Selecting Regression Tests for Object-Oriented Software*

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

Abstract

Regression testing is an important but expensive software maintenance activity aimed at providing confidence in modified software. Selective retest methods reduce the cost of regression testing by selecting tests for a modified program from a previously existing test suite. Many researchers have addressed the selective retest problem for procedural-language software, but few have addressed the problem for object-oriented software. In this paper, we present a new technique for selective retest, that handles object-oriented software. Our algorithm constructs dependence graphs for classes and applications programs, and uses these graphs to determine which tests in an existing test suite can cause a modified class or program to produce different output than the original. Unlike previous selective retest techniques, our method applies to modified and derived classes, as well as to applications programs that use modified classes. Our technique is strictly code-based, and makes no assumptions about methods used to specify or test the software initially.

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 software maintenance; without proper regression testing we are reluctant to release modified software. One characteristic distinguishing regression testing from developmental testing is the availability, at regression test time, of existing test suites. If we reuse such test suites to retest a modified program, we can reduce the effort required to perform that testing. Unfortunately, test suites can be large, and we may not have time to rerun all tests in such suites. Thus, we must often restrict our efforts to a subset of the previously existing tests. We call the problem of choosing an appropriate subset of an existing test suite the selective retest problem; we call a method for solving this problem a selective retest method.

Although many researchers have addressed the selective retest problem for procedural-language software[2, 3, 5, 9, 11, 15, 16, 18, 20, 24, 26, 29, 30], we are aware of only one technique that addresses the problem with respect to object-oriented software[7], and that approach applies only to test selection for derived classes. 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 selective retest methods. When a class is modified, the modifications impact every applications program that uses the class and every class derived from the class; ideally, we should retest every such program and derived class[25, 28]. The object-oriented paradigm also alters the focus of test selection algorithms, emphasizing and creating different concerns. For example, since most classes consist of small interacting methods, selective retest approaches for object-oriented programs must work at the interprocedural level. Also, since many methods for testing object-oriented software treat classes as testable entities, and design or employ suites of class tests for classes[6, 7, 12, 25, 27], selective retest methods must support the use of class tests.

In this paper, we present a new selective retest method that addresses the selective retest problem for object-oriented software. Our method constructs dependence graphs for classes and programs that use classes; we use these graphs to select all tests in a test suite that may cause a modified class, derived class, or applications program that uses a class to produce different output than the original program or class.

Our approach has several benefits. First, our method is currently the only selective retest method applicable to test selection for applications programs, classes, and derived classes. Second, our method selects tests using information gathered by code analysis, and does not require the specifications on which the code is based. Third, our approach is independent of the method used to generate tests initially for programs and classes. Fourth, unlike most selective retest methods, our method selects every test that may produce different output in the modified program. Fifth, unlike many selective retest algorithms, our approach handles both structural and nonstructural program modifications, and processes multiple modifications with a single application of the algorithm. Sixth, where most selective retest methods function at the unit test level, our approach works interprocedurally – a necessity where test selection for classes is concerned. Finally, our method is automatable.

*This work was partially supported by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University.
In the next section, we provide background information required for the rest of the paper. Section 3 addresses object-oriented program testing issues that are relevant to the paper. Section 4 describes our algorithm for selecting regression tests for modified object-oriented applications programs; our figures and discussion employ C++, but our approach applies, with adaptations, to other object-oriented languages. Section 5 presents our algorithm for selecting regression tests for modified or derived classes. Section 6 discusses some additional issues relevant to this work, and section 7 presents conclusions.

2 Background

In this section, we overview the program dependence graph used by our algorithms. Then, we briefly discuss the regression testing problem. Finally, we discuss the test history that our algorithms require.

Program Dependence Graphs

A program dependence graph (PDG) represents both control dependence and data dependence in a single graph. For statements X and Y in a program, if X is control dependent on Y then there must be at least two paths out of Y, where one path always causes X to be executed and the other path may result in X not being executed[4]. A data dependence exists between statements X and Y in a program if X defines a variable v, Y uses v, and there is a path from X to Y in the program on which v is not defined[1, 14].

Figure 1 shows procedure search and its partial PDG; node labels in the PDG correspond to statement numbers in the procedure. The search procedure is part of the List class shown in full in Figure 6. To facilitate presentation, we present the figure here using the same statement and node numbers used in Figure 6. For our discussion, we assume that a program has a single entry and a single exit. This implies that all procedures return to their callers; thus, we treat call sites as simple statements. Although we do not discuss it in this paper, with a simple modification in the construction of a PDG, we accommodate the possibility that procedures may exit before returning to their callers.

A PDG contains several types of nodes. Statement nodes, shown as ellipses in Figure 1, represent simple program statements such as assignment, input/output, variable declaration, or call statements. Circles represent region nodes, which summarize the control dependence conditions necessary to reach statements in the region. Predicate nodes, from which two edges may originate, are represented as squares. A hexagon represents the exit from the procedure. Although we do not represent them as such, we may think of statement and predicate nodes as possessing the text of the program statements to which they correspond as labels. We use a single statement node to summarize the variable declarations present in a procedure.

Control dependencies are represented explicitly in the PDG by control dependence edges, shown as solid lines in Figure 1. For any control dependence edge (A, B) in a PDG, A is a control dependence predecessor (cd-predecessor) of B, and B is a control dependence successor (cd-successor) of A. Nodes in a PDG that have no cd-successors are leaf nodes; nodes with cd-successors are interior nodes. In Figure 1, S35 is a cd-predecessor of R10, and R10 is a cd-predecessor of S35. Furthermore, S35 is a leaf node and R10 is an interior node. Statements that have the same control dependencies are grouped together into the same region in a PDG.

Data dependencies between program statements are represented in a PDG by data dependence edges, shown in Figure 1 as dashed lines. For any data dependence edge (A, B), A is a data dependence predecessor (dd-predecessor) of B, and B is a data dependence successor (dd-successor) of A. In the PDG in Figure 1, there are data dependence edges from S32, where loc is used. Thus, P33, P34, and S36 are dd-successors of S32, and S32 is a dd-predecessor of P33, P34, and S36.
Regression Testing

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

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

In previous work[21], we described a framework for evaluating regression testing techniques. Using this framework, we classify a regression testing technique as safe if it selects all tests from $T$ that could possibly exhibit different output when run on $P'$. However, a safe approach may include many tests in $T'$ that cannot possibly exhibit different output in $P'$. Thus, we further classify a safe regression testing technique as precise if it avoids choosing tests that will not cause $P'$ to produce different output than $P$. Since we cannot find an algorithm to determine, for an arbitrary choice of $T'$, whether a test will exhibit different output in $P'$ than in $P$, no technique can be both safe and precise[21].

When evaluating the efficiency of selective retest methods, it is important to recognize that regression testing is typically accomplished in two phases. In the initial phase of regression testing, while code is being modified, regression testing is not on the critical path; testers may be retesting modified units, developing test plans, and doing limited integration testing, but the bulk of the testing effort awaits inclusion of the final modifications. When modifications are complete, regression testing enters the critical phase, where final integration and system tests must be selected and executed. The time allotted to this critical phase is limited by the need to release the product. Efforts to increase regression testing efficiency yield the greatest payoff when they address activities in this critical phase; expensive activities that are confined to the initial phase are less significant.

Test History

To perform regression testing using an existing test suite, we must keep test history information about previous testing sessions. For our purposes, test history information lists the execution trace for each test in terms of PDG predicates and regions traversed; we call this a region trace. With each predicate and region in the PDG we associate the set of tests in $T$ whose region trace includes that predicate or region. We call this set a region history; for a predicate or region $R$, we use $R.history$ to refer to $R$'s region history. Note that if execution traces of statements instead of regions are provided, we can easily generate region traces from statement traces without rerunning all tests on the original program.

Figure 1 contains a table that describes a set of tests $T$ used initially to test procedure search. The table lists the region trace and the initial values for inputs and global variables for each test. In the PDG in the figure, $R9$ only has tests $T1$ and $T3$ associated with it while $R7$ has all tests in the test suite associated with it. Thus, $R9.history = \{T1,T3\}$ and $R7.history = \{T1,T2,T3,T4\}$.

3 Regression test selection for object-oriented software

Object-oriented languages raise interesting concerns for regression testing. In this section we discuss concerns relevant to selective retest.

Levels of testing. To test object-oriented software properly, we must test classes[19]. Class testing approaches typically invoke sequences of methods in varying orders, and then verify that the resulting state of the objects manipulated by the methods is correct[6, 7, 27].

The object-oriented paradigm provides new applications for selective retest algorithms. When we modify a class, we must retest the class, and classes derived from that class[7, 19]. Moreover, although we expect encapsulation to reduce the likelihood that object-oriented code modules will interact inappropriately, it is still the case that tests run on applications programs may reveal faults in methods that were not revealed by tests of the individual methods[19, 28]. Thus, if we want to be sure that we have rerun all existing tests that may expose errors in a modified class (i.e., select a safe test set) we must consider all applications programs that use the modified class[7, 19, 28].

Whether retesting applications programs, classes, or derived classes, we can benefit by applying selective retest algorithms to existing test suites.

Drivers, setup routines, and oracles. To perform class testing, we require a driver that invokes a sequence of methods. A typical class test driver first performs "setup" chores, calling constructor routines and/or other 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. Code-based selective retest methods must be able to distinguish between drivers, setup routines, oracle routines, and methods actually under test, and select only the tests that are relevant to changes in methods under test.

Polymorphism and dynamic binding. Object-oriented programs employ polymorphism and dynamic binding to a degree beyond that of procedural programs. In an object-oriented program a method invocation can be bound at run-time to a number of methods. For a given call, we cannot always determine statically the method to which it will be bound. Selective retest methods that rely on static analysis must provide mechanisms for coping with this uncertainty.

1 Clearly, in practice it may not be practical to retest all such programs. We discuss this at greater length in section 6.
4 Selecting regression tests for modified applications programs

To select regression tests for modified object-oriented application programs, or programs that use modified classes, we use an interprocedural program dependence graph (IPDG). Given program P, the IPDG for P consists of individual PDGs for procedures in P, with edges added to represent interprocedural data and control dependencies. Figure 2 depicts a program RemDups and its (partial) IPDG. RemDups uses a List object and methods from a List class. (The figure only shows List class methods used in RemDups; the entire List class is presented in Figure 6.) In the figure, the four bold edges are call edges, which summarize the control dependence of code in called procedures on regions enclosing call sites. The figure depicts one interprocedural data dependence edge, from S73 to P17 (the dashed line). In our IPDG, a state node lists all identifier declarations in the program that are not already listed in a declarations node in some PDG. The state node is a child of the entry node for the main procedure. In Figure 2 the state node contains declarations of variables list, numentries and maxentries, and constant MAXLIST, all of which appear in the declarations section of the List class (as shown in Figure 6). Note that a call edge connects S71-72 to E1, because the declaration of variable 1 invokes the List constructor routine to instantiate a List object. For simplicity, the figure omits the code and PDGs for getnextnum, dosomething, and other List methods not invoked by the application, as well as the remaining data dependence edges.

We use algorithm ConstructIPDG to construct an IPDG. The running time of ConstructIPDG is dominated by the cost of interprocedural dataflow analysis, which is at worst polynomial in the size of the program analyzed[10, 14]. Due to space limitations, we do not present ConstructIPDG here; interested readers are referred to [22].

To select tests for regression testing of applications programs, we use algorithm SelectAppTests, shown in Figure 3. SelectAppTests takes a program P, its modified version P', and a set of tests T used to test P, and selects a safe test set T' for P'. The algorithm finds regions in the IPDGs for P and P' that enclose code which, due to changes or the effects of changes, may cause P' to produce different output than P. For each such pair of regions R and R' in the IPDGs, SelectAppTests selects all tests in R.history, because these tests must either reach R', or reach some
**Algorithm**

\[ \text{SelectAppTests}(P, P', T) : T' \]

**Input**

- \( P, P' \): a program and its changed version
- \( T \): a test set used previously to test \( P \)

**Output**

- \( T' \): the subset of \( T \) selected for reuse

**Start** SelectAppTests

1. \( G = \text{ConstructIPDG}(P); \quad G' = \text{ConstructIPDG}(P') \)
2. \( \text{foreach} \ \text{node} \ n \ \text{in} \ G \ \text{and} \ G' \ \text{do mark} \ n \ \text{“not visited”} \)
3. \( T' = \text{Compare}(E, E') \), where \( E, E' \) are entry nodes of \( G, G' \), respectively

**End** SelectAppTests

---

**Figure 3:** Algorithm for selecting tests to rerun on an applications program.

---

**Figure 4:** Procedure search2 (left), and the IPDG for RemDups2, with changes enclosed in dotted rectangles.

---

other pair of regions enclosing code that has been changed. In either case, such tests may produce different output in \( P' \) than in \( P \). For more detailed discussion of the algorithm used by SelectAppTests, see \[22\] and \[23\]. The following examples demonstrate the use of the algorithm.

Suppose the search method is modified, yielding search2 as shown in Figure 4. In search2, line P34 has been (erroneously) modified, and a new line, S33a, has been added. Also, a new state variable, mostrecentlyfound, and a new method, getmostrecentlyfound, are added to the List class. Suppose further that the test set \( T \) for RemDups contains at least tests \( T1, T2, T3, \) and \( T4 \) that act, with respect to procedure search2, like the four tests given in Figure 1. We wish to determine which tests in \( T \) must be reexecuted when RemDups2 is linked with the modified version of the List class. In this case, we call SelectAppTests with RemDups, RemDups2, and \( T \). SelectAppTests first constructs the IPDGs \( G \) and \( G' \) for the two programs. Figure 4 depicts \( G' \) without data dependence edges. \( G' \) differs from \( G \) only with respect to the PDG for search, and with respect to the state node; changed parts are enclosed in dotted rectangles. Since getmostrecentlyfound is not invoked by RemDups2, its PDG is not needed or included in the IPDG for RemDups2.

After marking nodes in the graphs “visited”, SelectAppTests calls Compare, shown in Figure 5, with entry nodes \( E70 \) and \( E70' \). GetCorresp finds that the children of these nodes differ only in the state nodes: state' contains a declaration of the variable mostrecentlyfound that is not present in state. However, mostrecentlyfound has no uses in \( G' \), so Compare marks no nodes “affected” by this change. Compare then invokes itself on nodes \( S71-72 \) and \( S71-72' \), finding no changes beneath these nodes, and then on nodes \( S74 \) and \( S74' \), finding no changes beneath them. Traversal then continues with \( P75 \) and \( P75' \), \( R16 \) and \( R16' \), \( S76 \) and \( S76' \), and down into the PDGs for search and search2; no differences are found until nodes \( R8 \) and \( R8' \) are reached. At this point Compare notes the difference between P34 and P34', and selects all tests in R8.history (\( T2, T3, \) and \( T4 \)). The
algorithm does not continue further beneath R8 and R8'. Instead, it traverses other portions of the graphs. No further changes are discovered; tests T2, T3, and T4 are selected.

In the example just presented, due to the structure of RemDups and the location of the modifications, it is likely that most tests in T are selected: the only tests not reaching R8 are those in which RemDups2 passes an empty or one-element list to search. This test selection is reasonable: all tests reaching R8 may exhibit different output and should be rerun. Suppose, however, that the only changes to search were the insertion of line S35a and the declarations of variable mostrecentlyfound and method getmostrecentlyfound. In this case, Compare again finds that the state and state' nodes of the two graphs differ: the state' node listing the new global variable. Again, this variable has no uses in RemDups or RemDups2 so Compare does not mark any nodes "affected" by the global variable insertion. Compare next traverses G and G' until it reaches nodes R10 and R10', where it finds a difference in cd-successors. However, this difference involves only an assignment to mostrecentlyfound, and since mostrecentlyfound has no uses in RemDups2, Compare does not select the tests attached to R10 or mark any nodes "affected". SelectAppTests recognizes that no tests in T can produce different output in RemDups2 than in RemDups, and returns an empty set of tests.

The running time of SelectAppTests is dominated by the cost of the interprocedural dataflow analysis required to complete construction of the IPDG. The subsequent traversal of the IPDG using Compare is linear in the size of the larger of the IPDGs. Note that construction of the IPDG for P, and collection of test history information, can both be accomplished during the initial phase of regression testing. Thus, only the construction of the IPDG for P and the traversal of the IPDGs must be performed during the critical phase of testing.

5 Selecting regression tests for modified and derived classes

When a class is modified, we want to find tests in the class's test suite that should be reexecuted. Similarly, when we derive a new class from an existing one, we want to identify the tests from the test set for the base class that should be reexecuted on the derived class. The method of Section 4 does not apply directly to class regression testing, because it requires a program to have a single entry point, whereas a class has multiple entry points. However, by using a Class Dependence Graph (CDG) (a variant of the IPDG) we can depict control and data dependencies within a class in a manner that lets us use an algorithm similar to SelectAppTests to select regression tests for modified or derived classes.

To understand our approach, recall from the discussion in section 3 that to test a class, we create a number of driver programs. These driver programs either assume or instantiate some initial state, and then invoke methods in some required sequence. To retest a modified class, a naive approach is to treat each driver as an application program and use the method of the previous section to retest it. However, this approach has two drawbacks. First, it may require us to construct individual PDGs many times. Second,
const int MAXLIST = 10;
class List
{
    int list[MAXLIST];
    int numentries;
    int maxentries;
    public:
    List(int n = MAXLIST)
    {list = new int[n];
        numentries = 0;
        maxentries = n;
    }
    int getnumentries()
    {return numentries;}
    int getmaxentries()
    {return maxentries;}
    int search()
    {return 0;
    }
    void print()
    {
        int i = 0;
        while (i < numentries)
        {
            cout << list[i] << '
';
            ++i;
        }
    }
    int putitem(int item, int loc)
    {
        if (0 <= loc && loc < maxentries)
        {
            list[loc] = item;
            return 0;
        } else
        {
            return -1;
        }
    }
    int getitem(int item, int loc)
    {
        if (0 <= loc < maxentries)
        {
            item = list[loc];
            return 0;
        } else
        {
            return -1;
        }
    }
    ~List()
    {
        delete list;
    }
    int List::putitem(int item, int loc)
    {
        if (0 <= loc && loc < maxentries)
        {
            list[loc] = item;
            return 0;
        } else
        {
            return -1;
        }
    }
    void incnum()
    {numentries = n;}
    void setnum(int n)
    {
        numentries = n;
    }
};

Figure 6: The List class, and its CIDG.

in calling SelectAppTests for pairs of drivers, this
approach may traverse each PDG many times. The
CIDG lets us avoid these problems; it provides a sin-
gle graph in which each PDG is created and traversed
at most once by the test selection algorithm.

Figure 6 depicts the CIDG and code for the List
class used in earlier examples. A "representative
driver node" (RDN) labeled "listRDN" serves as root
of the graph, and summarizes the set of drivers for
class tests. Each public method in the class is made
a child of the root, by adding driver edges (shown as
bold dashed lines). A state node summarizes vari-
bles that make up the state of objects of that class.
Since methods in the List class make no calls on other
methods, the List CIDG contains no nesting of meth-
ods, and hence has no edges from the PDGs for List
methods to other PDGs. CIDGs also contain data de-
pendence edges to depict dataflow dependencies: these
are omitted from the figure.

Figure 7 depicts the code and CIDG for a Stack
class derived from List, again omitting data depen-
dence edges. The Stack CIDG illustrates how PDGs
in a CIDG may be connected both by driver edges to
the class’s RDN, and by call edges to other call sites
within the graph. For example, the putitem method
in the List class is a public method within the Stack
class, but also is called from both the pop and print
Stack methods. Thus, the Stack CIDG contains a
driver edge from stackRDN to E21 (the entry node
for the putitem method), and also contains call edges
from S48 and S60 to E21, representing the calls from
pop and print.
Figure 7: The Stack class derived from List, and its CIDG.
We use algorithm \texttt{ConstructCIDG} to construct a CIDG. The running time of \texttt{ConstructCIDG}, like that of \texttt{ConstructIPDG}, is dominated by the cost of inter-procedural dataflow analysis, which is at worst polynomial in the size of the class analyzed. Due to space limitations, we do not present \texttt{ConstructCIDG} here; interested readers are referred to [22], where \texttt{ConstructCIDG} is presented, and to [8], where we present an algorithm for performing interprocedural dataflow analysis on classes.

To select tests to rerun on a modified class, we use algorithm \texttt{SelectClassTests}, shown in Figure 8, which is similar to algorithm \texttt{SelectApplTests}. \texttt{SelectClassTests} takes as input a class $C$ and its modified version $C'$, the lists of public methods $PubM$ and $PubM'$ in $C$ and $C'$, respectively, and the set of tests $T$ used to test $C$. We assume that $T$ contains all tests used to test $C$, by all drivers used for that purpose, but that each test is uniquely distinguished from tests that employ other drivers. \texttt{SelectClassTests} is discussed in more detail in [22]; here we demonstrate the algorithm's use through examples.

To see how \texttt{SelectClassTests} works when a class is modified, suppose the search procedure is modified, yielding the \texttt{search2} procedure depicted in Figure 4. Assume that a new private variable mostrecentlyfound is declared, and a new method, getmostrecentlyfound, is implemented. When \texttt{SelectClassTests} is invoked, it creates CIDG $G$ shown in Figure 6, and CIDG $G'$ shown in Figure 9. Figure 9 also shows the new method and new declaration, and uses dotted lines to highlight portions of $G'$ that differ from $G$. Next \texttt{SelectClassTests} marks all nodes "not visited", and then marks the use of mostrecentlyfound in $S39$ "affected". The only new child of stackRDN is the entry node for getmostrecentlyfound, and $T$ contains no tests for it, so none are selected at that node. All other PDGs in $G$ correspond to PDGs in $G'$, so \texttt{SelectClassTests} runs \texttt{Compare} on them. However, the only changes in any of these PDGs lie in the PDGs for \texttt{search} and \texttt{search2}. \texttt{SelectClassTests} finds that nodes P34 and P34' differ, and selects all tests in P34.history. These are the only tests selected for this example.

Now consider the case where a class is inherited. Figure 7 shows the code and CIDG for a Stack class derived from the List class. When \texttt{SelectClassTests} is invoked with the List and Stack classes, it creates CIDGs $G$ and $G'$ shown in Figures 6 and 7. The Stack class contains new constructor, destructor, pop and push methods, and redefines the print method. \texttt{SelectClassTests} does not traverse the PDGs for the new methods, because they have no corresponding PDGs in the List class, and thus no tests of that class executed them. \texttt{SelectClassTests} does call \texttt{Compare} on the entry nodes E26 and E56 of the print method, and on discovering that the CD-successors of these nodes differ, selects every test that executed the print method in the List class. Other methods, such as the search method, remain unchanged. \texttt{SelectClassTests} runs \texttt{Compare} on the old and new versions of their PDGs and finds that no tests through these need to be selected. Of course, any tests that called both the search and print functions were selected when \texttt{SelectClassTests} considered the print methods.

As a final example, suppose we have developed and tested our Stack class and created a test set $T$ for it, and suppose we then modify the search procedure in the List class as above, creating \texttt{search2}. To see which tests in $T$ we need to rerun, we use the \texttt{SelectClassTests} algorithm on the Stack class as linked with the first version of List, and the Stack class as linked with the second version of List. The algorithm constructs the CIDGs for the two versions of the Stack class; the CIDG for the new version includes the modified version of search, the new state variable, and the new getmostrecentlyfound method. The algorithm then proceeds precisely as it did when we used it to select tests to retest List.
The running times for SelectClassTests and SelectApplTests depend on the same factors. Much of the processing required for SelectClassTests can be accomplished during the initial phase of testing.

6 Other issues

Several issues raised in the previous sections bear further scrutiny.

Polymorphism and dynamic binding. As detailed in section 3, the polymorphism inherent in object-oriented languages implies that method calls can invoke any of a set of methods. For each call, we can determine a set of methods it can invoke; the size of the set will depend on the precision of our algorithm for determining it, and on how strongly typed our language is. When the set is too large, further graph construction and traversal through that call is impractical. In this case, we summarize a set of callable methods, and summarize the set of variable uses in these methods, as a single node. When our test selection algorithms reach a summary node, if the node contains affected uses or summarizes a method that has been modified, we select all tests through that node.

Drivers, setup, and oracles. With class tests, 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 constructor is modified. If SelectClassTests treats all methods, including setup methods, equivalently, then SelectClassTests will select every test that uses the constructor. Clearly, every such test may exhibit different output due to the modifications to the constructor. However, we may not have time to run all such tests. In this case we may choose to test the constructor independently, sign off on it, and select tests for the rest of the class independent of the constructor. An easy way to prohibit SelectClassTests from considering the effects of changes in particular methods is to leave those methods uninstrumented when we calculate test histories. Since SelectClassTests only selects tests attached to regions enclosing changed code, or attached to both a region enclosing a changed definition of a variable and another enclosing a use of that variable, SelectClassTests will select tests only in routines where history information is gathered.

Specification- and code-based testing. When we test programs initially, we must employ some specification-based ("black-box") testing, because only such testing can identify errors of missing functionality[13]. However, if we rely solely on black-box testing we may miss significant faults, since we may fail to test some components of the code. Thus, when we test programs initially we must also employ "white-box" methods that analyze code[19]. The same conclusions may be drawn about regression testing: we must employ both black and white-box methods 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 selection, wherein tests related to the changed specification are selected. However, if we rerun only the tests identified as related to the changed specifications, we may omit tests that exercise portions of the code that have (inadvertently)
been affected by code modifications.

Our selective retest method is strictly code-based, and thus fulfills the need to employ a white-box test selection method. We apply it to a test set \( T \) that includes both white-box and black-box tests, and thereby select all tests in \( T \) that may produce different output in the modified program due to code modifications. However, to achieve adequate confidence in modified software, our method should be used in conjunction with a black-box test selection method that selects tests relevant to changed specifications.

**The feasibility of safe selective retest for applications programs.** No matter how well we test a class \( C \) we cannot be certain, in general, that \( C \) behaves correctly for all inputs, because we cannot test \( C \) with all inputs. When we test an applications program \( P \) that uses \( C \), our tests of \( P \) may exercise \( C \) with inputs not used during class testing. When we modify \( C \), creating \( C' \), if we do not retest \( C' \) in the context of \( P \), we may fail to detect faults that would be exposed by those inputs. Thus, we cannot confidently state that we have run all existing tests that might expose errors in \( C' \) unless we retest \( C' \) in the context of each applications program that uses it. Clearly, in practice we may have neither the time to retest all applications programs that use \( C' \), nor the ability to identify all such programs. In this case we must settle for less than safe test selection. We may choose instead to retest an identifiable subset of the applications programs that use the class. The algorithm of section 4 may be used to select tests for (and reduce the cost of retesting) this subset of programs.

**Encapsulation.** Object-oriented programming relies on encapsulation to reduce the likelihood that code modules will interact inappropriately. As the preceding paragraph shows, encapsulation does not eliminate the possibility that a test of an applications program will expose a fault in a class that was not exposed by some test of the class. Nevertheless, we expect that the use of encapsulation will reduce the size of the test sets our algorithm selects for applications programs that use modified classes, by eliminating data dependencies that would, if present, force selection of tests through the modified class. Since our algorithms for test selection use data dependence information to detect both tests that must be rerun, and tests that do not need to be rerun, it selects smaller test sets than algorithms that do not use such information\cite{21}.

**7 Conclusion**

We have presented a new method, based on code analysis techniques, for selecting regression tests for object-oriented software. Our method selects tests from existing test suites that may cause a modified program or class to produce different output than the original program or class. Our approach also selects tests that should be run for classes derived from other classes. The approach is advantageous because it handles selective retest needs in the object-oriented paradigm, is independent of program specifications and methods used to develop test suites initially, and selects safe test suites.

Selective retest strategies are cost-effective only if the cost of running the tests these strategies let us omit exceeds the cost of applying the strategies\cite{17}. As the size of an applications program increases, the cost of applying a selective retest strategy to that program may become prohibitive. However, we believe that in practice, the analysis our algorithms perform to select tests for classes will not be prohibitive. Clearly, experimentation is required to investigate the practicality of our test selection algorithms, and to determine under what conditions these algorithms are cost-effective. We are thus currently implementing our algorithms to select tests for C++ classes and applications programs. When the implementation is complete we will perform empirical studies to measure our methods' efficiency and effectiveness.

Several additional areas are open for future work. First, as Hoffman and Strooper\cite{22} observe, careful test selection does not lower the cost of executing the tests we select; it would be useful to consider combining our approach with an approach like theirs that reduces the cost of test execution. Second, we believe our algorithm, with minor modifications, can be used to help detect obsolete tests and determine not just which tests must be reexecuted, but which tests may be inherited by a derived class. Third, we will consider the problems presented by polymorphism and dynamic binding in greater detail, in order to select more precise test sets. Finally, we will investigate methods for reducing the costs of our algorithm associated with graph construction, by constructing portions of graphs on demand rather than constructing entire graphs up front, and by saving and reusing portions of graphs.

**References**


