output>,
in which identifier identifies the test, input is the input for that
execution of the program, and output is the specified output,

S(input), for this input. For simplicity, in the sequel we refer to a
test ht; i; S(i)i by its identifier t, and refer to the outputs P (i)
and S(i) of test t for input i as P (t) and S(t), respectively.
We assume that all statements in P are statically reachable. Because
it is possible to eliminate code that is not statically reachable from
a program, this is not an unreasonable restriction.

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 squares, stand
for conditional statements. Labeled edges leaving predicate nodes are
called branches { they 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. In the figure, for simplicity, we use statement
numbers as labels, but we could instead use the actual code of the
associated statements as labels. A unique entry node and a unique exit
node represent entry to and exit from P . The CFG for a procedure P
has size, and can be constructed in time, linear in the number of
simple and conditional statements in P [2].

To facilitate the presentation and discussion of our algorithms, we
think of unlabeled CFG edges as having the label \ffl"; we also assume
initially that case statements are rendered as nested if-else
statements. Given these assumptions, every CFG node has either one
out-edge labeled \ffl", or two out-edges, labeled \T" and \F",
respectively. Assume further that we associate all declaration and
nonexecutable initialization statements that pertain to a procedure
with a declaration node that we insert into the CFG as the immediate
successor of the entry node.2 We discuss other methods for handling
case statements, declarations, and other types of nonexecutable
initialization statements in Section 3.1.4.

Code Instrumentation

We can instrument a program P , such that when we execute the
instrumented version of P with test t, the program reports the
branches taken in the program for t. Using this branch trace
information, we can easily determine, for t, which edges in the CFG G
for P were traversed when t was executed.3 We call the 2For simplicity

we omit these nodes from our figures.
3We say that an edge (n1; n2) in G was traversed by test t if and only if, when t was run on P , the statements associated with n1 and n2 were executed sequentially at least once during the execution.

information thus gathered an edge trace for t on P . An edge trace for t on P has size linear in the number of edges in the CFG for P ; we can use a bit vector to represent such a trace.

Given program P and test suite T , we construct a test history for P with respect to T by gathering edge trace information for each test in T , and representing it such that for each edge (n1; n2) in the CFG for P, we know which tests traversed that edge. This representation requires O(ejT j) bits, where e is the number of edges in G, and jT j is the number of tests in T . For CFGs of the form we employ, 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, along with the corresponding test history, for program avg of Figure 1.

For convenience, in the remainder of this paper we assume the existence of a function, TestsOnEdge(n1; n2), that returns a bit vector v of size jT j bits, such that the ith bit in v is set if and only if test i in T traversed edge (n1; n2) in G.

2.2 Regression Testing

Research on regression testing spans a wide variety of topics, including test environments and automation[10, 12, 25, 24, 55], capture-playback mechanisms[34], test suite management[16, 23, 34, 45, 51], program size reduction[8], and regression testability[30]. Most recent research on regression testing, however, concerns selective retest techniques.

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[33] show that a selective retest technique is more economical than the retest-all technique 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 lets us omit. A typical selective

retest technique proceeds as follows:
1. Select T ? 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 00, a set of new functional or structural
tests for P .
4. Test P with T 00, establishing P 's correctness with respect to T
00.
5. Create T 000, a new test suite and test history for P , from T , T
, and T 00.

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
S0, 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 S0 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.4

We distinguish two 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, 4For examples of specification-based techniques, see References
[32] and [46].

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.5

3 Regression test selection algorithms

For reasons that shall become clear, our goal is to identify
nonobsolete tests in T that execute changed code with respect to P and
P . In other words, we wish to identify tests in T that (1) execute
code that is new
or modified for P , or (2) used to execute code in P that is no longer
present in P . We call such tests
\modification-traversing for P and 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 tested with t. We say that
two execution traces ET (P (t)) and ET (P (t)) are equivalent if they
have
different lengths, or if, when we compare their elements from first to
last, we find some pair of elements that are lexically nonidentical.
We say that t is modification-traversing for P and P (or for
simplicity, that t is
modification-traversing) if and only if ET (P (i)) is not equivalent
to ET (P (i)).
To identify the modification-traversing tests in T we must identify
nonobsolete tests in T that have different 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.6 In
addition, assume that for each test t 2 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.
We next present an algorithm that identifies modification-traversing
tests for intraprocedural test selection. Section 3.2 presents our
interprocedural test selection algorithm.

3.1 Intraprocedural test selection

We begin the presentation of our intraprocedural algorithm by
informally motivating our approach.

Given an execution trace, there is a unique mapping between that trace
and the nodes of a CFG. We obtain this mapping by replacing each
statement in the execution trace by its corresponding CFG node, or
equivalently, by the label of that node. We call such a mapping a

traversal trace and say that for 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)), we define the traversal trace prefix for
TR(P (t)) with respect to N to be the portion of TR(P (t)) beginning
with the first node in the trace and ending at N .
5There 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, turns to 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 with the big bang model; however, for both models, both phases
exist, and can be exploited.
6If we cannot effectively determine test obsolesence, we cannot
effectively judge test correctness. Thus, this assumption is necessary
if we intend to reuse tests at all, whether selectively or not.

Given two execution traces, we can perform a pairwise comparison of
the traces by comparing the first statements in each trace to each
other, then comparing the second statements 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
execution traces for t in P and P reaches a first pair of statements
that are lexically different. Assume
that CFG nodes contain, as labels, the text of the statements to which
they correspond. In this case, we can perform a pairwise comparison of
the nodes in two traversal traces by comparing node labels. If t has
different traversal traces in P and P we shall reach a first pair of
nodes that have lexically different labels.

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
differ 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 G0, such
that N and N differ, and N and N are endpoints of traversal trace
prefixes that are identical 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 G0, looking for pairs of nodes N and N that
differ. When we find such a pair, we
use TestsOnEdge to select all tests known to have reached N .

On first view, it may appear expensive to compare traversal traces,
because such traces can be lengthy { particularly when procedures
contain loops. However, three facts combine to render the approach
efficient. First, when we walk a pair of traversal trace prefixes p
and p0 in G and G0, respectively, reach a pair of nodes N and N that
differ, and select the tests that reach N , we need not walk beyond N
and N : we
have selected all tests that have traversal trace prefixes p and p0
and reach additional modifications beyond N and N . Tests that reach
modifications accessible along other traversal trace prefixes are
selected when we traverse those other prefixes. Second, when we reach,
in our walk, a pair of nodes N and N that have
already been pairwise compared, we do not need to compare N and N
again. Third, when we attach test
information to edges in G, a single walk of G simultaneously accounts
for all tests attached to each edge as it crosses the edge; we do not
require a separate graph walk for each test in T . For these reasons
the graph walks required to select modification-traversing tests on
CFGs require worst-case time quadratic in the number of nodes in the
graphs.

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 OE, and then constructs
CFGs G (with entry node E) and G0 (with entry node E0) for P and P ,
respectively. Next, the algorithm

calls Compare with E and E0. Compare ultimately places tests that are
modification-traversing for P and P into T . SelectTests returns these
tests.7

7An earlier version of this algorithm[40] 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). Both approaches select safe test sets, but the
CFG-based approach is more efficient, and easier to implement, than
the earlier approach.

Compare is a recursive procedure. Compare is called with pairs of
nodes N and N , from G and G0,
respectively, that are reached simultaneously during our algorithm's
synchronous walks of traversal trace prefixes. Given two such nodes N
and N , Compare determines whether N and N have successors that
are syntactically nonequivalent along pairs of identically labeled
edges. If N and N have syntactically
nonequivalent successors 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 syntactically equivalent
successors along a pair of identically labeled
edges, Compare walks forward along the edges in G and G0, 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 a pair 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 C such
that (N ,C) has label L. (Such a C necessarily exists, because
Compare is called only with pairs of nodes
N and N that are syntactically equivalent; thus N and N are either
both predicate nodes with out-edges
labeled \T" and \F", or N and N are both statement nodes with
out-edges labeled \ffl".) Next, Compare
considers C and 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 Equivalent with C

and C . If Equivalent(C,C) is false, then tests that traverse edge
(N; C) are modification-traversing for P
and P , so Compare uses TestsOnEdge to identify these tests, and adds
them to T . If Equivalent(C,C) is

true, Compare invokes itself on C and C , to continue the graph traversals beyond these nodes.
The Equivalent function takes a pair of nodes N and N , and determines whether the statements S
and S0 associated with N and N may be semantically nonequivalent.8 A simple version of Equivalent
answers conservatively the question of whether S and S0 are semantically nonequivalent, by comparing the two statements lexicographically, ignoring extra white-space characters when not included in character constants. We discuss more precise, albeit more complex, versions of Equivalent in Section 3.1.4.

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 OE, constructs the CFGs for the two procedures, and calls Compare with entry and entry0. Compare marks entry \entry0-visited", and then considers the successor of entry, S1. Compare finds that S10 is the corresponding successor of entry0, and because S1 is not marked \S10-visited", calls Equivalent with S1 and S10. Because S1 and S10 are syntactically equivalent, Equivalent returns true, and Compare invokes itself on S1 and S10 (invocation 2). Recursive calls continue in this manner on nodes S2 and S20 (invocation 3), and P3 and P30 (invocation 4); in each case the successors of the nodes are equivalent. On invocation 4, Compare must consider two successors of P 3: P4 and S9. When Compare considers S9, it calls itself with S10 and S100, and subsequently with exit and exit0, and selects no tests. When Compare considers P4 and P40, it first seeks a true child 8The phrase \may be" is important here, because the problem of determining semantic equivalence of statements is, in general, undecidable[26].

of P40 to compare with S5; it finds S5a and calls Equivalent with S5 and S5a. The statement associated with S5 and the statement associated with S5a are not syntactically equivalent, so Equivalent returns false; Compare uses TestsOnEdge(P4,S5) to locate the set of tests (ft2g) that reach S5 in avg and adds these tests to T . When Compare seeks a false successor of P40, it finds S60, and calls Equivalent with S6 and S60; Equivalent returns true for these nodes, so Compare invokes itself on S6 and S60. Compare finds the successors of these nodes, S7 and S80, nonequivalent, and adds to T the set of tests that traverse edge

(S7,S8); T is now ft2,t3g. At this point the algorithm has walked 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 = ft2,t3g,
in which both tests are modification-traversing.
Many other regression test selection techniques[4, 7, 14, 18, 20, 23,
37, 45] omit t2 or t3.

If, for avg and avg2, the deletion of S7 had been the only change,
SelectTests would have returned only ft3g. If the addition of S5a had
been the only change, SelectTests would have returned only ft2g. In
this latter case, observe that Compare would eventually invoke itself
with S8 and S80, but find the successor of S8, P 3, already marked
\P30-visited"; thus, Compare would not reinvoke itself with P3 and
P30.

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 P40 in avg2's CFG. In this case, called with the CFGs for
avg and avg2, SelectTests proceeds as in the previous example until it
reaches P3 and P30. Here it finds nonequivalent successors P4 and P40,
and selects ft2,t3g. The procedure does not need to analyze successors
of P4 and P40.

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 initial invocation of Compare,
when Compare detects the differences between successors of entry and
entry0. The procedure does not need to analyze successors of entry and
entry0: 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, twovisits0 (far right), and the
CFGs for the two procedures (next to their respective procedures). The
two versions produce identical output for all values of x other than
\0". When x = 0, the execution trace for twovisits is h entry, P 1,
S2, S3, S4, exit i, and the procedure prints \1". For the same input,
the execution trace for twovisits0 is h entry0, P10, S20, S50, exit0
i, and the procedure prints \2".

When SelectTests runs on the CFGs for twovisits and twovisits0, it
considers entry and entry0

first, and then invokes itself on P1 and P10. Suppose SelectTests next
invokes itself on S3 and S30 (we have not required the algorithm to
visit the successors of a pair of nodes in any particular order; if we
had, 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 \S30-visited", then continues with invocations on
S4 and S40, and exit and exit0, selecting no tests, because tests that
take these paths in the two versions are non-modification-traversing.
Now, SelectTests resumes its consideration of successors of P1 and P10
and

invokes itself on S2 and S20. SelectTests marks S2 \S20-visited" and
considers the successors of S2. S2 has only one successor, S3; the
corresponding successor of S20 is S50. 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 S50 and would not detect the need
to select tests through S3. However, as the algorithm is written,
SelectTests sees that S3 is only marked S30-visited, not S50-visited,
and thus, proceeds to check for equivalence of S3 and S50, 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 .9
Figure 6, which illustrates this possibility, depicts a C function,
pathological, and a modified version of that function, pathological0,
with the CFGs for the versions. 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 we run
pathological and pathological0 on these tests. When we run
pathological on tests t1 and t2, it outputs \1" for both. When we run
pathological0 on test t1, it also outputs \1"; however, when we run
pathological0 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
pathological0. Called with entry and entry0, Compare invokes itself
with P1 and P10, then with P2 and P20. When invoked with P2 and P20
Compare considers their successors, P1 and P30, respectively, finds

them lexically identical, and invokes itself with them. Compare finds their successors, S4 and P40, nonequivalent, and selects the tests on edge (P1; S4), that is, tests t1 and t2.

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, 9This 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 [39].

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 terms. Thus, t1 is not modification-traversing for the two versions, and SelectTests chooses it unnecessarily.

3.1.2 Efficiency, effectiveness, and correctness of the algorithm

In this section, we summarize results that are presented in detail in Reference [39].

We define controlled regression testing 10 as the practice of testing P under conditions equivalent to those
that were used to test P . As such, controlled regression testing applies scientific method to regression testing: to determine whether code modifications cause errors, we test the new code, holding all other factors that might affect program behavior constant.11

For the purpose of regression test selection, we would like to identify all tests t 2 T that cause P to
produce output that differs from its specified output and thus, reveal faults in P . We call such tests faultrevealing,
and we call an algorithm that selects every fault-revealing test in T safe. There is no effective procedure that, in general, precisely identifies the fault-revealing tests in T . However, under controlled regression testing, the modification-traversing tests are a superset of the fault-revealing tests. Thus, for cases where controlled regression testing is possible, a regression test selection algorithm that selects all modification-traversing tests necessarily selects all fault-revealing tests, and is safe. This result is significant,

because we can show that for cases where controlled regression testing
is practiced, if t 2 T is modificationtraversing, then SelectTests
selects t. Thus, the following theorem holds:

Theorem 1: SelectTests is safe for controlled regression testing.
Proof: See Reference [39], pages 77-81.
Theorem 1 is important, because although we cannot find an algorithm
that is safe in general, we know from the theorem that SelectTests is
safe in situations in which controlled regression testing is possible.

Theorem 1 lets us draw conclusions about the ability of our technique
to select tests that may reveal faults. However, we are also
interested in how well our technique does at omitting tests that
cannot reveal faults. For controlled regression testing, tests that
are non-modification-traversing cannot be fault-revealing; ideally, we
would like to omit all such . In the previous section, however, we
showed that there are cases in which SelectTests selects tests that
are non-modification-traversing. The theorem identifies a necessary
condition for SelectTests to select such tests: the
multiply-visited-node condition. The multiply-visitednode 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. Our next
theorem is as follows: Theorem 2: Given procedure P , modified version
P and test suite T for P , if the multiplyvisited-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: see Reference [39], pages 82-85.
Theorem 2 gives us a way to characterize the class of programs and
modified versions for which SelectTests selects
non-modification-traversing tests. The theorem is significant for two
reasons. First, procedures like pathological and pathological0, which
cause the multiply-visited-node condition to hold, are atypical. With
some work, we can construct other examples that illustrate similar
effects; however, all examples located to date have been contrived.
Second, in our empirical studies using \real" programs, we have never
found a case in which the multiply-visited-node condition holds. Thus,
we conjecture that in practice, given P , P , and T , SelectTests
selects exactly the tests in T that are modification-traversing for P
and P .
10In earlier work[39, 42] we used the phrase \proper regression
testing" in this context; however, the term \proper" has a pejorative
connotation that we do not intend to imply. Hence, in this work, we

substitute the more neutral term \controlled". 11The 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."

Our final theorem is as follows:

Theorem 3: SelectTests terminates.
Proof: 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. See Reference [39], page 81, for details.

3.1.3 Complexity of the algorithm

The running time of SelectTests is bounded by the time required to
construct CFGs for P and P , plus
the number and cost of calls to Compare. Let n be the number of
statements in P , and n0 the number of statements in P . CFG
construction is an O(n) operation[2]. We obtain an upper bound on the
number of calls to Compare by assuming that Compare can be called with
each pair of nodes N and N in G and G0,
respectively. Under this assumption, the overall cost of SelectTests
is O(n + n0 +m(nn0)), 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 Equivalent (line 13).
Depending on the results of the Equivalent 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 Equivalent 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 kjT j for some constant k. It
follows that given a pair of procedures for which CFGs G and G0
contain n and n0 nodes, respectively, and given a test suite of t
tests, if Compare is called for each pair of nodes (N ,N) (N 2 G and
N 2 G0),
the running time of SelectTests is O(tnn0).
The assumption that Compare may be called for each pair of nodes N and
N from G and G0 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 minfn; n0g times.
Thus, if our conjecture that the multiply-visitednode condition does
not hold in practice is correct, then for practical purposes,
SelectTests runs in time O(t(minfn; n0g)).

3.1.4 Improvements to the basic algorithm

In the preceding sections, to facilitate the presentation, we
presented a deliberately simplified version of SelectTests. There are
several ways in which to increase the efficiency or precision of the
basic algorithm. We discuss three improvements here; Reference [39]
discusses additional improvements.

Determining statement nonequivalence more precisely

The Compare procedure that we presented in Figure 3 relies on a
function, Equivalent, that takes a pair of nodes N and N , and
determines whether the statements S and S0 associated with N and N may
be

semantically nonequivalent. To retain safety, Equivalent must
conservatively estimate whether pairs of statements are semantically
nonequivalent, erring only on the side of claiming nonequivalence when
the statements are in fact equivalent.

A simple version of Equivalent performs a lexicographic comparison of
S and S0, ignoring white-space characters except when they are part of
character constants. This approach is safe, and easy to implement, but
also imprecise for several obvious cases. Consider, for example, the
statements \if (x<0)" and \if (0>x)". Clearly, the two statements are
lexicographically nonidentical but semantically equivalent. The same
applies to the statements \x=(n*n)-(2*n)+1" and \x=(n-1)*(n-1)", and
to the statements \x=1; /* comment */" and \x=1; /* different comment
*/".

A more sophisticated implementation of Equivalent can detect, for
cases like these, whether statements are semantically nonequivalent.
For example, Equivalent can easily ignore comments when it performs
lexicographic comparisons of statements. Additionally, for certain
classes of expressions such as assignment statements, Equivalent can

determine equivalence using techniques such as polynomial interpolation or symbolic computation[50]. At further expense, Equivalent can use algorithms, such as those described in References [4, 8, 26, 53], that conservatively determine statement equivalence.

Improving Equivalent increases the precision of our test selection algorithms, but also increases the cost of the algorithms. In practice, we expect that some improvements to Equivalent will produce precision improvements sufficient to justify this added cost, while others will not.

Handling variable and type declarations

A second test selection issue involves 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. A simple way to handle variable and type declarations in the context of our intraprocedural test selection algorithm is to associate each declaration with a CFG node, and to attach every test that enters the procedure to the edge that enters that node. An analogous technique handles the interprocedural case. In this case, our algorithms detect syntactic differences between variable and type declarations, and flag tests that encounter these differences as modification-traversing.

This approach has drawbacks. Just as a modified variable initialization statement may unnecessarily force selection of all tests through a procedure, so 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 t 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 t. Our test selection algorithms can be modified to reduce imprecise test selection at declaration changes.

One approach requires the algorithm to determine affected variables, which are variables whose declarations have changed or whose declarations are dependent on changed type definitions, and keep a list of these variables. The Equivalent 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 separate case. This representation can yield more precise test selection than the nested if-else representation. 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 when 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 G0, and selects all tests that took the edge in G. With minor modifications, our Compare routine copes with these differences, and handles the alternative representation of switches.

3.2 Interprocedural test selection

Thus far we have considered our technique in an intraprocedural context. Our examples demonstrate that the technique reduces the number of tests that must be rerun on a single procedure. However, the examples also suggest that when procedures are small and uncomplicated, and their test sets are small, it may be more cost effective to rerun all tests.

This objection is mitigated when we consider 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. Moreover, in the interprocedural context, the problem of regression test selection is even more differentiable from the problem of coverage

identification than in the intraprocedural context. Structural
coverage criteria, if employed at all, are typically applied only
during intraprocedural testing. Although integration and system-level
coverage criteria have been proposed[35], in practice, most system and
integration testing is functional or specification-based. It is
precisely with functional and specification-based tests that we are
most interested in running all tests that may reveal faults, without
regard to coverage considerations. It is precisely with functional and
specification-based tests that we can benefit most from an automatable
regression test selection technique that selects tests based on
program structure, because we cannot in general determine tests that
are not fault-revealing simply by referring to program specifications.
Assume that we can obtain a mapping of procedure names in P to
procedure names in P , that tells us, for
each procedure P 2 P , the name P of its counterpart in P . A simple
but naive approach to interprocedural regression test selection runs the intraprocedural
SelectTests
algorithm on every pair of procedures (P ; P)
such that P is the counterpart of P . With this simple approach, if a
procedure P 2 P is no longer present in P , tests that used to enter P
are selected at former call sites to P . Similarly, if a procedure P
is inserted
into P , tests that enter P are selected at the call sites to P .
Notice that the number of times P is called
from within P is immaterial: trace information tells us precisely
which tests reach which edges in P , and a single traversal of P and P
suffices to find tests that are modification-traversing for P and P .
If we run SelectTests on all pairs of corresponding procedures in P
and P , we obtain a safe test set
T , but we 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, sys0. In this case, all tests in T are modification-traversing
for sys and sys0, and are selected when SelectTests, called with main
and main0, reaches S1 in main. There is no need in this case to
compare procedures A, B, and C to their counterparts in sys0: we have
already selected all tests that could reach those procedures and
become modification-traversing within them. The naive algorithm does
unnecessary work for this example.

Alternatively, suppose statement S6 (and no other statement) in sys is modified. In this case, because every test of sys that enters main necessarily 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 C0. There is no need in this case to compare B with its counterpart in sys0, 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. If the algorithm has previously completed its walk of a procedure, it uses the knowledge, gained during that walk, of whether tests that enter the procedure can exit it without becoming modification-traversing: the algorithm continues its graph traversal beyond call nodes if and only if tests can pass through called procedures without becoming modification-traversing.

3.2.1 The basic interprocedural test selection algorithm

Figure 9 gives our algorithm, SelectInterTests, for selecting 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". SelectTests first initializes T and proctable to OE, and invokes SelectTests2 on the entry procedures,
PE 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 begins by inserting P into proctable, and setting the status flag for P to \visited", to indicate that a traversal of P has begun. The procedure then creates CFGs for P and P , and calls Compare2 with the entry nodes of those
CFGs. When control returns from Compare2 to SelectTests2, SelectTests2 checks to see whether the exit node of P was reached during the traversal of Compare2. If not, tests that enter P become

modificationtraversing 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 two
nodes C and C equivalent, before it
invokes itself on those nodes it checks to see if 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
lexically identical; 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 O0 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 the 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

algorithm SelectInterTests(P, P , PE, P E , T) : T input P,P : base
and modified versions of a program or subsystem 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 our basic intraprocedural algorithm, discussed in Section 3.1.4, also apply to our interprocedural algorithm.

To illustrate the use of SelectInterTests, we offer the following example. Suppose we change program sys of Figure 8 to sys0, by modifying the code associated with node S5 in procedure B. (We do not show the modified graphs; to discuss the example we distinguish nodes in the CFG for sys from nodes in the CFG for sys0 by adding primes to them). Initially, SelectInterTests calls SelectTests2 with main and main0. 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 A0, because A is not listed in proctable, invokes SelectTests2 on A and A0. SelectTests2 adds A to proctable with status \visited", then builds the CFGs for the two procedures, and begins traversing them. On reaching calls to C, the algorithm adds C to to proctable with status \visited", makes CFGs for C and C0, and begins traversing them. The algorithm finds no calls or differences in C and C0 and thus, when the call to Compare with the entry nodes of C and C0 terminates, the algorithm marks the exit node of C \exit0-visited". This marking means that tests entering C and C0 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 C0, Compare resumes at line 29, finds that the status flag is not set, and continues traversing A and A0 by invoking itself with the call C node and its counterpart in the graph for sys0. 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 B0. This invocation ultimately identifies the differences in B and B0, 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 B0 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 A0, Compare2 resumes, at line 29, with the call A node in main and its corresponding node in main0. The algorithm traverses the graphs through node S3 and its counterpart in main0. 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 B0. Furthermore, Compare2 sees that all procedures called in the call B node have status \selectsall", and thus does not traverse the graph for main any further.

3.2.2 Complexity of the algorithm

SelectTests and SelectInterTests are of comparable complexity. To see this, suppose P contains p procedures, and n statements of which c contain procedure calls, and suppose P contains n0 statements. Assume that the number of procedure calls in a single statement, and the length of a statement, are bounded by constants k1 and k2, respectively. We obtain an upper bound on the running time of SelectInterTests 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(n0). We obtain an upper bound
on the number of calls to Compare2 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(nn0). Excluding, for the moment, the cost of the P rocT able lookups in lines 24-31, a call to Compare2 requires the same amount of work as a call to Compare, namely, k2jT j. Regardless of the number of calls to Compare, lines 24-31, over the course of a complete execution of SelectInterTests, require at most k1c 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 k2. Thus, the time required by SelectInterTests, in the worst case, is O(n +n0 + jT jnn0 + cp), where cp is bounded above by k1n2. In practice, however, we expect a lower bound on execution time. When the multiply-visited-node condition does not hold, the term jT jnn0 becomes jT j(minfn; n0g). Furthermore, if we use an efficient hashing scheme to implement P rocT able, we reduce the O(cp) table lookup time to (expected time) O(c), which is O(n).

4 Evaluations of the Algorithms

In this section, we evaluate our algorithms, and compare them to existing test selection techniques. Section 4.1 presents an analytical evaluation and comparison; Section 4.2 presents empirical results.

4.1 Analytical Evaluation

In Reference [42], we present a framework for analyzing regression test selection techniques, that consists of four categories: inclusiveness, precision, efficiency, and generality. Inclusiveness measures the extent to which a technique selects tests that reveal faults in a modified program; a 100% inclusive technique is safe. Precision measures the extend to which a technique omits tests that cannot reveal faults in a modified program. Efficiency measures the space and time requirements of a technique, focusing in particular on critical phase costs. Generality measures the ability of a technique to function in some practical, and sufficiently wide, range of situations. We use our framework to compare and evaluate all code-based regression test selection techniques that we have found in the literature. We also use our framework to evaluate our technique and compare it to existing techniques. Here, we summarize the results reported in that work.

Inclusiveness.

Our test selection algorithms are safe for controlled regression testing. Only three other techniques[11, 23, 28] can also make this claim. Moreover, all of these safe techniques depend, for their safety, upon the same assumptions on which our algorithms depend. Thus, at present, 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 efficient algorithm that is both safe and 100% precise. Thus, because our goal is safety, some imprecision is unavoidable.

Nevertheless, our algorithms are the most precise safe algorithms currently available. For cases where the multiply-visited-node condition does not hold (which we expect 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 test
selection techniques, and we have strong evidence to suggest that our
algorithms are more precise than the third technique[39].

Efficiency.

As we discussed previously, our algorithms run in time O(jT jn2) for
procedures or programs of n statements, and test set size jT j. This
is an improvement over the efficiency of two of the other safe
techniques. Moreover, in practice, we expect the basic algorithms
(without added precision) to run in time O(jT jn); a bound comparable
to the worst-case run time of the third safe technique. 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 other 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. Our technique supports both intraprocedural and
interprocedural test selection. 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, we also need multiple versions of the
software. 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 our subject population is the universe of software systems, we do not know what it means to select a fair cross-section of that population[47]. Nor do we know what it means to select a fair cross-section of the universe of modified versions or test suites for software. 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.12 We implemented our tools and conducted our empirical studies on a Sun Microsystems SPARCstation 10 with 128MB of virtual memory.13

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.

Subjects

Hutchins, Foster, Goradia, and Ostrand[27] 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 8 to 21 procedures. They manufactured 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.14

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 capability, the
study employed faulty versions of base programs. For our purposes, we
think of these faulty versions as ill-fated attempts to create
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.[27] describe the process used by the Siemens
researchers to manufacture test suites and 12We describe the
requirements for our tools, and their design and implementation, in
Reference [39]. 13SPARCstation is a trademark of Sun Microsystems,
Inc.
14There are a few differences between the numbers reported in Table 1
and the numbers reported in Reference [27]. 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). In some cases, we can attribute the difference
in numbers of tests to repetitions of equivalent tests in their test
plan. In any event, the difference amounts to less than 1% of the
total test pool size and does not affect the results of this study.

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, 36]; 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 manufactured 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 Siemens test suites are enormous, and the seeded faults 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 served previously 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 of the 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. We obtained execution timings 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 that we report for this study list these average results. In our experimentation, we practiced controlled regression testing.

Results

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

Figure 11 depicts the test selection results for our interprocedural test selection tool, DejaVu2, in Study 1. The graph depicts, for each of the 7 base versions of Siemens programs, the percentage of tests selected by DejaVu2, on average, over the set of modified versions of that base program. As the graph shows, in Study 1, 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 43.29% (on replace) to 93.58% (on schedule2) of the size of the total test sets for those programs.

The fact that our test selection algorithms reduce the number of tests required to retest modified programs does not by itself indicate the possible worth of our algorithms. If we required ten hours of analysis to determine that we could save one hour of testing, this might not be of benefit { unless the ten hours were fully automated, the hour saved was an hour of human time, and we could spare the ten hours. We would like to show that the time we save in not having to rerun tests exceeds the time we spend analyzing programs and selecting reduced test suites. Toward this end, Figure 12 depicts some timings. For each of the seven base programs, the figure shows the average time required to run all of the tests on the modified version of the program (darkest column), the time required to perform analysis, on average, of the base and modified versions of the program (lightest column), and the time required, on average, to run the selected tests on the modified version of the program. Times are shown in minutes and seconds. Note that in this study, we were able to fully automate both the execution of tests, and the validation of test results.

As the figure indicates, the cost of our safe test selection algorithms, in terms of time, is negligible: never exceeding 21.8 seconds. Our 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, 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, using DejaVu2 saved 9 minutes and 17 seconds, or 53% of total effort.

By absolute measures, savings of a few minutes and seconds, such as this study illustrates, may be relatively unimportant. However, in practice, test suites may take hours, days, or even weeks to run, and much of this effort may be human-intensive. Moreover, lengthy testing times may occur despite efforts to

automate test execution and results validation. If results such as those depicted by this study scale up to such larger-scale cases, a savings of even 10% may matter, and a savings of 50% may offer a big win. In fact, we conjecture that the savings obtainable from DejaVu2 increase, on average, as larger programs are used as subjects. Our second study investigates this conjecture.

4.2.2 Study 2: Interprocedural Test Selection on a Larger Scale

Our second study investigated the efficacy of our interprocedural test selection algorithm on a larger subject.

Subjects

For our second study, we obtained a program, player, that is one of the executable components of an internet-based game called Empire. The player executable is essentially a transaction manager; its main routine contains initialization code, followed by a five-statement event loop that waits for receipt of a user command, and on 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 as an internet-based game, Empire has been rewritten many times; several 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 specifica-

tions; 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, for which test suites often contain only functional tests.

The player program is a reasonable 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 a variety of user commands.15 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 reasonably large.

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, but we were able to obtain close estimates of those times.16

Results

Figure 13 depicts the test selection results for our interprocedural test selection algorithm for the modified versions of player. The graph depicts, 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 4.83% of the tests in the existing test suite. In other words, on average, the algorithm reduces the number of tests that must be rerun by over 95%.

Figure 14 depicts timings for this study. The graph depicts, for each of the five modified versions, the time required to run all of the tests on the modified version of the program (darkest column), the estimated time required to perform analysis of the base and modified versions of the program (lightest column), and the time required to run the selected tests on the modified version of the program. Times are shown in hours, minutes, and seconds. 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).

15Other 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.

16We determined the time required to build CFGs and run DejaVu1 on 85% of the code. We report that time scaled up by 1.15. Work is ongoing to improve the robustness of our analysis tools.

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, we could build the CFGs for procedures in the base version during the preliminary phase, and build CFGs for procedures in modified versions on demand, lowering the analysis cost. Furthermore,

our 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, then, the time required to run tests would be much
larger than the times shown, and the resulting savings would increase.

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. The results support our
conjecture that in practice, our algorithms will not select
non-modification-traversing tests.

Second, although we have reported our results as averages over sets of
modified programs, it is interesting to examine the behavior of our
algorithms for individual cases.17 Consider, for example, the results
for program replace. The test pool for replace contains 5542 tests, of
which DejaVu2 selects, on average, 2399 (43.29%). 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 1611.48. In
contrast, for schedule2, DejaVu2 selects, on average, 2508 (93.58%) of
the program's pool of 2680 tests, with a standard deviation of 253.49.
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 be able to determine
the factors that influence the success of test selection. This could
help us determine when test selection is likely to be successful, and
also help us draw 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 the modifications in P , and the
extent
and type of code coverage achieved by tests in T . Although these
factors can interact, they may also operate independently.

Finally, as an interesting side-effect of this study, our results
support some hypotheses about the faultrevealing capabilities of
regression test selection techniques. Study 1 involved faulty modified
program versions; for these versions, a very small percentage of tests

are fault-revealing. For example, consider program replace, with a test suite of 5542 tests. For version 26 of replace, 302 of the 5542 tests 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 17Reference [39] lists results for all programs and modified versions individually.

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 .00064% chance of selecting a test that exposes the fault. In either case, DejaVu2 guarantees that the fault is exposed. Every case considered in Study 1 supports a comparable conclusion.

It would not be fair, on the basis of Study 1 alone, to draw general conclusions about the relative effectiveness of minimization and safe test selection techniques, because the Siemens study deliberately restricted program 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, and such as those found in the Siemens study in general, arise in practice. Our results thus give us good reason to question the efficacy of minimization test selection techniques where fault detection is concerned.

4.2.4 Summary of empirical results, and limitations of the studies

We summarize the major conclusions derived from our empirical studies about the use of our algorithms in practice as follows:

ffl 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.
ffl We expect interprocedural test selection to offer greater savings than intraprocedural test selection. ffl Regression test selection algorithms may yield greater savings when applied to large, complex programs than when applied to small, simple programs.
ffl There exist programs, modified versions, and test suites for which test selection offers little in the way of savings.

ffl 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.
ffl Our conjecture that in practice, programs contain no multiply-visited-nodes, remains plausible. ffl Our results support a conjecture that minimization techniques for test selection may be ineffective at revealing faults.

For the sake of fairness, we mention the following limitations on our studies:

ffl We have studied only a small sample of the universe of possible programs, modified programs, and test suites. We have no hard data on which to base a 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.
ffl Both of our studies required some manufactured artifacts: study 1 used manufactured modified versions and tests, and study two used manufactured tests. In both cases, however, efforts were made to ensure that manufactured artifacts were representative of real counterparts.
ffl Due to limitations in our program analysis tools, our second study required an estimation of analysis times. However, we were careful to estimate fairly and conservatively. We believe that our estimates understate the analysis time required by our technique.

5 Conclusions and Future Work

We have presented a technique for selecting tests from an existing test suite for use in regression testing a modified procedure or program. We have shown that for controlled regression testing, our algorithms are safe: they select every test in the existing test suite that may reveal faults in the modified program. Our algorithms are also the most precise safe test selection algorithms currently available. The algorithms are efficient, and handle both intraprocedural and interprocedural regression test selection. Our empirical studies suggest that in practice, our technique can reduce the cost of regression testing a modified program. This work is important for two reasons. According to Pressman[38], the cost of software maintenance dominates the overall cost of software. Moreover, the cost of maintenance, measured in terms of the percentage of software budget spent on maintenance, is increasing. In the 1970s, typical industry expenditures on software maintenance comprised 35 to

40 percent of total software budget. In the 1980s, expenditures rose to between 40 and 60 percent of total budget, and in the 1990s the figure exceeds 70 percent. Beizer presents similar statistics[6]. Because regression testing constitutes a significant percentage of maintenance costs, there are good reasons to seek improvements in regression testing processes.

Both development testing and regression testing are expensive. However, the cost of development testing is incurred once, when a software product is created; this cost is then amortized over the lifetime of the product. In contrast, a software product is regression tested every time a new version of that product is released. The cost of regression testing is thus compounded over the lifetime of the product. Thus, improvements in regression testing may have greater potential to reduce the overall cost of software than improvements in development testing.

The motivation for finding better regression testing techniques is not just economic. Both developers and users of software want to believe that their software works correctly. Both developers and users of software want the probability that their software will fail to be reduced. Testing is the method of choice both for building confidence in software, and for increasing its reliability. Regression testing plays an important role in both of these tasks. At present, however, according to Beizer, regression testing is often overlooked or performed inadequately: either the testing of new features, or the revalidation of old features, or both, are sacrificed. As a result, software reliability decreases over the software's lifetime[6]. Practical, effective selective retest techniques promote software quality.

We have discovered several promising directions for future work in this area. First, while the empirical results reported in this paper are encouraging, they are also preliminary. Additional implementations of the algorithms and enhancements described in Section 3, and further empirical studies with those implementations, would be useful. Empirical work to examine the cost and the fault-revealing capabilities of these and other regression test selection techniques would be particularly valuable.

Second, our work has focused on the problem of selecting tests from an existing test suite. An equally important problem, however, is that of ensuring that code modified or affected by modifications is adequately tested. The tests that we select from an existing test suite can help establish adequacy, but new tests may also be required. Future work should consider the extension of this technique to help identify the need for new tests.

Third, our work considers the problem of code-based test selection; future work should consider the application of the technique to the problem of specification-based selection.

Fourth, our research has revealed that the size of the test sets our algorithms select may vary significantly. This variance is a function of program structure, 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.

Fifth, our technique is safe only for controlled regression testing. In this respect, all existing safe regression test selection techniques are alike. In the absence of controlled regression testing, techniques that are otherwise safe may omit fault-revealing tests. This raises the question: can we find ways to increase the conditions under which safe test selection is possible? There are several ways in which to address this question. We might try to find ways to make controlled regression testing possible in situations wherein it is currently difficult to attain. Or, we might try to find ways to identify the types of fault-revealing tests that techniques like ours might omit in the absence of controlled regression testing, and extend the techniques to select these tests. Or finally, we might find ways to otherwise identify the errors that could be exposed by tests omitted by these techniques.

Finally, even when we cannot employ controlled regression testing, and thus cannot guarantee safety, our algorithms may still select test suites that are useful. It is not the case that our algorithms function only in the presence of controlled regression testing: it is simply that they, like all other regression test selection algorithms, are not safe in the absence of controlled regression testing. Viewed differently, our algorithms select all tests in T that, if we had practiced controlled regression testing, could have exposed faults in P .
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. Thus, even in the absence of controlled regression testing, we may find our algorithm useful.

Empirical studies could investigate this conjecture further.

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, 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, 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, January 1993.

[5] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, New York, NY, 1990.

[6] B. Beizer. Black-Box Testing. John Wiley and Sons, New York, NY, 1995.

[7] 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, October 1988.

[8] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In Proceedings of the Conference on Software Maintenance - 1992, pages 41{50, November 1992.

[9] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In Proceedings of the Conference on Software Maintenance - 1995, October 1995.

[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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, December 1989.

[26] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In Proceedings of the 14th International Conference on Software Enginnering, pages 392{411, May 1992.

[27] 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, May 1994.

[28] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintentance. In Proceedings of the Conference on Software Maintenance - 1992, pages 282{290, November 1992.

[29] J.A.N. Lee and X. He. A Methodology for Test Selection. The Journal of Systems and Software, 13(1):177{185, September 1990.

[30] H.K.N. Leung and L. White. Insights Into Regression Testing. In Proceedings of the Conference on Software Maintenance - 1989, pages 60{69, October 1989.

[31] H.K.N. Leung and L. White. Insights into testing and regression testing global variables. Journal of Software Maintenance, 2:209{222, December 1990.

[32] 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, November

1990.

[33] 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, October 1991.

[34] 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, September 1989.

[35] U. Linnenkugel and M. Mullerburg. Test data selection criteria for (software) integration testing. In Systems Integration '90. Proceedings of the First International Conference on Systems Integration, pages 709{717, April 1990.

[36] 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.

[37] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In Sixth Annual Pacific Northwest Software Quality Conference, pages 233{247, September 1988.

[38] R. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, New York, NY, 1987.

[39] G. Rothermel. Efficient, Effective Regression Testing Using Safe Test Selection Techniques. Ph.D. dissertation, Clemson University, May 1996.

[40] 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, September 1993.

[41] 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, September 1994.

[42] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. Technical Report OSU-CISRC- 4/96-TR23, The Ohio State University, February 1996.

[43] S. Schach. Software Engineering. Aksen Associates, Boston, MA, 1992.

[44] B. Sherlund and B. Korel. Modification oriented software testing. In Conference Proceedings: Quality Week 1991, pages 1{17, 1991.

[45] 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, September 1989.

[46] 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, September 1994.

[47] E.J. Weyuker. Empirical techniques for assessing testing strategies,. (Panel discussion at the International Symposium on Software Testing and Analysis), August 1994.

[48] 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, November 1992.

[49] 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, September 1993.

[50] S. Wolfram. Mathematica: A System for Doing Mathematics on a Computer. Addison-Wesley, Reading, MA, second edition, 1991.

[51] 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, April 1995.

[52] W. Yang. Identifying syntactic differences between two programs. Software|Practice and Experience, 21(7):739{ 755, July 1991.

[53] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics-preserving transformations. ACM Transactions on Software Engineering and Methodology, 1(3):311{54, July 1992.

[54] 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, October 1987.

[55] 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, October 1989.