

Regression Test Selection for C++ Software

Gregg Rothermel
Dept. of Computer Science
Oregon State University
Corvallis, OR 97331
grother@cs.orst.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
harrold@cc.gatech.edu

Jeinay Dedhia
Dept. of Computer Science
Oregon State University
Corvallis, OR 97331
dedhia@cs.orst.edu

Abstract

Regression testing is an important but expensive software maintenance activity performed with the aim of providing confidence in modified software. Regression test selection techniques reduce the cost of regression testing by selecting test cases for a modified program from a previously existing test suite. Many researchers have addressed the regression test selection problem for procedural language software, but few have addressed the problem for object-oriented software. This paper presents a regression test selection technique for use with object-oriented software. The technique constructs graph representations for software, and uses these graphs to select test cases, from the original test suite, that execute code that has been changed for the new version of the software. The technique is strictly code based, and requires no assumptions about the approach used to specify or test the software initially. The technique applies to modified and derived classes, and to application programs that use modified classes.

Keywords: software maintenance, software testing, object-oriented, regression testing, selective retest, regression test selection

1 Introduction

Regression testing is applied to modified software to provide confidence that modified code behaves as intended, and does not adversely affect the behavior of unmodified code. Regression testing plays an integral role in maintaining the quality of subsequent releases of software as that software undergoes maintenance. One characteristic that distinguishes regression testing from development testing is the availability, at regression test time, of existing test suites. By reusing such test suites to retest modified programs testers can reduce the effort required to perform that testing. However, test suites can be large, and the time and effort required to rerun all test cases in an existing test suite may be excessive. In such cases, testing efforts must be restricted to a subset of the test suite. The problem of choosing an appropriate subset of an existing test suite is the *regression test selection problem*; a technique for solving this problem is a *regression test selection technique*.

Although many researchers have addressed the regression test selection problem for procedural software (e.g., [1, 3, 5, 8, 20, 32, 36, 47, 54, 58]), far fewer have addressed the problem for object-oriented software

[17, 25, 26, 27, 44]. This is surprising, because the emphasis on code reuse in the object-oriented paradigm both increases the cost of regression testing, and provides greater potential for obtaining savings by using regression test selection techniques. When a class is modified, the modifications may impact every application program that uses that class and every class derived from that class; ideally, every such program and derived class should be retested [50, 56]. The object-oriented paradigm also raises different concerns for regression test selection algorithms. For example, because most classes consist of small interacting methods, regression test selection techniques for object-oriented software must function on interacting routines — that is, at the *interprocedural* level. Furthermore, because many techniques for testing object-oriented software (e.g., [10, 17, 23, 50, 53]) treat classes as a primary unit of test, regression test selection techniques must be applicable to class testing.

This paper presents a regression test selection technique that addresses the regression test selection problem for C++ software. The technique constructs control flow representations for classes and programs that use classes; it uses those representations to select test cases, from an existing test suite, that execute code that has been changed for a new version of the software.

The technique has several advantages. It can be fully automated, it operates interprocedurally, and it performs test selection for C++ application programs, classes, and derived classes. The technique handles both structural and nonstructural program modifications and processes multiple modifications with a single application of the algorithm. The approach selects test cases that may now execute new or modified code, and test cases that formerly executed code that has been deleted from the original program. The technique selects test cases using information gathered by code analysis but without requiring software specifications, and it is independent of the approach used to generate test cases initially for programs and classes.

The next section provides background information required for the rest of the paper. Section 3 describes the basic algorithm for selecting regression tests for modified C++ application programs, and Section 4 shows how that algorithm can be used to select regression tests for modified or derived classes. Section 5 shows how the approach accommodates language features important in the object-oriented paradigm, such as polymorphism and dynamic binding and the passing of objects as parameters. Section 6 describes empirical studies in which the technique is applied to a commercially available C++ class library. Section 7 discusses the relation of this work to previous work, and Section 8 presents conclusions.

2 Background

The following notation is used throughout the rest of this paper. Let P be a method, class, or program, let P' be a modified version of P , and let T be a set of test cases (a *test suite*) created to test P .

2.1 Control Flow Graphs

A *control flow graph* (CFG) for method 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 method `go` and its CFG.¹ In the figure, *statement nodes*, shown as ellipses, represent simple statements. *Predicate nodes*, shown

¹In Figure 1, and other figures in this paper, the code and node numbering scheme was chosen for expository convenience. For example, lines of code in `go` are numbered 30–54 because `go` is part of an application program to be presented later.

```

30 virtual void go (int floor) {
31     int valid = valid_floor(floor);
32     if (!valid_floor(floor)) {
33         cout << "Invalid floor request\n";
34         return;
35     }
36     if (floor > current_floor) {
37         up();
38         cout << "Elevator is going up";
39     }
40     else if (floor < current_floor) {
41         down();
42         cout << "Elevator is going down";
43     }
44     else
45         return;
46     if (current_direction == UP) {
47         while ((current_floor != floor)
48             && (current_floor <= top_floor))
49             add(current_floor, 1);
50     }
51     else {
52         while ((current_floor != floor)
53             && (current_floor <= bottom_floor))
54             add(current_floor, -1);
55     }
56 };

```

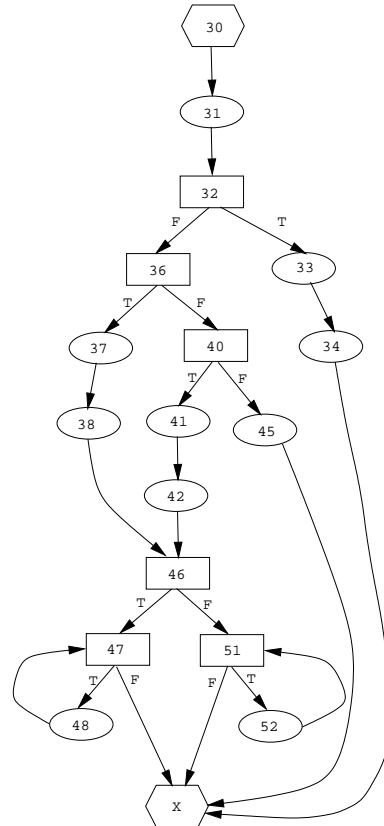


Figure 1: Method go and its CFG.

as rectangles, represent conditional statements. *Branches*, shown as labeled edges leaving predicate nodes, represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in P to which they correspond. The figure uses statement numbers as node labels; however, the actual code of the associated statements could also serve as labels. For simplicity, switch statements are represented in CFGs as nested if-else statements; under this assumption, every CFG node has either one unlabeled out-edge or two out-edges labeled “T” and “F”.² A unique *entry node*, labeled with a statement number, and a unique *exit node*, labeled “X”, represent entry to and exit from P , respectively. Declarations and nonexecutable initialization statements, when present, are also represented as nodes. The time and space required to construct and store a CFG for method P is linear in the number of simple and conditional statements in P [2].

2.2 Interprocedural Control Flow Graphs

A CFG encodes control flow information for a single method. To encode control flow for a group of interacting methods that have a single entry point, such as a group of methods that constitute an entire program, an

²Reference [47] describes an alternative representation for switch statements that allows more precise test selection.

```

1  #include <iostream.h>
2  #include <stdlib.h>
3  #define UP 1
4  #define DOWN 2
5  typedef int Direction;
6
7  class Elevator {
8  public:
9      Elevator (int l_top_floor) {
10         current_floor = 1;
11         current_direction = UP;
12         top_floor = l_top_floor;
13         bottom_floor = 1;
14     }
15
16     virtual ~Elevator() {}
17
18     void up() {
19         current_direction = UP;
20     }
21
22     void down() {
23         current_direction = DOWN;
24     }
25
26     Direction direction() {
27         return current_direction;
28     }
29
55
56 private:
57     add(int &a, const int &b) {
58         a = a+b;
59     }
60
61     int valid_floor(int floor) {
62         if ((floor > top_floor) ||
63             (floor < bottom_floor))
64             return 0;
65         return 1;
66     };
67 protected:
68     int current_floor;
69     Direction current_direction;
70     int top_floor;
71     int bottom_floor;
72 };
73
74 void main (int argc, char **argv) {
75     Elevator *e_ptr;
76
77     e_ptr = new Elevator(10);
78     e_ptr->go(2);
79 }

```

Figure 2: Code for an Elevator class, and a main routine that utilizes that class. Lines 30-54, which give the code for the `go` method, were shown in Figure 1, and thus are omitted here.

interprocedural control flow graph (ICFG) [28] is used. An ICFG for program P contains a control flow graph for each method in P , with each call site in P represented as a pair of nodes called *call* and *return* nodes. Each call node is connected to the entry node of the called method by a *call edge*, and each exit node is connected to the return node of the calling method by a *return edge*.

To illustrate, Figure 2 shows an `Elevator` class, and a `main` routine that uses methods in that class. The program created by compiling `Elevator` and linking it with `main` is hereafter referred to as “ElevatorApp”. Figure 3 displays the ICFG for `ElevatorApp`.

2.3 Code Instrumentation

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

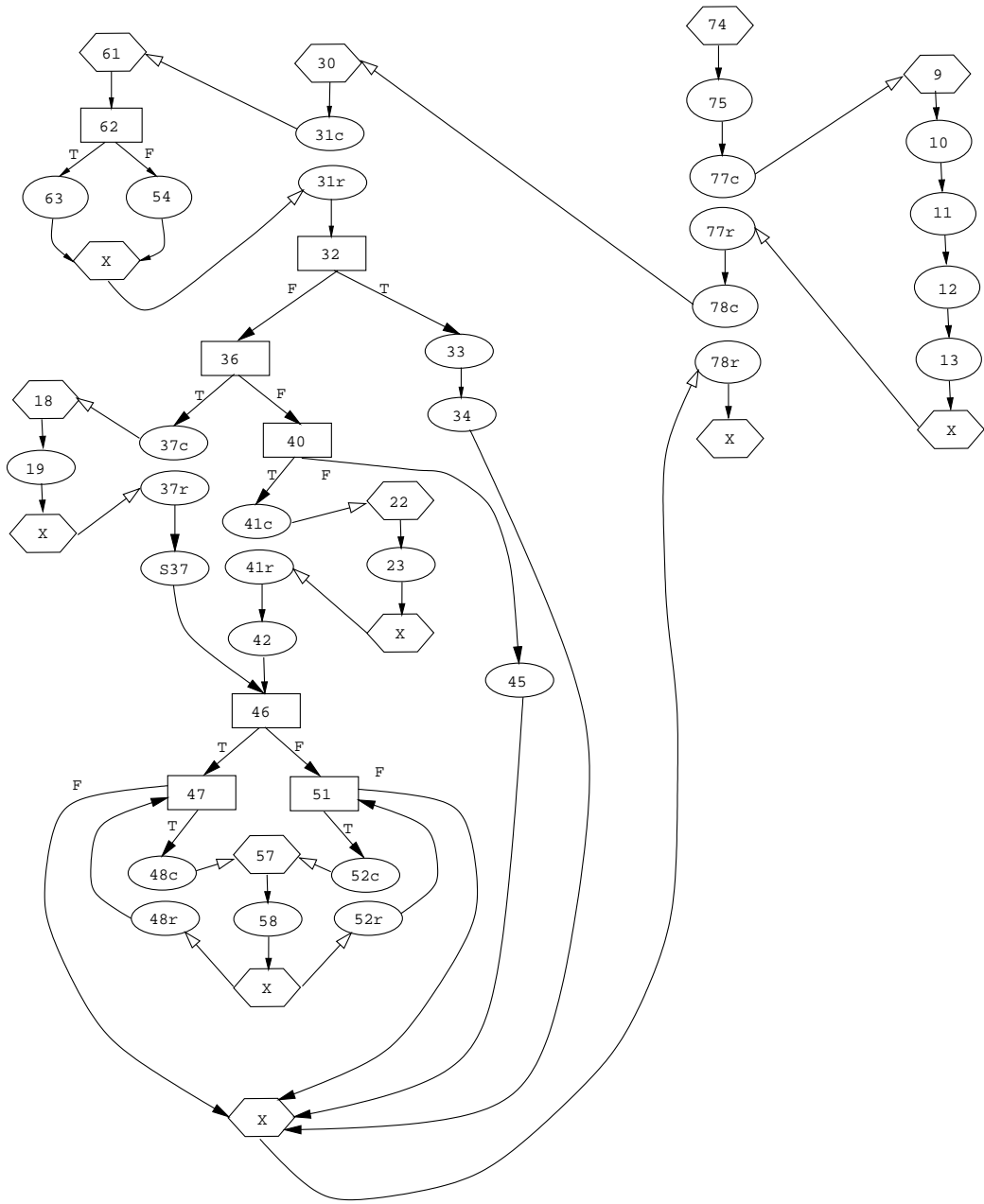


Figure 3: ICFG for program ElevatorApp.

TEST HISTORY INFORMATION

test	top_floor	bottom_floor	current_floor	floor	edge trace
t1	10	1	1	-1	(30,31),(31,32),(32,33),(33,34),(34,X)
t2	10	1	1	5	(30,31),(31,32),(32,36),(36,37),(37,38),(38,46),(46,47),(47,48),(48,47),(47,X)
t3	10	1	5	2	(30,31),(31,32),(32,36),(36,45),(45,41),(41,42),(42,46),(46,51),(51,52),(52,51),(51,X)
t4	10	1	2	2	(30,31),(31,32),(32,36),(36,40),(40,45),(45,X)

Figure 4: Test history information for `go`.

be represented by a bit vector.³

Given test suite T for P , a *test history* is constructed for P with respect to T by gathering edge trace information for each test case in T and representing it such that for each edge (n_1, n_2) in G , the test history records the test cases that traverse (n_1, n_2) . This representation requires $O(e|T|)$ bits, where e is the number of edges in G and $|T|$ is the number of test cases in T .

To illustrate, Figure 4 depicts test history information for the `go` method, for a suite of four test cases. It is assumed that a test driver sets the values of `top_floor`, `bottom_floor`, and `current_floor` as indicated, then invokes `go` with the indicated value of `floor` as parameter. An analogous approach could record test history information for the entire `ElevatorApp` application in terms of ICFG edges.

For convenience, the existence of function `TestsOnEdge(n_1, n_2)` is assumed; this function returns a bit vector v of size $|T|$ bits such that the k th bit in v is set if test case k in T traversed edge (n_1, n_2) in G .

2.4 Regression Testing

Previous research on regression testing has addressed many topics, including test environments and automation [6, 9, 22, 61], capture playback mechanisms [34], test suite management [16, 21, 34, 51, 59], program size reduction [4], and regression testability [30]. Most recent research on regression testing, however, concerns *selective retest techniques* (e.g., [1, 3, 5, 8, 14, 15, 20, 21, 25, 26, 27, 32, 36, 43, 45, 49, 51, 54, 55, 58, 60]).⁴

Selective retest techniques reduce the cost of regression testing by reusing existing test cases, 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 test cases in the existing test suite. Leung and White [33] show that a selective retest technique is more economical than the retest all technique only if the cost of selecting a reduced subset of test cases is less than the cost of running the test cases that the selective retest technique omits.

A typical selective retest technique proceeds as follows:

1. Select $T' \subseteq T$, a set of test cases to execute on P' .
2. Test P' with T' , establishing P' 's correctness with respect to T' .

³Encoding traces as bit vectors discards information on execution order and on the number of times that branches were executed. Such information could conceivably be useful in regression test selection, but its encoding and processing would inflate the cost of a test selection algorithm that utilized it. As shown later, the bit vector approach is sufficient for purposes of the algorithms presented in this paper; moreover, in all empirical data gathered to date on the use of the approach in practice, the bit vector approach has proven as precise as an approach involving more extensive trace encoding [47].

⁴For a comprehensive list and review of the research, see [46].

3. If necessary, create T'' , a set of new functional or structural test cases for P' .
4. Test P' with T'' , establishing P' 's correctness with respect to T'' .
5. Create T''' , a new test suite and test history for P' , from T , T' , and T'' .

In performing these steps, a selective retest technique addresses several problems. Step 1 involves the *regression test selection problem*: the problem of selecting a subset T' of T with which to test P' . This problem includes the subproblem of identifying test cases in T that are *obsolete* for P' . Test t is obsolete for P' if t specifies an input to P' that, according to S' , is invalid for P' , or t specifies an invalid input-output relation for P' . Step 3 addresses the *coverage identification problem*: the problem of identifying portions of P' or S' that require additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing test cases 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, this paper considers only the regression test selection problem. The focus is further restricted to code based regression test selection techniques, which rely on analysis of P and P' to select test cases.

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

2.5 Testing and regression testing object-oriented software.

Object-oriented application programs, like other application programs, require testing. To test object-oriented software properly, however, a technique for testing classes [39] is also required. Class testing techniques typically invoke sequences of methods in varying orders, and after each sequence, verify that the resulting state of the objects manipulated by the methods is correct [10, 17, 53].

To perform class testing, a typical approach uses a *test driver* that invokes a sequence of methods. Such a driver first performs setup chores, calling constructor routines and other initialization methods. Next, the driver invokes the sequence of methods under test. Finally, the driver invokes an “oracle” method that verifies that objects have attained proper states.

When a class is modified, it is necessary to retest that class itself, and classes derived from that class [17, 39]. Unfortunately, it has been shown that a function that has been adequately tested in isolation may not be adequately tested in combination with other functions [39, 56]. Thus, it is advisable to also test application programs that use the modified class. In practice, of course, it may be impractical or impossible to retest all such application programs; nevertheless, testers need to be aware of the consequences of this impracticality, and testers may wish to reduce the associated risks by regression testing those application programs that can economically be tested.

The sections that follow address the regression test selection problem at all three of these levels of testing: application program testing, class testing, and derived class testing.

3 Selecting regression tests for modified application programs

Reference [47] presented a technique, based on theoretical foundations presented in Reference [46], for selecting regression tests for procedural software. That technique includes both intraprocedural and interprocedural algorithms. Those algorithms, however, operate on individual CFGs. In this paper, the technique of [47] is modified to provide an algorithm that functions on ICFGs and supports regression test selection for C++ application programs. As will be shown in Section 4, with these modifications and the aid of an alternative representation for classes, the new algorithm also performs regression test selection for modified and derived C++ classes.

The presentation begins with an informal description of and motivation for the overall approach used by the algorithm. (The motivation and overall approach are similar to those presented in Reference [47]; more formal aspects of the approach are described there.) The algorithm is then presented, and its application is illustrated on an example.

3.1 The overall approach.

Two assumptions are made. First, it is assumed that T contains no obsolete test cases, either because it contained none initially, as in purely corrective maintenance, or because they have been removed. This assumption is reasonable because it is required for regression testing of any form: if test case obsolescence cannot be determined, test correctness cannot be judged. Second, it is assumed that each test case in T terminated when run on P , and produced its specified output. This assumption too is reasonable, because if any test cases violate it (as in practice they may, because software is often released with known bugs) these test cases are known, and can be removed from T prior to performing regression test selection and then reincluded in T' following the selection phase.

The goal is to identify test cases in T that execute changed code with respect to P and P' . In other words, it is desired to identify test cases in T that (1) execute code that is new or modified for P' or (2) executed code in P that is no longer present in P' . To formalize the notion of these test cases, an *execution trace* for test case t on P is defined to consist of the sequence of statements in P that are executed when P is executed with t . Two execution traces are *equivalent* if they have the same lengths and if, when their elements are compared from first to last, the text representing the pairs of corresponding elements is equivalent when white space is ignored. A test case t is said to be *modification traversing* for P and P' if its execution traces from P and P' are nonequivalent.

In general, it may be inordinately expensive to compare execution traces, and an algorithm that attempted it would be required to run all test cases in T on P' (exactly what testers wish to avoid) in order to obtain their traces and select a subset of T . It is possible to take advantage, however, of the mapping between execution traces and paths in ICFGs obtained by replacing each statement in the execution trace with its corresponding ICFG node (or equivalently, with the text of the statement associated with that node) to ensure selection of

test cases that are modification traversing. In this context, given programs P and P' with ICFGs G and G' , respectively, an algorithm can synchronously traverse ICFG paths that begin with the entry nodes of G and G' , looking for pairs of nodes N and N' whose associated statements are not lexicographically equivalent. When the algorithm finds such a pair, it uses test history information, obtained through the `TestsOnEdge` function, to select all test cases known to have reached N . This traversal essentially considers all test cases at once; it is not necessary to perform one traversal per test case. By marking nodes “visited” as it performs the traversal, the algorithm avoids visiting nodes multiple times, and is ensured to terminate in time proportional to the size of the graph, rather than to the size of the execution traces.

Thus far, the focus has been on changes in *executable* statements; but this is insufficient. A change in a nonexecutable variable or type declaration may cause a test case to reveal a fault, even though that test case executes no changed executable statements. For example, in a C++ program, changing the type of a variable from `int` to `double` can cause the program to fail even if no executable statements are altered. To handle this situation, one approach is to create, in ICFGs, `declaration` nodes that correspond to variable and type declarations, and associate each test case that executes a method, class, or program with all `declaration` nodes contained in the representation for that method, class or program. Following this approach, when nonexecutable initialization statements change, all test cases attached to the associated `declaration` nodes are selected. The figures in this paper illustrate the use of these nodes with respect to declarations that occur within methods: for example, see node 31 in Figure 1 and node 75 in Figure 3. Declaration or type statements associated with a class rather than an individual method, such as statements 5 and 68 – 71 in the code for Elevator shown in Figure 2, can be represented as nodes within the constructor method for the class (these nodes are omitted from the figures); changes in these statements will prompt selection of all test cases that cause an object of that class to be instantiated. Section 5 discusses this issue further and presents an alternative approach.

3.2 A regression test selection algorithm.

Figure 5 presents regression test selection algorithm `SelectTests`. `SelectTests` takes a program P , its modified version P' , and the test suite T for P , and returns T' , a set that contains test cases that are modification traversing for P and P' . `SelectTests` first initializes T' to \emptyset , and initializes E , which will hold the set of edges in the ICFG for P on which test cases must be selected, to \emptyset . Next, the algorithm constructs ICFGs G (with entry node e) and G' (with entry node e') for P and P' , respectively. The algorithm then calls `Compare` with e and e' . `Compare` ultimately places edges through which test cases become modification traversing for P and P' into E . `SelectTests` then uses `TestsOnEdge` to retrieve these test cases (lines 6–7).

`Compare` is a recursive procedure called with pairs of nodes N and N' , from G and G' , respectively, that are reached simultaneously during the algorithm’s walk of the graphs. Given two such nodes N and N' , `Compare` first determines whether the two nodes have equivalent outgoing edges. If not, then test cases that reach N' may become modification traversing. Thus, `Compare` selects all edges out of N : the test cases on these edges include all test cases of interest. Note that if N and N' represent typical true-valued or false-valued predicate statements, then each node necessarily has a pair of outgoing edges labeled “true” and “false”, and

```

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

1. begin
2.    $T' = \emptyset$ 
3.    $E = \emptyset$ 
4.   construct  $G$  and  $G'$ , ICFGs for  $P$  and  $P'$ , with entry nodes  $e$  and  $e'$ 
5.   Compare( $e, e'$ )
6.   for each edge  $(n_1, n_2) \in E$  do
7.      $T' = T' \cup \text{TestsOnEdge}((n_1, n_2))$ 
8.   return  $T'$ 
9. end

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

10. begin
11.   mark  $N$  “ $N'$ -visited”
12.   if  $\neg \text{OutEdgesEquivalent}(N, N')$ 
13.     for each successor  $C$  of  $N$  in  $G$  do
14.        $E = E \cup (N, C)$ 
15.     endfor
16.   else
17.     for each successor  $C$  of  $N$  in  $G$  do
18.        $L$  = the label on edge  $(N, C)$  or  $\epsilon$  if  $(N, C)$  is unlabeled
19.        $C'$  = the node in  $G'$  such that  $(N', C')$  has label  $L$ 
20.       if  $C$  is not marked “ $C'$ -visited”
21.         if  $\neg \text{NodesEquivalent}(C, C')$ 
22.            $E = E \cup (N, C)$ 
23.         else
24.           Compare( $C, C'$ )
25.         endif
26.       endif
27.     endfor
28.   endif
29. end

```

Figure 5: Algorithm for test selection for object-oriented software.

Compare will not need to select any edges. This step in the algorithm is necessary, however, to accommodate polymorphic calls; further discussion of this issue is postponed to Section 5.⁵

If N and N' have equivalent outgoing edges, Compare considers successor nodes of N and N' to determine whether N and N' have successor nodes whose labels differ along pairs of identically labeled edges. If N and N' have any such successors, test cases that traverse the edges to the successors are modification traversing due to changes in the code associated with those successors. In this case, Compare selects the edge in G that connects N to that successor. If N and N' have successors whose labels are the same along a pair of identically labeled edges, Compare continues along the edges in G and G' by invoking itself on those successors.

Lines 10–29 of Figure 5 describe Compare’s actions more precisely. When Compare is called with ICFG nodes N and N' , Compare first marks node N “ N' -visited” (line 11). After Compare has been called once with N and N' it does not need to consider them again – this marking step lets Compare avoid revisiting pairs of

⁵Although it is not discussed here, this step is also necessary to accommodate an alternative representation of switch statements as multi-branched predicates, discussed in [47], that allows more precise test selection with respect to changes in switch cases.

nodes. (Lists to hold “visited” information are associated with ICFG nodes in G , and created and initialized during ICFG construction.) Next, `Compare` uses `OutEdgesEquivalent(N, N')` to check for equivalence of outgoing edges. This function returns true only if there is a one-to-one-correspondence between the labels on edges leaving N and the labels on edges leaving N' ; if there is no such correspondence the function returns false and lines 13–15 select the necessary edges. If N and N' have equivalent outgoing edges, `Compare`, in the `for` loop of lines 17-27, considers each control flow successor of N . For each successor C , `Compare` locates the label L on the edge from N to C , then seeks the node C' in G' such that (N', C') has label L ; if (N, C) is unlabeled ϵ is used for the edge label. Next, `Compare` considers C and 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 `NodesEquivalent` with C and C' . The `NodesEquivalent` function takes a pair of nodes N and N' , and determines whether the statements S and S' associated with N and N' are lexicographically equivalent. If `NodesEquivalent(C, C')` is false, then test cases that traverse edge (N, C) are modification traversing for P and P' , and test cases that executed C must be selected. In this case, `Compare` adds edge (N, C) to E , the list of edges on which test cases will be selected. If `NodesEquivalent(C, C')` is true, `Compare` invokes itself on C and C' to continue the graph traversals beyond these nodes.

`SelectTests` is an *interprocedural* algorithm: it functions on entire programs rather than single methods. By beginning with the entry nodes of “main” routines, and processing called methods only when it reaches calls to those methods, `SelectTests` avoids analyzing methods called only after code changes.

To see how `SelectTests` works, consider the following example. Suppose the `Elevator` class (originally presented in Figure 2) is modified, creating `Elevator'`; the new and modified code is shown as an inset in Figure 6. In `Elevator'`, the `go` method has changed: line 40 has been (erroneously) altered, and a new line, 44.1, has been added. Also, a new method, `which_floor`, has been added. Suppose further that the test set T for `ElevatorApp` contains at least test cases $t1$, $t2$, $t3$, and $t4$ that act, with respect to method `go`, like the four test cases described in Figure 4. The goal is to select test cases from T for re-execution when `ElevatorApp` is built with the modified version of the `Elevator` class, yielding `ElevatorApp'`.

In this case, `SelectTests` is called with `ElevatorApp`, `ElevatorApp'`, and T . `SelectTests` first constructs the ICFGs G and G' for the two programs. Figure 6 depicts G' ; G' differs from G only with respect to the CFG for `go`; changed portions are enclosed in dashed rectangles. Because `which_floor` is not invoked by `ElevatorApp'`, its CFG is not needed or included in the ICFG for `ElevatorApp'`.

Next, `SelectTests` invokes `Compare` with entry nodes 74 and 74'. `Compare` begins to visit pairs of nodes: $(74, 74')$, $(75, 75')$, $(77c, 77c')$, $(9, 9')$, $(10, 10')$, $(11, 11')$, $(12, 12')$, $(13, 13')$, (X, X') , $(77r, 77r')$, $(78c, 78c')$, $(30, 30')$, $(31c, 31c')$, $(61, 61')$, $(62, 62')$, $(63, 63')$, $(64, 64')$, (X, X') , $(31r, 31r')$, $(32, 32')$, and then $(36, 36')$. On visiting $(36, 36')$, `Compare` first marks 36 “36'-visited”, and then considers the successor of 36, 40. The successor in G' with an equivalently labeled edge (“F”) is 40'. 40 is not marked “40'-visited”, so `Compare` proceeds to step 21. 40 and 40' are not lexicographically equivalent; thus, `Compare` inserts edge $(36, 40)$ into E . The algorithm does not continue its traversal beyond 40 and 40'; instead, it traverses other portions of the ICFGs. No further changes are discovered; on return to the algorithm’s main routine, test cases $t3$ and $t4$ are the only test cases selected. If the change at 40 had not been present, the algorithm would have continued its traversal from 40

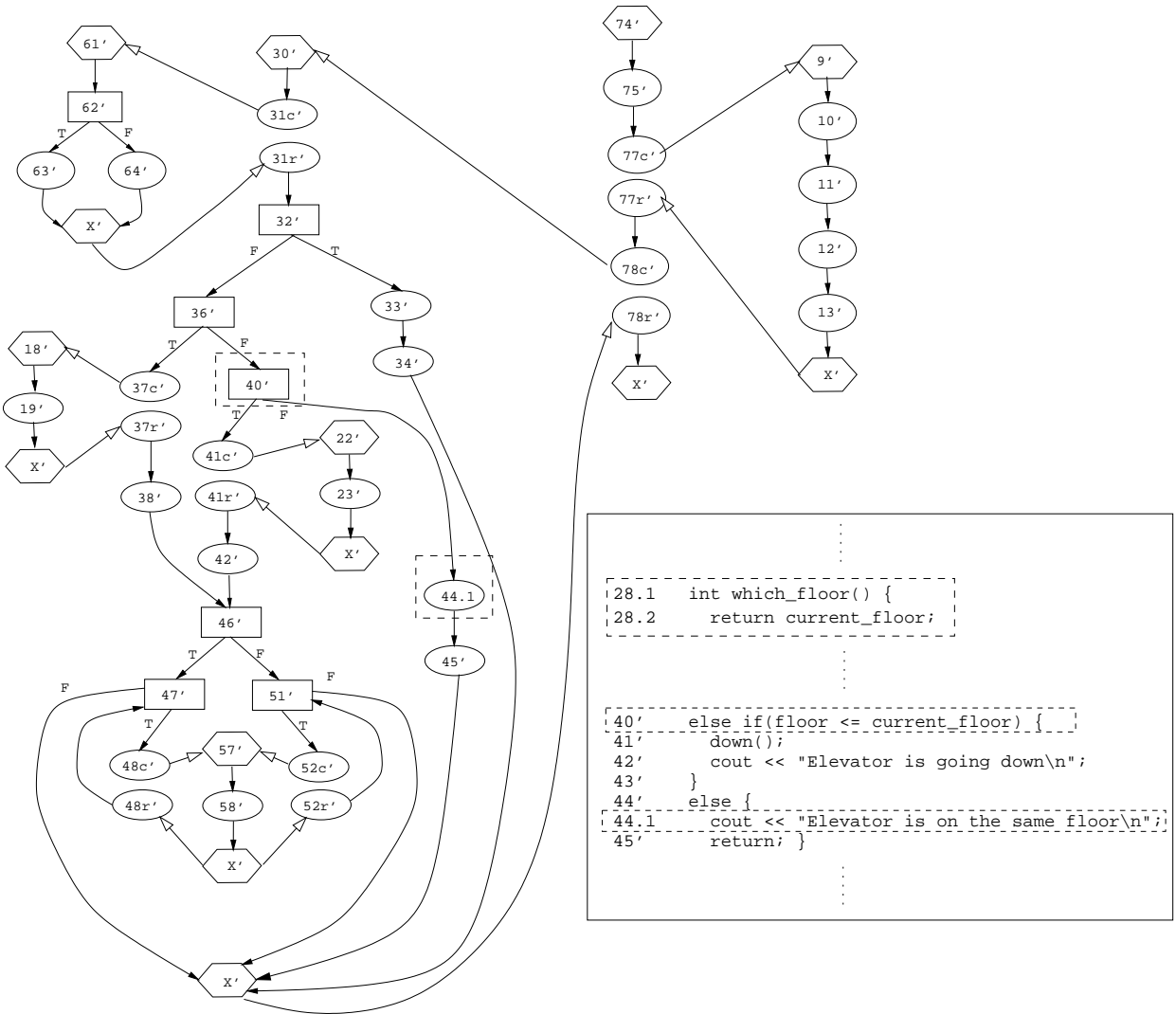


Figure 6: Code modifications made to Elevator to create Elevator', and the ICFG for ElevatorApp', with changes enclosed in dashed rectangles.

and 40', found their successors 45 and 44.1 lexicographically different, and selected edge (40,45) (requiring only test case *t4*).

Reference [47] shows why this approach must, at line 11, mark N “ N' -visited” rather than just “visited”: without this action, in certain cases, the algorithm can fail to detect test cases that are modification traversing for P and P' . As written, the algorithm does not miss such test cases.

The running time of `SelectTests` is bounded by the sum of (1) the time required to construct ICFGs G and G' for P and P' , respectively (line 4), (2) the number and cost of calls to `Compare` (line 5), and (3) the cost of the set unions (lines 6–7). Let n be the number of statements in P , and let n' be the number of statements in P' . ICFGs are constructed in time $O(n)$ and $O(n')$ (including the cost of initializing visited lists). A bound on the cost of the set unions is $O(n|T'|)$. An upper bound on the number of calls to `Compare` is obtained by assuming that `Compare` can be called with each pair of nodes N and N' in G and G' , respectively. Assuming that the size of the strings and labels compared by `NodesEquivalent` and `OutEdgesEquivalent`, and the number of successors of a node, are bounded by constants, and noting that the set union at line 22 can be accomplished by merely adding (N, C) to E without checking for its presence in E (because no edge in G can be considered for inclusion in E more than once), it follows that each call to `Compare` requires constant work. Thus the run time of `SelectTests` is $O(nn' + n|T'| + n + n')$, which simplifies to $O(nn' + n|T'|)$.

In certain cases, `SelectTests` can select test cases that are not modification traversing for P and P' . However, the problem of precisely identifying the modification traversing test cases, in general, is PSPACE-hard [42]. Thus, unless $P=NP$, no efficient algorithm will always identify precisely the test cases that are modification traversing for P and P' . Reference [47] identifies a necessary condition, the *multiply visited node condition*, for graph walk algorithms such as `SelectTests` to be imprecise. Empirical results obtained with an implementation of a version of the algorithm developed for non-object-oriented programs, however, have never yet revealed a case where this condition held in practice. When the multiply visited node condition does not hold, the first term in the runtime analysis of `SelectTests` listed above reduces to $\min\{n, n'\}$.

Having selected a test suite with `SelectTests`, an important question involves whether that test suite might omit any test cases that could expose faults in the modified program. *Controlled regression testing* [46] is the practice of testing P' under conditions equivalent to those that were used to test P . Controlled regression testing applies the scientific method to regression testing: to determine whether code modifications cause errors, test the new code, holding all other factors that might affect program behavior constant. For the purpose of regression test selection, a goal is to identify all test cases $t \in T$ that reveal faults in P' – the *fault revealing* test cases. An algorithm that selects every fault revealing test case in T is *safe*. There is no effective procedure that, in general, precisely identifies the fault revealing test cases in T [42]. However, under controlled regression testing, the modification traversing test cases are a superset of the fault revealing test cases [42]. Thus, for controlled regression testing, a regression test selection algorithm that selects all modification traversing test cases is safe. This result is significant, because it supports a proof that test selection algorithms based on graph walks, such as `SelectTests`, are safe for controlled regression testing [47].

4 Selecting regression tests for modified and derived classes

When a class is modified, testers want to identify test cases in the class’s test suite that should be reexecuted. Similarly, when a new class is derived from a base class, testers want to identify the test cases in the base class’s

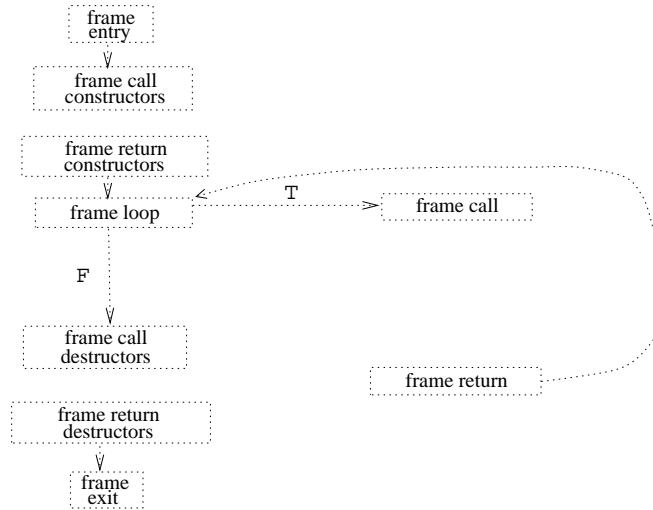


Figure 7: A CCFG frame, prior to inclusion in a CCFG.

test suite that should be reexecuted on the derived class. The regression test selection technique described in Section 3 does not apply directly to classes, because it requires a program to have a single entry point, whereas a class can have multiple entry points. By using a new abstract model for classes, however, it is possible to depict control flow within a class in a manner that lets `SelectTests` select regression tests for modified and derived C++ classes.

4.1 An abstract model for representing C++ classes

To understand the approach, recall from the discussion in Section 2.5 that to test a class, it is usual to create a number of driver programs. These driver programs assume or instantiate some initial state, and then invoke sequences of methods. To perform regression test selection on a modified class, a naive approach is to treat each driver as an application program and apply `SelectTests` to it. This approach, however, has a disadvantage: it requires construction and traversal of individual ICFGs for each driver.

To more efficiently address this problem, a new representation for C++ classes, the class control flow graph (CCFG), is defined. A CCFG is a collection of individual control flow graphs for the methods in a class, in which call sites are expanded and connected to called methods in the same manner as in an ICFG, and to which a *frame* has been added. A frame (see Figure 7) is an abstraction of a driver program, that simulates arbitrary sequences of calls to public methods.⁶ A frame for a CCFG consists initially of nine vertices: *frame entry* and *frame exit*, which represent entry to and exit from the frame; *frame call constructors* and *frame return constructors*, which represent calls to, and returns from, class constructor methods; *frame call destructors* and *frame return destructors*, which represent calls to, and returns from, class destructor methods; *frame loop*, which facilitates sequencing of methods; and *frame call* and *frame return*, which represent calls to and returns

⁶With modifications, frames can actually yield a variety of graphical representations for classes, including, in addition to CCFGs, *class call graphs* and *class dependence graphs* [19]. Framed graphs have also been used for purposes other than regression test selection, including dataflow analysis and dataflow testing of classes [18] and slicing of classes [29].

```

algorithm ConstructCCFG(C):G
input      C: a class.
output    G: the CCFG for C
declare   frame : set of frame nodes and edges
begin ConstructCCFG
    Build control flow graphs for methods in C and add them to G
    Add the frame to G
    Connect constructor methods in G to frame nodes
    Connect destructor methods in G to frame nodes
    Connect public methods in G to frame nodes
    Replace remaining call sites in G with call and return nodes
    Add edges connecting call and return nodes in G to appropriate entry and exit nodes in G
    Return the completed CCFG
end ConstructCCFG

```

Figure 8: Algorithm for constructing a class control flow graph (CCFG).

from public methods other than constructors or destructors. Frame edges connect the vertices.

Figure 8 presents an algorithm for constructing a CCFG for a class *C* in worst-case time linear in the size of *C*. To construct CCFG *G* for class *C*, the algorithm first builds control flow graphs for individual methods in *C* and adds them to *G*, and then adds the initial frame to *G*; at this point, the frame and control flow graphs are not connected. The algorithm next connects all constructor methods to the frame, by creating call and return nodes for each constructor method, and connecting these nodes to the appropriate frame and method nodes. The algorithm then performs an analogous task with destructor methods. Next, the algorithm connects public methods to the frame, by creating call and return nodes for each such method, and connecting them to appropriate frame and method nodes. At this point, all necessary method-frame connections have been established, and the algorithm considers other call sites in *C*'s methods. For each call site that invokes a method also contained in *C*,⁷ the algorithm expands that site into call and return nodes and connects those nodes to the entry and exit nodes of the called procedure, as for ICFG construction. Finally, the algorithm returns *G*, the completed CCFG. For reasons that will become clear later, the algorithm labels all edges out of *frame call*, *frame call constructor*, and *frame call destructor* nodes with the signature of the invoked method.

Figure 9 shows the framed class control flow graph for the **Elevator** class. The frame call constructors node is now connected to the entry node of the **Elevator** constructor through a call node, and the frame return constructors node is now connected to the exit node of the **Elevator** constructor via a return node. The frame call destructors node and frame return destructors node have been treated similarly. (If multiple constructors or destructors had been present, frame call constructor and destructor nodes would have been expanded into multiple call and return nodes – one pair for each method.) The frame call node has been expanded to represent calls to each of the public methods (**up**, **down**, and **go**) in **Elevator**. The **Elevator** class also contains two private methods (**add** and **valid_floor**); they are not connected to the frame call, but instead, are connected to call and return nodes at sites in other methods from which they are invoked. On

⁷In this section, attention is restricted to *intra*class testing, which considers only methods in a single class or class hierarchy. Section 5 discusses relaxation of this restriction.

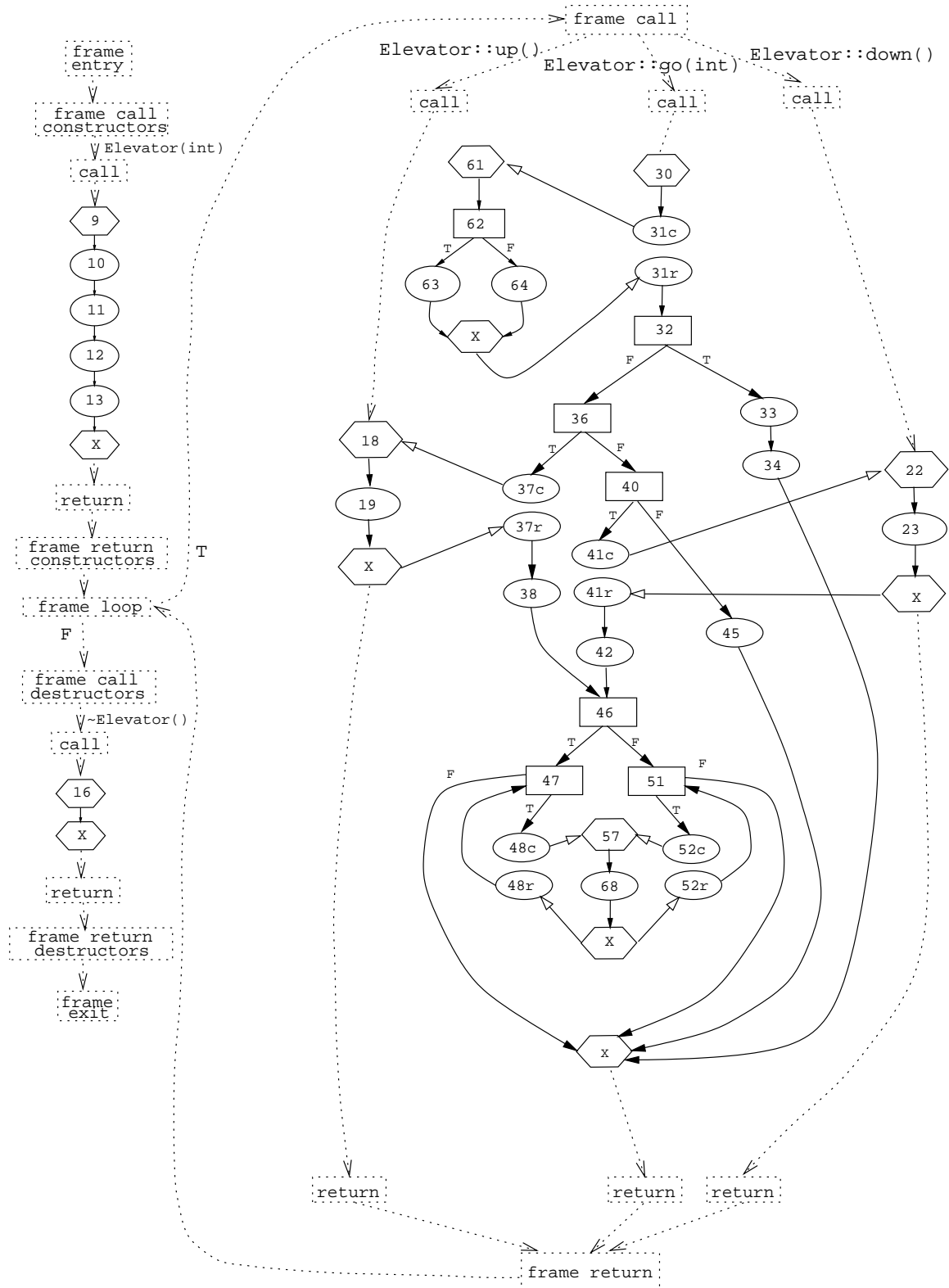


Figure 9: Framed class control flow graph for Elevator.

edges into call nodes, labels are attached, consisting of the signature of the call to the method entered.

The `Elevator` CCFG also illustrates the way in which individual CFGs in a CCFG may be connected to both a class's frame call node and to each other by call edges. For example, the `up` and `down` methods in the `Elevator` class are public methods, but they are also called from the `go` method. Thus, the CFGs for those methods (rooted at nodes 18 and 22, respectively) are connected both to the frame call and frame return nodes, and also to call and return nodes in the CFG for `go`.

When complete, the framed class control flow graph depicts flow of control as passing through constructors, and then entering a loop that contains calls to each public method in the class. On each iteration of the loop, control can pass to any of the public methods. After the loop terminates, control passes through class destructors to the frame exit node.

CCFGs are also used to represent derived classes. In essence (and potentially in implementation), such CCFGs inherit CFGs from base classes, supplying new CFGs to represent new or redefined methods.⁸ To illustrate, Figure 10 depicts the code for an `AlarmElevator` class derived from `Elevator`, and Figure 11 depicts the framed CCFG for the `AlarmElevator` class. The latter figure shows how the CCFG for `AlarmElevator` inherits CFGs for methods inherited from the `Elevator` class, and shows where a new CFG (rooted at node 106) for the new `go` method declared in the `Elevator` class would be substituted. The figure also includes appropriate constructor graph connections.

⁸Care must be taken in determining where CFGs can be inherited, to properly account for the effects of changes in inheritance hierarchies. Reference [35] provides an approach.

```

100 class AlarmElevator : public Elevator {
101 public:
102     AlarmElevator(int top_floor) : Elevator(top_floor) {
103         elevAlarm = new Alarm();
104         fire_alarm_on = 0;
105     }
106     void go(int floor) {
107         if (!(fire_alarm_on))
108             Elevator::go(floor);
109     }
110     void check_weight(int weight) {
111         if (weight > MAXWEIGHT) {
112             elevAlarm->alarm_type = WEIGHT;
113             check_alarm(elevAlarm);
114         }
115     }
116     void check_alarm(Alarm *x) {
117         if (x->alarm_type == FIRE) {
118             x->set_alarm();
119             fire_alarm_on = 1;
120         }
121         else if ((x->alarm_type == WEIGHT)
122                 && !(fire_alarm_on)) {
123             x->set_alarm();
124             sleep(10);
125             x->reset_alarm();
126         }
127         else {
128             x->reset_alarm();
129             fire_alarm_on = 0;
130         }
131     }
132     protected:
133         Alarm *elevAlarm;
134         int fire_alarm_on;
135 };
136
200 class Alarm {
201 public:
202     Alarm()
203         { alarm_on = 0; }
204     void set_alarm()
205         { alarm_on = 1; }
206     void reset_alarm()
207         { alarm_on = 0; }
208     int is_alarm_on()
209         { return alarm_on; }
210     int alarm_type;
211 protected:
212     int alarm_on;
213 };

```

Figure 10: Code for an AlarmElevator class, derived from Elevator, and for an Alarm class used by AlarmElevator.

4.2 Using the CCFG to select regression tests for classes

Having defined and constructed CCFGs for classes and for modified or derived classes, it is possible to run `SelectTests` on those CCFGs to select regression tests for those classes.

To see how `SelectTests` works on CCFGs when a class is modified, suppose the `go` method in `Elevator` is modified, yielding the `go'` method depicted in Figure 6. Suppose further that a new method, `which_floor`, is added to the class. When `SelectTests` is invoked for the old and new version of `Elevator`, it creates CCFG G shown in Figure 9, and CCFG G' shown in Figure 12. Figure 12 shows the new method, and uses dashed lines to highlight portions of G' that differ from G .

Next, the algorithm traverses the two CCFGs from frame entry nodes. On reaching the frame call nodes, it uses signatures on edges to decide which pairs of calls to visit, and which pairs of CFGs beneath them. The algorithm traverses the CCFGs up through nodes 36 and 36', where it encounters the change in node 40' and selects edge (36,40), leading to selection of test cases t2, t3, and t4. Note that the new method, `which_floor`, with entry node 26, is not paired with anything, thus the algorithm does not visit it. This is appropriate

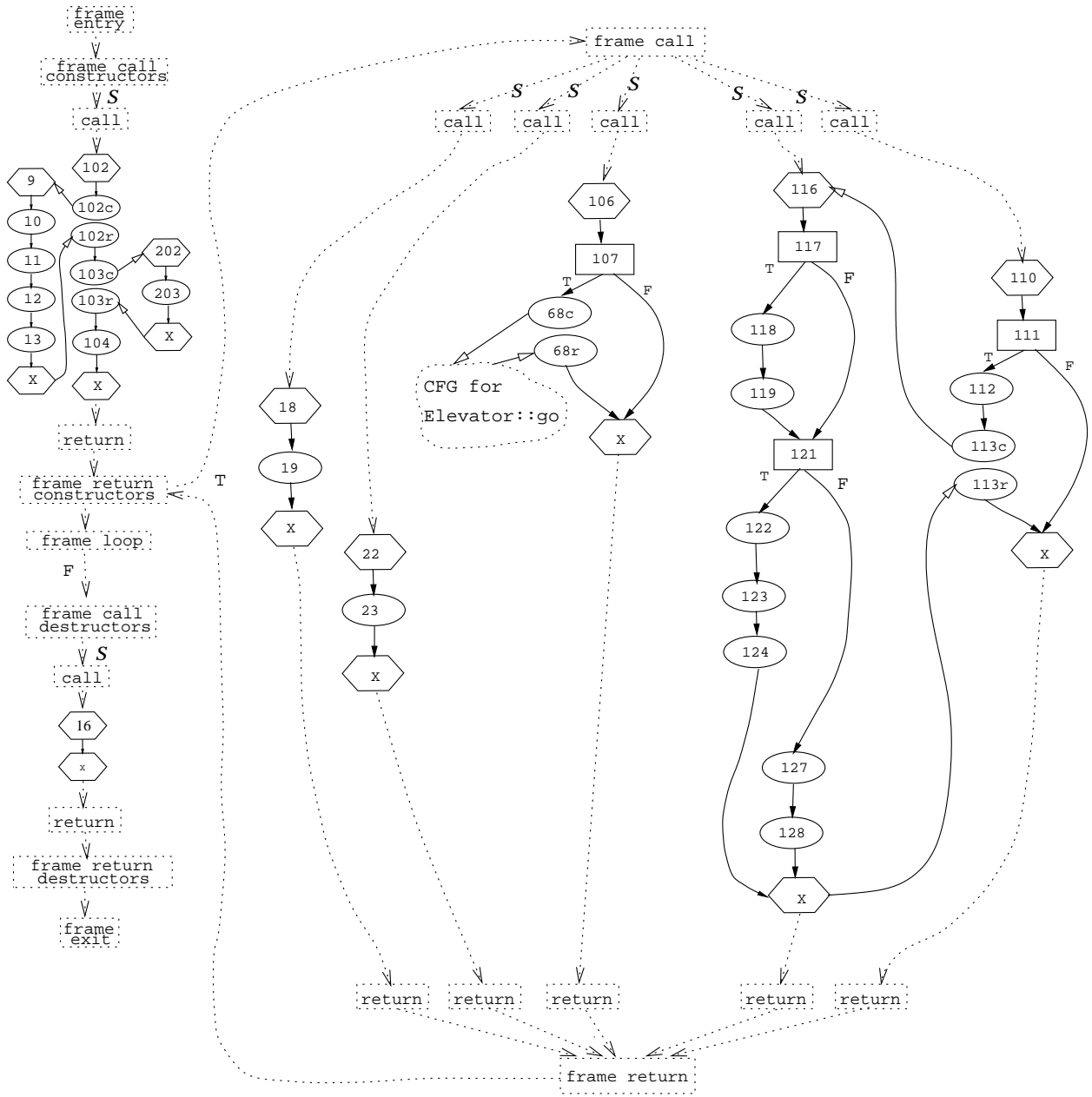


Figure 11: Framed class control flow graph for class AlarmElevator. The CFG for Elevator::go is omitted, and signatures are replaced by “S”, to simplify the presentation.

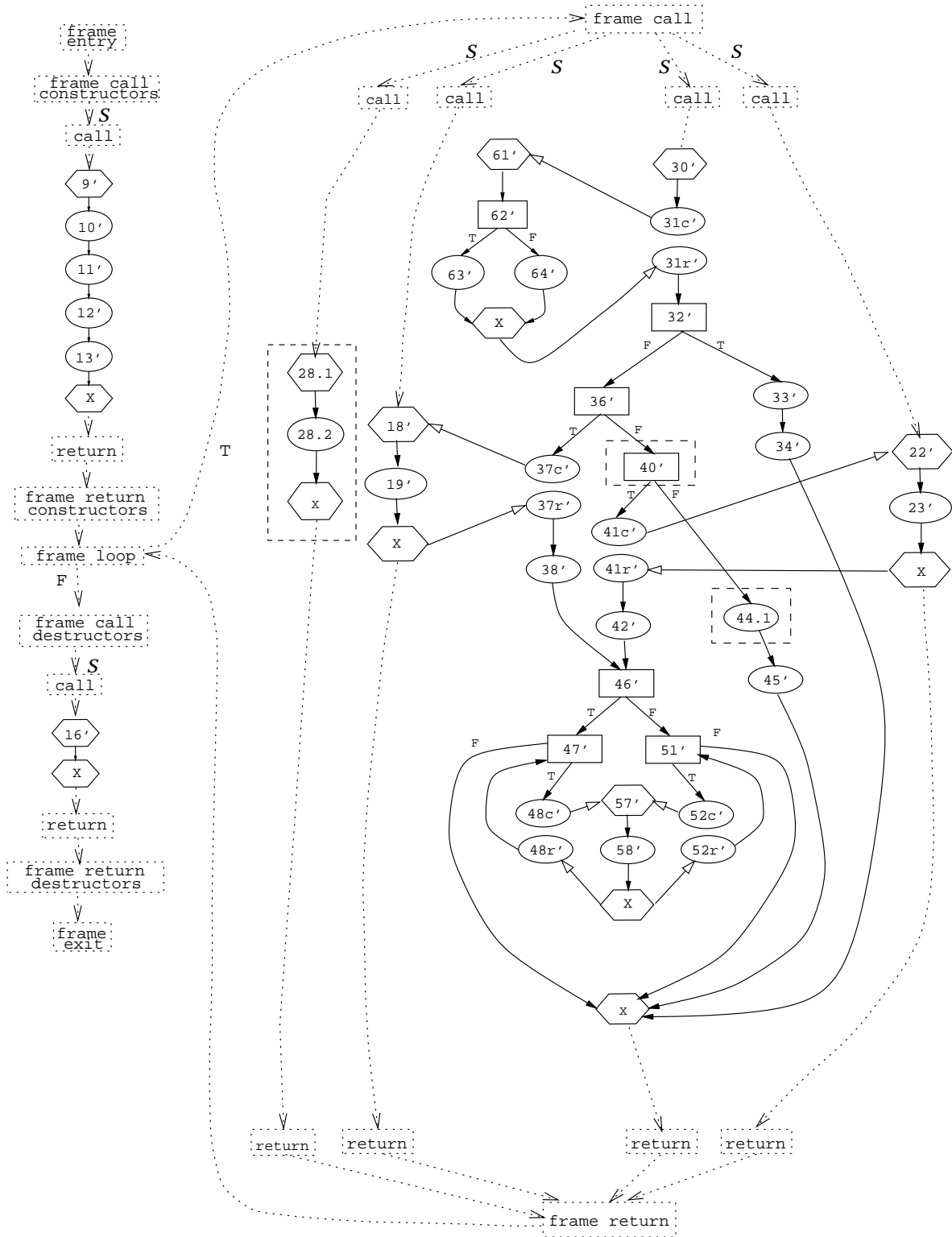


Figure 12: Modified Elevator code, and the framed class control flow graph for the modified Elevator. Signatures are replaced by “S”, to simplify the presentation.

because there are no existing test cases associated with this method. (Testers will need to develop new test cases to exercise this method.)

Now consider the case where a class is derived from `Elevator`, as in the case of `AlarmElevator` (Figure 10). The `AlarmElevator` class contains new constructor, destructor, `check_weight` and `check_alarm` methods, and redefines the `go` method. When `SelectTests` is invoked with the `Elevator` and `AlarmElevator` classes, it creates CCFGs G and G' shown in Figures 9 and 11.

With the new constructors, `SelectTests` selects all test cases almost immediately, at nodes 9 and 102. In one sense, this is appropriate, because all of these test cases execute changed code. In practice, however, this may be more test cases than can affordably be executed; also, in many cases, the constructor can be adequately tested using fewer test cases. Thus, an alternative version of `SelectTests` can avoid traversing constructor code, instead continuing its traversal from the return nodes associated with the constructor calls. In this case, in the above example, `SelectTests` selects just test cases that reach changes in `go`. At least one of these test cases must also execute the constructor. Testers might then select additional test cases of the constructor functionality. This approach is further explored in empirical studies described in Section 6.

As a final example, suppose the `AlarmElevator` class has been developed, and test suite T has been created for it. Suppose the `go` procedure in the `Elevator` class is then modified as shown in Figure 6, creating `go'`. In this case, `AlarmElevator` is not directly modified; however, its operation may be affected by the changes in `go`, and thus, it should be retested. To see which test cases in T should be rerun, the `SelectTests` algorithm is used on the `AlarmElevator` class as built with the first version of `Elevator` and the `AlarmElevator` class as built with the second version of `Elevator`. The algorithm constructs the CCFGs for the two versions of the `AlarmElevator` class; the CCFG for the new version includes the modified version of `go` and the new `which_floor` method. The algorithm then proceeds exactly as it did when it was used to select test cases to retest `Elevator`.

5 Other issues

The basic technique for selecting regression tests for C++ application programs, classes, and derived classes having been presented, it is now shown how this technique accommodates additional testing needs and language features that are important in the object oriented paradigm (and in C++).

5.1 Intraclass and interclass testing

The foregoing examples and discussion have focused on *intra*class testing, in which a single class is tested, and its interactions with classes outside its class hierarchy are not specifically considered [18]. For example, in the CCFG for `AlarmElevator` shown in Figure 11, invocations of method `Alarm::set_alarm` at lines 118 and 122 and method `Alarm::reset_alarm` at lines 124 and 127 are represented as single nodes. This is appropriate for intraclass testing of `AlarmElevator`, because these two methods are outside of the class under test.

As part of validation efforts, however, testers may also want to perform *inter*class testing [18], in which multiple classes are considered simultaneously. To accommodate regression test selection for interclass testing, it is necessary to expand all call sites to methods that belong to classes under test. For example, in the ICFG

```

300 main(int argc, char **argv) {
301     Elevator *eptr;
302     if (argv[1]) {
303         eptr = new AlarmElevator(10);
304     }
305     else {
306         eptr = new Elevator(10); }
307     eptr->go(5);
308 }

```

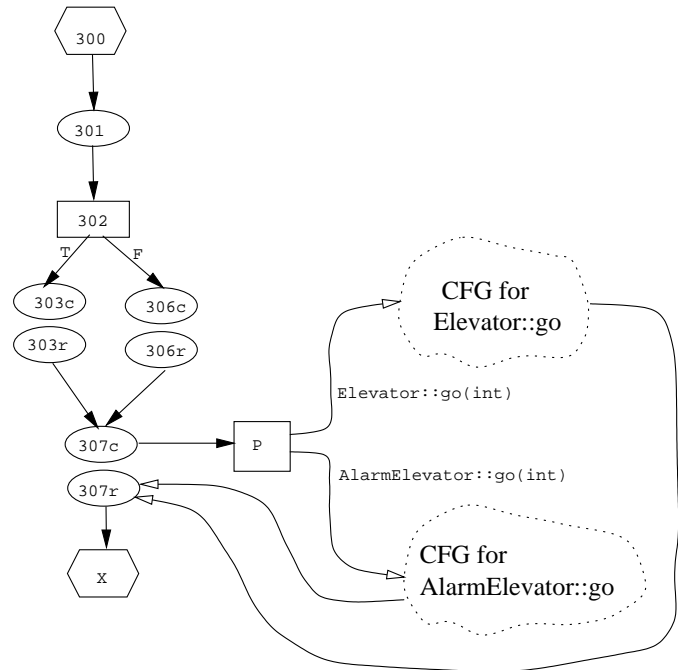


Figure 13: Polymorphic call representation.

of Figure 11, node 118 is replaced by call and return nodes, the CFG for `Alarm::set_alarm` is included, and that CFG is connected to the call and return nodes. If `Alarm` or `AlarmElevator` are modified, the ICFGs can be constructed, in this manner, for the original and modified versions, and `SelectTests` can be invoked on those ICFGs.

5.2 Polymorphism and dynamic binding

The use of polymorphism in C++ implies that method calls can invoke any of a set of methods. For example, Figure 13 shows a routine `main` that, linked with `Elevator` and `AlarmElevator`, contains a polymorphic call to method `go` (307). At this call statement, the `go` method can be dynamically bound either to the `go` method in `Elevator` or to the `go` method in `AlarmElevator`.

To accommodate polymorphic calls, graph representations are extended to include *polymorphic call nodes*. Figure 13 shows the relevant portions of the ICFG, thus extended, for the example program. In the ICFG, the polymorphic call node, labeled “P”, indicates that 307 is polymorphic. Edges from that node to the two possible CFGs for `go` are labeled to indicate their target.

To build ICFGs such as this, for each call, a set of methods it may invoke must be determined, using an algorithm such as one of those given in [11, 37, 38, 52]. The practicality of this approach depends upon the ability of static analysis algorithms to sufficiently narrow the set of methods that may be invoked at a call site. In general, the size of this set depends on the precision of the algorithm for determining it, and on how strongly typed the language being analyzed is. Empirical results suggest, however, that for strongly typed

languages such as C++, efficient algorithms exist that can obtain sufficiently precise results [37, 52].

If the set of methods that may be invoked from a call site is too large, further graph construction and traversals through that call may be impractical. In this case, two alternatives exist. First, a set of callable methods can be summarized as a single *summary node*. When `SelectTests` reaches a summary node, if the node summarizes a method that has been modified the algorithm selects all test cases through that node. (Information on modified methods can be provided by the configuration management system). Second, the set of CFGs to which the polymorphic call nodes in G and G' are linked can be limited to methods observed to be invoked during the execution of T on P . This set can be collected along with test history information during the preliminary phase of regression testing. These two approaches present different tradeoffs: the first sacrifices precision, whereas the second may sacrifice safety.

As mentioned in Section 3, it is in the handling of polymorphic calls that lines 12-15 of `SelectTests` are required. These steps permit the algorithm to detect cases where a code modification has altered the number or identity of the methods that may be invoked at a polymorphic call site, and select test cases that may be affected by this modification. Such an alteration, being due to changes in the inheritance hierarchy rather than in method signatures, would not be visible by simple inspection of the code associated with the call site.

5.3 Objects as parameters.

In C++ programs, methods can pass objects as parameters when invoking other methods. In this case, the called method may itself contain method invocations by the passed object, and the identity of the method invoked may vary, depending on the type of the object. For example, in line 113 of the `AlarmElevator` class (Figure 10), the `check_weight` method passes an object of type `Alarm` to method `check_alarm`. The `check_alarm` method takes different actions depending upon the type of `Alarm`. In the example these actions consist of invoking nonpolymorphic methods in class `Alarm`; however, one can imagine subclassing `Alarm` for the various alarm types, such that a call to `set_alarm` at line 118 would be polymorphic.

This scenario is accommodated by the approach to handling polymorphic calls described in the preceding subsection. Using an algorithm such as those of [11, 37, 38, 52] a set of methods that can be invoked from the call site can be determined, and the call site can be connected to the CFG for each such method, using a polymorphic call node if the set contains more than one element. If the set is too large, the same approximations apply.

5.4 Handling changes in nonexecutable statements more precisely

Thus far, changes in nonexecutable statements such as declarations and typedefs have been handled by an approach that involves creating a node for each such statement. Each such node is included in the representation for the method, class or program in which its source statement occurs, and associated with each test case that executes that method, class, or program. Using this approach, when variable or type declaration statements change, `SelectTests` selects all test cases associated with the nodes that represent those statements.

This approach has drawbacks: a new, modified, or deleted variable or type declaration may necessitate selection of all test cases associated with a program, class, or method. Many of these test cases may actually

be unaffected by the change, and their selection may be unnecessary.

An alternative approach can obtain more precision by postponing test selection. Instead of selecting test cases through modified declaration nodes, an algorithm can locate and mark as affected all ICFG nodes in G and G' whose associated statements contain references to variables whose declarations have changed or whose declarations are dependent on changed type definitions. The algorithm can then skip declaration nodes in its traversal (or those nodes can be omitted from the ICFGs) and treat nodes that are marked “affected” as if they signal the presence of modified code. This approach incurs the additional cost of establishing a correspondence between nonexecutable statements in G and G' and determining where changes in those statements have taken place; however, the approach can reduce the size of the test set the algorithm selects.

5.5 Distinguishing driver, setup, and oracle code from code under test

When testing classes, it is important to distinguish code being tested from code used to drive, setup, or check results of the test. To see why, consider what happens when a method M necessary to initialize an object’s state, but not considered a subject of the test, is modified. If `SelectTests` treats all methods, including setup methods, equivalently, then `SelectTests` will select every test case through M . However, there may not be time, and it may make no sense, to run all such test cases. In this case testers may choose to first test M independently, and then select test cases for the rest of the class independent of M . Similar cases arise with respect to code used to drive tests, and code used to check state and results following a test run.

To prohibit `SelectTests` from considering the effects of changes in particular methods, a tester must “inform” the algorithm about the identity of those methods, either directly, or by leaving prohibited methods uninstrumented. The algorithm can then leave unexpanded, in ICFGs or CCFGs, call nodes to those methods.

5.6 Specification and code based testing

When programs are tested initially, specification based (“black box”) testing is often employed, because only such testing can identify errors of missing functionality [24]. If testers rely solely on black box testing, however, they may miss significant faults, because they may fail to test some components of the code. Thus, when testers test programs initially they should also employ “white box” techniques that analyze code [39]. The same conclusions may be drawn about regression testing: testers should employ both black and white box techniques when selecting regression tests. If the specifications for modified software have changed, and the code modifications necessary to implement the changed specifications have not been made, such a fault can only be detected by black box test case selection, wherein test cases related to the changed specification are selected. However, if testers rerun *only* the test cases identified as related to the changed specifications, they may omit test cases that exercise portions of the code that have (inadvertently) been affected by code modifications.

The regression test selection technique presented in this paper is strictly code based, and thus fulfills the need to employ a white box test case selection technique. Testers can apply it to a test set T that includes *both* white box and black box test cases, and thereby select all test cases in T that exercise modified code. To achieve adequate confidence in modified software, however, this technique should be used in conjunction with

a black box test selection technique that selects test cases relevant to changed specifications.

6 Empirical Studies

To empirically investigate the regression test selection approach presented in this paper, a commercial C++ class library was obtained, with its actual versions and test suites, and the approach was applied to these artifacts.

6.1 Subjects

The study utilized the code for six successive versions (6.00, 6.01, 6.02, 6.03, 6.04, and 6.10) of the `Tools.h++` C++ class library. The `Tools.h++` library, developed by Rogue Wave Software, Inc., consists of a large set of concrete classes that can be used in isolation and do not depend on other classes for their implementation or semantics [40]. It also includes a set of abstract classes that define an interface for various objectives and a set of implementation classes that implement that interface. The most recent version of the library (among those utilized in the study) contains 186 classes and 24,849 lines of code.

The study utilized test cases that had been created by the developer to test and regression test these versions of the `Tools.h++` library. These test cases are implemented as 61 C++ driver programs, each exercising functionality in one or more classes. A test script runs these drivers one after the other, and compares the output of each driver to expected outputs. Figure 14 shows what a portion of such a driver might look like; the code in the figure includes test cases that exercise two different constructors of a class `Date`. Following each constructor invocation, the driver contains several “asserts”; these print error messages if methods do not return expected values.

Given such a set of test drivers, each test driver can be considered to be a single test case, or each sequence of statements in a driver that could be placed independently into a separate driver can be considered to be an individual test case. These two views provide two different levels of test case granularity, here called *coarse* and *fine*, respectively. For example, the driver of Figure 14 can be treated as providing just one test case (coarse granularity), or two test cases (fine granularity). In practice, regression test selection at fine granularity could be supported by rewriting test drivers at the finer granularity. Alternatively, it could be achieved with the given drivers by using a code instrumenter that can distinguish the trace for each sequence of statements that is to be treated as an independent test case.

Under these two views, the test drivers provided with `Tools.h++` provide 61 test cases (coarse granularity), or between 281 and 317 test cases depending on the version (fine granularity).

6.2 Empirical Procedure

A full implementation of the test selection technique presented in this paper requires a front end language analyzer that can instrument programs and output control flow graphs and symbol table information. Such an analyzer for C++ was not available, and thus it was not possible to create the graphs or test histories that would support an implementation of `SelectTests` on `Tools.h++`. However, by using a simulation technique

```

// first constructor test
cout << "Checking constructor Date(unsigned day, unsigned year)...";
Date d2(d, y);
assert(d2.isValid() == 1);
assert(d2.day() == 36);
assert(d2.dayOfMonth() == 5);
assert(d2.firstDayOfMonth() == 32);
cout << "Done" << endl;

// second Constructor test
cout << "Checking constructor Date(unsigned, unsigned, unsigned)...";
Date d1(4, 12, 1984);
assert(d1.isValid() == 1);
assert(d1.day() == 339);
assert(d1.dayOfMonth() == 4);
assert(d1.firstDayOfMonth() == 336);
assert(d1.leap() == 1);
cout << "Done" << endl;

```

Figure 14: Sample test driver.

it was possible to determine the test selection results that would result if an implementation of the algorithm was applied to `Tools.h++`.

To do this, the Unix `diff` utility was used to locate differences between each version of `Tools.h++` and its successor version, `Tools.h++'`. At each executable code location in the successor version where the two versions differed, code was inserted that opens a file `testresults`, writes “SELECTED” into the file overwriting text previously present, and then closes the file. In cases where class data members or variable declarations differed, the locations were found in the executable code where references to those variables occurred, and instrumented as if they contained modified code. `Tools.h++'` was then compiled and linked to create a library file.

In each test driver, code was inserted immediately before the first statement of each fine granularity test case to open the file `testresults` and write “NOT SELECTED” to it, overwriting any text previously present. Immediately following the last statement of each fine granularity test case, code was inserted to append the contents of the `testresults` file to the file `result.drivername`. These test drivers were then linked with the instrumented libraries. Each run of a test driver thus instrumented produces a file `result.drivername`, in which the k th line corresponds to selection results for the k th fine granularity test case in that driver.

Given the foregoing instrumentation, when the test drivers are run, fine granularity test i in driver d is modification traversing and would be selected by `SelectTests` if and only if the `result.d` file contains the phrase “SELECTED” as its i th entry. Furthermore, the coarse grained test case corresponding to d would be selected by `SelectTests` if and only if the `result.d` file contains one or more lines whose text is “SELECTED”. This simulation requires execution of all test drivers in order to determine which test cases `SelectTests` would select, and thus is of interest only as a simulation. However, this simulation makes it possible to measure test selection results on `Tools.h++` in the absence of the infrastructure necessary to run `SelectTests`.

This procedure was performed on each pair of successive versions of `Tools.h++`; test selection results were collected for each such pair.

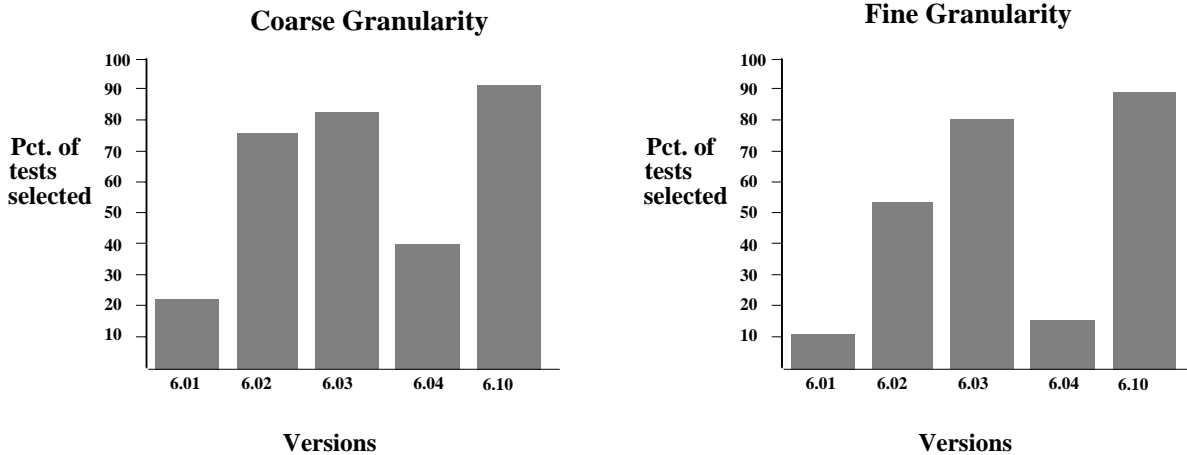


Figure 15: Test selection results for `Tools.h++`.

6.3 Results

Figure 15 shows test selection results. The graph on the left shows results at the coarse granularity level, and the graph on the right shows results at the fine granularity level. On average, at the coarse granularity level, 62.5% of the test cases were selected. Test selection results varied, however, from a minimum of 21.3% to a maximum of 91.8%. At the fine granularity level, an average of 47.2% of the test cases were selected. In this case, the results varied from a minimum of 10.6% to a maximum of 82.4%.

These results indicate that `SelectTests` can reduce the number of regression tests that must be run – at times substantially. Of course, these results were gathered only for one particular program with its versions and test suites; determining the extent to which these results generalize will require further studies.

A reduction in the number of test cases that must be run does not, by itself, promise savings; savings will result only if the cost of test selection analysis plus the cost of running the selected test cases is less than the cost of simply running all test cases. If test execution and validation are fully automated and the entire test suite can be executed within the allotted testing time then test selection may be unnecessary. If execution and/or validation are excessively expensive in time or human interaction, and if test selection analysis is sufficiently inexpensive, test selection may be worthwhile. Studies of test selection for procedural programs [48] indicate that test selection analysis can be sufficiently inexpensive, even on large programs, to be practical. Determining whether those results extend to C++ software will require further studies.

Another important implication of the results of this study concerns the effects of test granularity on test selection. It seems intuitively obvious that fine granularity test cases should promote greater savings in test selection than coarse granularity test cases. These results support this intuition, and in doing so, they have ramifications for test design. Other factors, such as the ease of tracing failures to faults, also favor fine granularity test cases. Of course, factors such as the cost of initializing system state prior to running a test, and the desire to exercise interactions between various functionalities, favor the design of coarse granularity test cases. The possible tradeoffs in coarse versus fine grained test case design should be further examined.

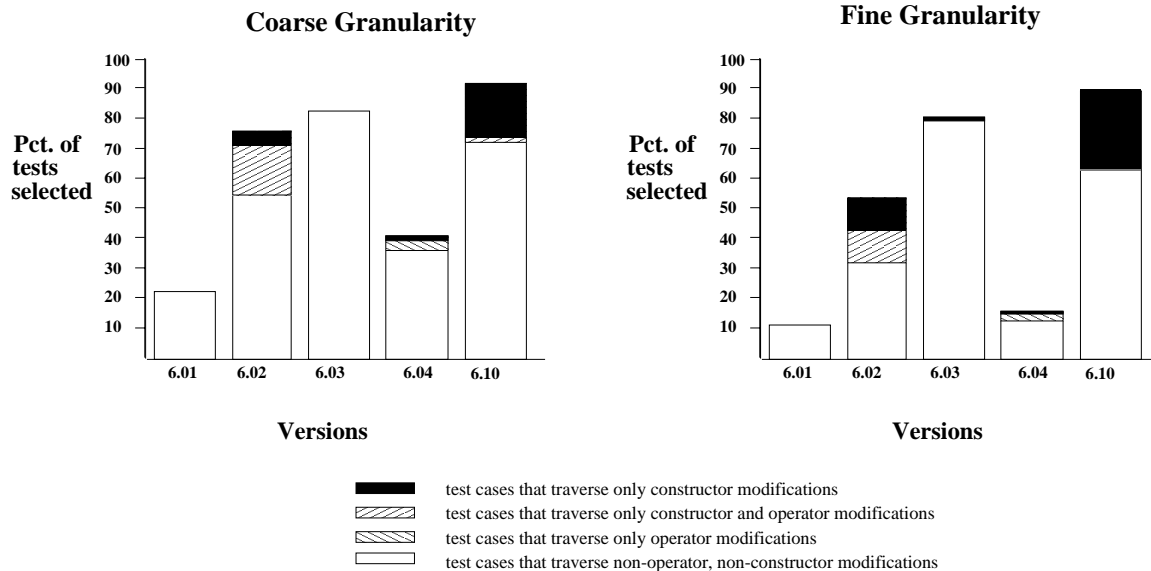


Figure 16: Contribution of constructor and operator modifications to test selection results for `Tools.h++`.

6.4 Followup Study

To gain further insight into the use of `SelectTests`, the modifications in source code that led to selection of test cases were reviewed. They were categorized into (a) modifications in constructor methods, (b) modifications in operator methods⁹ and (c) other modifications. It was questioned whether modifications in constructor and operator methods had caused selection of test cases that exercised no changed functionality outside of those methods. If so, a cost effective selective retest strategy could be to first independently test constructor and operator methods, and then, in test selection, ignore changes in those methods (an effect that can be achieved by summarizing such methods, in graphs, as single, unchanged nodes.)

To perform this investigation, changes in the `Tools.h++` versions were examined, and conditional compilation directives were placed around the code that outputs “SELECTED” to provide a way to turn on, or off, consideration of modifications of types a, b, and c. The instrumented versions were then rerun three times, once with each type of modification toggled on and the other two types toggled off, and test selection data was collected.

Figure 16 shows the results of this followup study. Constructor and operator modifications contributed to test selection in three cases at coarse granularity, and in four cases at fine granularity. However, in two of these cases (6.03 and 6.04), constructor and operator modifications together accounted for fewer than 5% of the test cases selected. On versions 6.02 and 6.10, however, constructor and operator modifications accounted for between 22% and 35% of the test cases selected. In these cases, depending on the various cost factors involved in the testing, it could be advisable to ignore the effects of changes in constructor and operator methods on test selection, and test those methods independently.

⁹Type (b) corresponds to operator overloading in C++. The overloaded operator has new functionality defined in a method.

7 Related Work

Many researchers have considered the problem of regression test selection for procedural software; Section 2 cited relevant papers. Here, previous work on regression testing of object-oriented software is discussed.

Fielder [13], and Cheatham and Mellinger [7], present techniques for reducing the effort required to test subclasses. Both of these techniques, however, require analysis that must be performed by hand, and neither attempts to reuse test cases associated with the parent class.

Harrold, McGregor, and Fitzpatrick [17] present a technique for reusing testing information associated with a parent class in the design of test suites for derived classes; as such, this technique addresses one of the subproblems addressed by our methodology. However, Harrold et. al’s technique does not address problems in regression test selection for modified classes or derived classes, or problems in test selection for modified object-oriented application software or software that use modified classes. Furthermore, the technique reuses test cases based on their association with modified methods and attributes, and thus, is less precise than `SelectTests`, selecting test cases that do not exercise changed code or code actually affected by modified declarations.

Reference [44] presented previous algorithms by Rothermel and Harrold for selecting regression tests for C++ software based on walks of program dependence graphs [12]. The algorithms presented there, like the algorithms here, apply to C++ application programs, classes, and derived classes. Construction of program dependence graphs, however, is more expensive than construction of control flow graphs ($O(n^2)$ runtime versus $O(n)$ for programs of n statements). Thus, the algorithms presented here are more efficient than those of Reference [44].

In References [26, 27] and [25], Kung et al. and Hsia et al. present a technique for selecting regression tests for class testing. This technique is based on the concept of *firewalls* defined originally by Leung and White for procedural software [31, 32, 58] and later extended to object-oriented software [57]. The technique of Kung, Hsia et al. (here called the “ORD technique”) constructs an *object relation diagram* (ORD) that describes static relationships among classes. The represented relationships include inheritance, aggregation (the use of composite objects), and association (the existence of data dependence, control dependence, or message passing relationships between classes). The ORD technique instruments code to report the classes that are exercised by test cases. The firewall for a class C is defined as the set of classes that are directly or transitively dependent on C (by virtue of inheritance, aggregation, or association) as described by an ORD. When class C is modified, the ORD technique selects all test cases that were determined through instrumentation to exercise one or more classes within the firewall for C .

The ORD technique and the technique presented in this paper are similar in that they both select all test cases associated with some set of code components, and the association of test cases with code components is determined dynamically through instrumentation. The primary difference between the techniques is the granularity at which they consider components. The ORD technique selects all test cases associated with classes within the firewall; it performs no further analysis within classes and methods to attach test cases to entities at a finer granularity. Not all of those test cases necessarily execute changed code, or code that accesses changed data objects. Similarly, a class D may be determined by the ORD technique to be dependent

by message passing on some class C , and if C is modified, all test cases associated with D will be selected. Not all of these test cases necessarily exercised code involving any interaction with C . In cases such as these, the ORD technique selects test cases that could be omitted from retesting with no ill effects – test cases that `SelectTests` does not select. Thus, `SelectTests` is more precise than the ORD technique.

Precision, however, is not the only issue to consider when choosing a test selection technique [46]; the ORD technique may require less analysis than `SelectTests`, and if so, then in cases where its results are sufficiently precise, it may be more cost effective than `SelectTests`. It is not clear whether the analysis required by the ORD technique is less than that required by `SelectTests`: this would depend on the level of precision incorporated into algorithms used by the ORD technique to calculate data and control dependencies utilized to determine association relationships. Imprecise analysis is less expensive, but the precision sacrificed may lead to selection of inordinate numbers of test cases. Ultimately, precision/efficiency tradeoffs such as this must be investigated empirically.¹⁰

Although Reference [25] does not discuss it, the ORD technique, like the technique presented in this paper, must also determine sets of methods that may be invoked from particular call sites, in order to calculate association via message passing.

8 Conclusions and Future Work

This paper has presented a new method, based on code analysis techniques, for selecting regression tests for C++ software. The technique selects test cases from existing test suites that execute code that has changed in the production of a C++ application program or class. The approach also selects test cases that should be run for C++ classes derived from other classes. The approach is advantageous because it handles selective retest needs for C++, and is independent of program specifications and methods used to develop test suites initially.

Although the current work has addressed test selection for C++ software, it is believed that the approach can be adapted to other strongly-typed languages such as Java. Future work will consider the possibility of such adaptation.

Also of importance for future work is obtaining empirical data on the effects of polymorphism on graph size and algorithm runtime. As described earlier, existing empirical results suggest that for the strongly typed languages that are being targeted, these effects will be negligible, because it will typically be possible to determine a small set of methods that may be invoked from a given call site. When the set of statically determinable called methods is not small, by relying on dynamic information in test histories, it is possible to further restrict graph size. Whether such an approach would suffice for languages that are not strongly typed must still be investigated. The effects of varying the approaches to handling changes in nonexecutable statements must also be empirically investigated.

Additional future work involves considering other important regression testing problems. This paper has considered only the problem of selecting test cases, from an existing test suite, for re-execution. In general, modifications to classes may render existing test suites inadequate: new test cases to exercise new functionality

¹⁰In empirical studies of two regression test selection techniques for procedural programs that differ only in cost and efficiency, just such a tradeoff has been observed [41]; the challenge it raises is one of predicting which approach will be more cost effective in a given situation.

are needed, and the problem of identifying where such test cases are needed is at least as important for software quality as the regression test selection problem.

Acknowledgements

This work was supported in part by gifts from Microsoft and from Rogue Wave Software, Inc., and by NSF under Faculty Early CAREER Award CCR-9703108 to Oregon State University, NYI award CCR-9696157 to Ohio State University, and Experimental Software Systems Award CCR-9707792 to Ohio State University and Oregon State University. Randall Robinson and Rogue Wave Software supplied the artifacts used in the empirical studies.

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, Boston, MA, 1986.
- [3] T. Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA)*, pages 134–142, March 1998.
- [4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50. IEEE Computer Society Press, November 1992.
- [5] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, August 1997.
- [6] 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.
- [7] T.J. Cheatham and L. Mellinger. Testing object-oriented software systems. In *Proceedings of the 1990 Computer Science Conference*, pages 161–165, 1990.
- [8] 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 211–220. IEEE Computer Society Press, May 1994.
- [9] T. Dogsa and I. Rozman. CAMOTE - computer aided module testing and design environment. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 404–408. IEEE Computer Society Press, October 1988.
- [10] Roong-Ko Doong and P.G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, October 1991.

- [11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [12] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–49, July 1987.
- [13] S.P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, April 1989.
- [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, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, May 1992.
- [18] M.J. Harrold and G. Rothermel. Performing dataflow testing on classes. In *Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 154–63, December 1994.
- [19] M.J. Harrold and G. Rothermel. A coherent family of analyzable graph representations for object-oriented software. Technical Report OSU-CISRC-11/96-TR60, The Ohio State University, November 1996.
- [20] 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.
- [21] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.
- [22] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the Third Workshop on Software Testing, Analysis, and Verification*, pages 97–102. ACM Press, December 1989.
- [23] D. Hoffman and P. Strooper. Graph-based module testing. In *Proceedings of the 16th Australian Computer Science Conference*, pages 479 – 487, February 1993.
- [24] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, 1976.
- [25] P. Hsia, X. Li, D. Kung, C-T. Hsu, L Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice*, 9:217–233, 1997.

- [26] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y-S. Kim, and Y-K. Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, October 1995.
- [27] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–40, January 1996.
- [28] W.A. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.
- [29] L. Larsen and M.J. Harrold. Slicing object-oriented software. In *18th International Conference on Software Engineering*, pages 495–505, March 1996.
- [30] H.K.N. Leung and L. White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 60–69. IEEE Computer Society Press, October 1989.
- [31] H.K.N. Leung and L. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance: Research and Practice*, 2:209–222, December 1990.
- [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 201–8, October 1991.
- [34] R. Lewis, D.W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *Proceedings of the 2nd European Software Engineering Conference Proceedings (ESEC '89)*, pages 487–496. Springer-Verlag, September 1989.
- [35] D. Liang and M.J. Harrold. Slicing objects using system dependence graphs. In *Proceedings of the 1998 International Conference on Software Maintenance*, pages 358–367, November 1998.
- [36] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–47, September 1988.
- [37] H. Pande. *Compile Time Analysis of C and C++ Systems*. Ph.D. dissertation, Rutgers University, New Brunswick, NJ, May 1996.
- [38] H. Pande and B.G. Ryder. Static type determination in C++. Technical Report LCSR-TR-250, Rutgers University, July 1995.
- [39] D.E. Perry and G.E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January 1990.

- [40] Rogue Wave Software, Corvallis, Oregon. *Tools.h++ Introduction and Reference Manual, Version 6*, January 1996.
- [41] D. Rosenblum and G. Rothermel. An empirical comparison of regression test selection techniques. In *Proceedings of the International Workshop for Empirical Studies of Software Maintenance*, pages 89–94, October 1997.
- [42] G. Rothermel. Efficient, effective regression testing using safe test selection techniques. Technical Report 96-101, Clemson University, January 1996.
- [43] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 358–67, September 1993.
- [44] G. Rothermel and M.J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 14–25. IEEE Computer Society Press, September 1994.
- [45] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 169–184, August 1994.
- [46] G. Rothermel and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, August 1996.
- [47] G. Rothermel and M.J. Harrold. A safe, efficient regression test selection algorithm. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.
- [48] G. Rothermel and M.J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.
- [49] B. Sherlund and B. Korel. Logical modification oriented software testing. In *Proceedings: Twelfth International Conference on Testing Computer Software*. Frontier Technologies, June 1995.
- [50] M.D. Smith and D.J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, June 1992.
- [51] A.B. Taha, S.M. Thebaut, and S.S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 527–34, September 1989.
- [52] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing. In *Proceedings of the 19th International Conference on Software Engineering*, pages 433–442, May 1997.
- [53] C.D. Turner and D.J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the Conference on Software Maintenance, 1993*, pages 302–11, September 1993.

- [54] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on textual differencing. In *ENCRESS '97, Third International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.
- [55] A. von Mayrhauser, R.T. Mraz, and J. Walls. Domain based regression testing. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 26–35. IEEE Computer Society Press, September 1994.
- [56] E.J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–38, December 1986.
- [57] L. White and K. Abdullah. A firewall approach for the regression testing of object-oriented software. In *Conference Proceedings: Quality Week 1997*, May 1997.
- [58] 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 262–70, November 1992.
- [59] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference on Software Engineering*, pages 41–50. IEEE Computer Society Press, April 1995.
- [60] S.S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Proceedings of the COMPSAC '87: The Eleventh Annual International Computer Software and Applications Conference*, pages 272–77, October 1987.
- [61] J. Ziegler, J.M. Grasso, and L.G. Burgermeister. An Ada based real-time closed-loop integration and regression test tool. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 81–90. IEEE Computer Society Press, October 1989.