Proc. of the ACM Conf. on OO Programming, Systems, Languages, and Applications (OOPSLA'01), ACM Copyright.

Regression Test Selection for Java Software[*]

Mary Jean Harrold
harrold@cc.gatech.edu

James A. Jones
jjones@cc.gatech.edu

Tongyu Li
tongyu@cc.gatech.edu

Donglin Liang
dliang@cc.gatech.edu

Alessandro Orso
orso@cc.gatech.edu

Maikel Pennings
pennings@cc.gatech.edu

Saurabh Sinha
sinha@cc.gatech.edu

S. Alexander Spoon
lex@cc.gatech.edu

Ashish Gujarathi
ashish.gujarathi@citrix.com

ABSTRACT

Regression testing is applied to modified software to provide confidence that the changed parts behave as intended and that the unchanged parts have not been adversely affected by the modifications. To reduce the cost of regression testing, test cases are selected from the test suite that was used to test the original version of the software—this process is called regression test selection. A *safe* regression-test-selection algorithm selects every test case in the test suite that may reveal a fault in the modified software. Safe regression-test-selection techniques can help to reduce the time required to perform regression testing because they select only a portion of the test suite for use in the testing but guarantee that the faults revealed by this subset will be the same as those revealed by running the entire test suite. This paper presents the first safe regression-test-selection technique that, based on the use of a suitable representation, handles the features of the Java language. Unlike other safe regression test selection techniques, the presented technique also handles incomplete programs. The technique can thus be safely applied in the (very common) case of Java software that uses external libraries or components; the analysis of the external code is not required for the technique to select test cases for such software. The paper also describes Retest, a regression-test-selection system that implements our technique, and a set of empirical studies that demonstrate that the regression-test-selection algorithm can be effective in reducing the size of the test suite.

Keywords

Regression testing, testing, test selection, software evolution, software maintenance

[*]All authors except Gujarathi are affiliated with the College of Computing, Georgia Institute of Technology, Atlanta, GA. Gujarathi is affiliated with Citrix Systems, Inc, Ft. Lauderdale, FL.

1. INTRODUCTION

Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modifications. Because regression testing is expensive, researchers have proposed techniques to reduce its cost. One approach reduces the cost of regression testing by reusing the test suite that was used to test the original version of the software. Rerunning all test cases in the test suite, however, may still require excessive time. An improvement is to reuse the existing test suite, but to apply a *regression-test-selection* technique to select an appropriate subset of the test suite to be run. If the subset is small enough, significant savings in time are achieved. To date, a number of regression-test-selection techniques have been developed for use in testing procedural languages (e.g., [3, 8, 22, 29, 35, 37]) and for use in testing object-oriented languages (e.g., [14, 16, 19, 31, 36]).

Testing professionals are reluctant, however, to omit from a test suite any test case that might expose a fault in the modified software. A *safe* regression-test-selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program [32]. To date, several safe regression-test-selection techniques have been developed [3, 8, 29, 31, 35]. These techniques use some representation of the original and modified versions of the software to select a subset of the test suite to use in regression testing. Empirical evaluation of these techniques indicates that the algorithms can be very effective in reducing the size of the test suite while still maintaining safety [5, 12, 15, 30, 31, 35]. These studies also show that factors such as the structure of the code, location of the changes, granularity of the test suite, and frequency of the testing affect the reduction in test-suite size that can be achieved by the techniques.

Regression-test-selection techniques are particularly effective in environments in which changed software is tested frequently [15]. For example, consider an environment in which nightly builds of the software are performed and a test suite is run on the newly built version of the software. In this case, regression test selection can be used to select a subset of the test suite for use in testing the new version of the software. The main benefit of this approach is that, in many cases, a small subset of the test suite is selected, which reduces the time required to perform the testing. For another example, consider a development environment that includes such a regression-test-selection component. In this case, after developers modify their software, they can use the regression test selector to select a subset of the test suite to use in testing. With this approach, developers can frequently test their software as they make changes, which can help them locate errors early in development [15]. The techniques are also effective when the cost of test cases is high. An example is the regression testing of avionics software. In this case, even the reduction of one test case may save thousands of dollars in testing resources.

Although object-oriented languages have been available for some time, only two safe regression-test-selection algorithms that handle features of object-oriented software have been developed [31, 36]. However, both approaches are limited in scope and can be imprecise in test selection. Rothermel, Harrold, and Dedhia's algorithm [31] was developed for only a subset of C++, and has not been applied to software written in Java. The algorithm does not handle some features that are commonly present in object-oriented languages; in particular, it does not handle programs that contain exception-handling constructs. Furthermore, the algorithm must be applied to either complete programs or classes with attached drivers. For classes that interact with other classes, the called classes must be fully analyzed by the algorithm. Thus, the algorithm cannot be applied to applications that call external components, such as libraries, unless the code for the external components is analyzed with the applications. Finally, because of its treatment of polymorphism, the algorithm can be very imprecise in its selection of test cases. Thus, the algorithm can select many test cases that do not need to be rerun on the modified software.

White and Abdullah's approach [36] also does not handle certain object-oriented features, such as exception handling. Their approach assumes that information about that classes that have undergone specification or code changes is available. Using this information, and the relationships between the changed classes and other classes, their approach identifies all other classes that may be affected by the changes; these classes need to be retested. White and Abdullah's approach selects test cases at the class level and, therefore, can select more test cases than necessary.

This paper presents the first safe regression-test-selection technique for Java that efficiently handles the features of the Java language, such as polymorphism, dynamic binding, and exception handling. Our technique is an adaptation of Rothermel and Harrold's graph-traversal algorithm [29, 31], which uses a control-flow-based representation of the original and modified versions of the software to select the test cases to be rerun. Our new graph representation efficiently represents Java language features, and our graph-traversal algorithm safely selects all test cases in the original test suite that may reveal faults in the modified software. Thus, unlike previous approaches, our technique can be applied to common commercial software written in Java. Another novel feature of our technique is that the representation models (under certain conditions) the effects of unanalyzed parts of the software, such as libraries. Thus, the technique can be used for safe regression test selection of applications without requiring complete analysis of the libraries that they use. Because most Java programs make frequent use of libraries, such as the AWT [28], our technique's ability to select test cases for applications without requiring analysis of the library may provide significant savings during regression testing. Finally, the technique provides a new way to handle polymorphism that can result in the selection of a smaller, but still safe, subset of the test suite.

The paper also describes our regression-test-selection system, Retest, and a set of empirical studies that we performed using the system. Retest includes a profiler to determine coverage information for a program, a module that compares two program versions and identifies the parts of code that are affected by the modification, and a module that combines information from the other two modules to select regression test cases. Using Retest, we performed two empirical studies on a set of Java subjects. These studies show that, for the subjects we considered, our algorithm can provide significant reduction in the size of the test suite and suggest that the technique scales well to larger software systems. The studies also indicate that the granularity of the code entities on which the selection is based (e.g., edges, methods) can result in considerable differences in the sizes of the test suites selected by the technique.

In the next section, we discuss regression test selection in general; we also give details about Rothermel and Harrold's regression-test-selection algorithm, and illustrate it with an example. In Section 3, we discuss our algorithm, including details about the assumptions for safety. In Section 4, we discuss our regression-test-selection system, Retest, and in Section 5, we present the results of our empirical studies to evaluate our technique. In Section 6, we present related work and compare it to our work. Finally, in Section 7, we summarize our results and discuss future work.

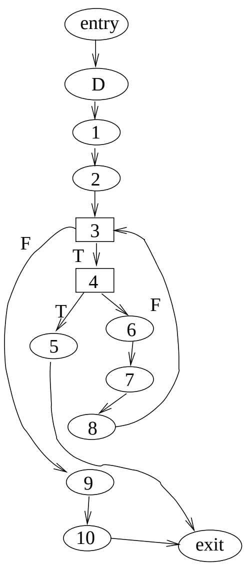
2. BACKGROUND

Software testing is the activity of executing a given program `P` with sample inputs selected from the input space for `P`, to try to reveal failures in the program. The underlying idea is that, under the hypothesis that a test case is well-designed, its inability to reveal failures increases our confidence in the tested code. To perform testing, it is necessary to select a set of input data for `P`, to determine the expected behavior of `P` for such data with respect to a given model, and to check the results of `P`'s execution against the expected behavior. A *test case* consists of the input data that is provided to `P`, together with the corresponding expected output. A *test suite* is a set of test cases, and a *test run* is the execution of `P` with respect to a test suite `T`. For each test case t in `T`, a test run for `T` consists of the following three steps: (1) initialization of the environment, (2) execution of `P` with input specified in t, and (3) checking of the output of the execution with respect to the expected output.

To assess the adequacy of a given test suite `T`, we measure the level of coverage achieved by the test run. Although functional coverage can be measured as well, coverage is usually computed for structural entities (i.e., entities in the code). Common examples of entities considered for structural coverage are statements, branches, and conditions. Coverage is measured as a percentage. For example, statement coverage is defined as the percentage of statements covered by the test run with respect to the total number of executable statements. The coverage information can be obtained in several ways. One method produces an

```
procedure avg

 1. int count = 0
 2. fread(fileptr,n)
 3. while (not EOF) do
 4.    if (n<0)
 5.       return(error)
       else
 6.       numarray[count] = n
 7.       count++
       endif
 8.    fread(fileptr,n)
    endwhile
 9. avg = calcavg(numarray,count)
10. return(avg)
```



```
procedure avg'

 1. int count = 0
 2. fread(fileptr,n)
 3. while (not EOF) do
 4.    if (n<0)
5a.       print("bad input")
 5.       return(error)
       else
 6.       numarray[count] = n

       endif
 8.    fread(fileptr,n)
    endwhile
 9. avg = calcavg(numarray,count)
10. return(avg)
```

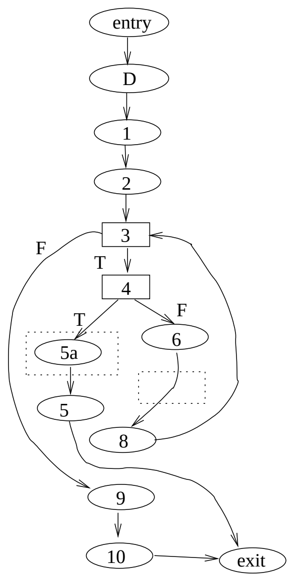


Figure 1: Example program, avg and its control-flow graph (left); modified version, avg′ and its control-flow graph (right).

instrumented version of P such that when this instrumented version of P is executed with a test case t, it records the entities in the program, such as statements or branches, that are executed with t. An alternative method for obtaining the coverage information modifies the runtime environment so that when P is executed with t, the environment gathers the information about the entities covered.

Let P′ be a modified version of P, and T be the test suite used to test P. During regression testing of P′, T and information about the testing of P with T are available for use in testing P′. In attempting to reuse T for testing P′, two problems arise. First, which test cases in T should be used to test P′ (the *regression-test-selection* problem). Second, which new test cases must be developed to test parts of P′ such as new functionality (the *test-suite augmentation* problem). Although both problems are important, in this paper we concentrate on the regression-test-selection problem. Regression-test-selection techniques attempt to reduce the cost of regression testing by selecting T′, a subset of T, and using T′ to test P′.

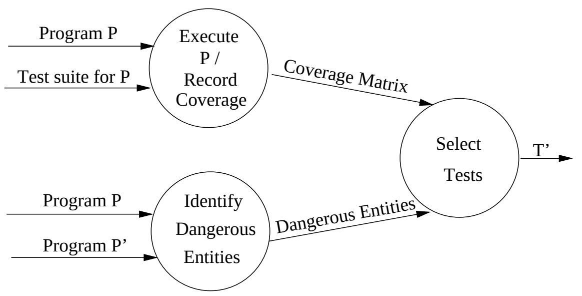


Figure 2: A general system for regression test selection.

A number of safe regression-test-selection techniques that vary in precision and efficiency have been presented (e.g., [3, 8, 29, 31, 35]). We can view these techniques as a family of regression-test-selection techniques that use information about the program's source code to select T′. Figure 2 illustrates a general regression-test-selection system.

In this system, a program P is executed with a test suite T. In addition to the results of the execution—the pass/fail information—the system records coverage information about which entities in P are executed by each test case t. The types of entities recorded depend on the specific regression-test-selection technique. After all test cases have been run, the coverage information is compiled into a coverage matrix that associates each t in T with the entities that it executes.

In addition to computing coverage information, these techniques compare P and P′, and identify in P a set of dangerous entities [4]. We define P(i) as the execution of P with input i. A *dangerous entity* is a program entity e such that for each input i causing P to cover e, P(i) and P′(i) may behave differently due to differences between P and P′.[1] The technique ensures that any test case that does not cover a dangerous entity will behave in the same way in both P and P′, and thus, cannot expose new faults in P′. Thus, it is safe to select only those test cases for which the coverage matrix indicates coverage of a dangerous entity.

Rothermel and Harrold, for example, describe a regression-test-selection technique that uses a control-flow graph (CFG)[2] to represent each procedure in P and P′ and uses edges in the CFGs as potential dangerous entities [29]. Dangerous entities are selected by traversing in parallel the CFGs for P and the CFGs for P′; whenever the targets of like-labeled CFG edges in P and P′ differ, the edge is added

[1]Although an entity is not dangerous in itself, and the term *affected* (by a change) may be more appropriate, we use the term *dangerous entity* to be consistent with the terminology used in Reference [4].

[2]In a *control-flow graph*, nodes represent program statements and edges represent the flow of control between the statements.

to the set of dangerous entities.

To illustrate, consider Figure 1, which shows an example program `avg` and its control-flow graph (left) and a modified version `avg′` and its control-flow graph (right). From `avg` to `avg′`, statement 5a has been inserted and statement 7 has been deleted. The algorithm begins the traversal at entry nodes in `avg` and `avg′`, and traverses like paths in the two programs by traversing like-labeled edges until a difference in the target nodes of such edges is detected. When the algorithm considers node 4 in `avg` and node 4 in `avg′`, it finds that the targets of the edges with label "T" differ, and it adds edge (4, 5) to the set of dangerous entities. The algorithm stops its traversal along this path: any test case that traverses other dangerous entities, if any, that can be reached only through this edge (i.e., dominated[3] by this edge) would necessarily traverse this edge, and thus, is already selected. The algorithm then considers the "F" labeled edges from node 4 in `avg` and node 4 in `avg′`. When the algorithm considers node 6 in `avg` with node 6 in `avg′`, it discovers that the targets of the out edges differ; therefore, it adds edge (6, 7) to the set of dangerous entities, and stops the traversal along this path. Subsequent traversals find no additional dangerous edges. In the example, the nodes labeled "D" are declaration nodes. In the algorithm, *declaration nodes* are used to model variable declarations explicitly; such nodes contain information about all declarations. By doing so, the algorithm can identify changes that involve only declarations of variables, which do not appear in the CFG. For example, if the declaration of variable `count` in `avg′` changed from `int` to `long`, the algorithm would find a difference between the two declaration nodes and add edge (entry, D) to the set of dangerous entities.

After dangerous edges have been identified, the select-tests component of the regression-test-selection system uses the dangerous entities and the coverage matrix to select the test cases in `T` to add to `T′`.

Continuing with the above example, suppose the test suite shown in Table 1 were used for `avg`.

Table 1: A test suite for program `avg` of Figure 1.

Test Case	Input	Expected Output
1	empty file	0
2	-1	error
3	1 2 3	2

When the program executes, the edge covered by each test case in the test suite are recorded. For this example, test cases 1, 2, and 3 cover edge (entry, 1), test cases 1 and 3 cover edge (2, 9), and test cases 2 and 3 cover edge (3, 4). Table 2 shows the edge-coverage matrix for this test suite and program `avg`.

Table 2: Edge-coverage matrix for test suite of Table 1 and program `avg` of Figure 1.

Edge	Test Case
(entry, 1), (1, 2), (2, 3)	1, 2, 3
(3, 9), (9, 10), (10, exit)	1, 3
(3, 4)	2, 3
(4, 5), (5, exit)	2
(4, 6), (6, 7), (7, 8), (8, 3)	3

[3]An edge e_i dominates a edge e_j if every path from the beginning of the program to e_j goes through e_i.

Using the edge-coverage matrix and the set of dangerous entities computed by the regression-test-selection algorithm, the final step in the test selection is performed by simply indexing into the matrix using the dangerous entities, and returning the corresponding test cases. In our example, the dangerous edges are (4, 5) and (6, 7). Thus, test cases 2 and 3 need to be rerun.

3. REGRESSION TEST SELECTION FOR JAVA

The regression-test-selection technique for Java that we present is also control-flow based. Like the technique described in Section 2 [29], our technique performs three main steps. First, it constructs a graph to represent the control flow and the type information for the set of classes under analysis. Then, it traverses the graph to identify dangerous edges. Finally, based on the coverage matrix obtained through instrumentation, it selects, from the test suite for the original program, the test cases that exercise the dangerous edges identified in the previous step.

For efficiency, our technique avoids analyzing and instrumenting code components, such as libraries, that are used by the program but have not been modified. Therefore, we can consider the program being tested as divided into two parts: one part that we analyze and the other that we consider as external and do not analyze. For convenience, in the rest of the paper we refer to the set of classes that are analyzed and instrumented by our regression-test-selection system as *internal classes*, and those that are not analyzed and instrumented by our system as *external classes*. Analogously, we refer to a method in an internal class as an *internal method* and a method in an external class as an *external method*.

In the rest of this section, we present the details of our regression-test-selection technique for Java: we describe the assumptions on which the technique is safe; we illustrate the representation used to model Java programs; we present the traversal algorithm; and we discuss two alternative approaches for Java-program instrumentation that allow for gathering the coverage information required by our regression-test-selection technique.

3.1 Assumptions

To be safe without being too inefficient and too conservative, our technique must rely on some assumptions about the code under test, the execution environment, and the test cases in the test suite for the original program. Bible and Rothermel call this set of assumptions the *regression bias* [4].

Reflection. Our technique assumes that reflection is not applied to any internal class or any component of an internal class. Reflection "allows programmatic access to information about the fields, methods and constructors of loaded classes, and the use of reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restriction" [27]. In this paper, we consider methods that inspect the information about a specific class, such as the methods in `java.lang.Class`, as a form of reflection as well. If a statement uses information obtained through reflection about either an internal class or its members, the behavior of this statement may be affected by several kinds of changes occurring in the class and/or its members. In such cases, the identification of all the points in the code affected by a change may require sophisticated and expensive analysis of all the reflection constructs in the code. Moreover, if a statement in an external class uses reflection to inspect the information about an internal class, then the external class must be analyzed to identify the code affected by a change of the internal class.

Independent external classes. Our technique assumes that the external classes can be compiled without the internal classes, and that external classes do not load any internal class explicitly by invoking a class loader with the class name as a parameter. In other words, we assume that external code has no knowledge of the internal classes. This assumption guarantees that the external classes interact with the internal classes only through a set of predefined virtual methods. Thus, this assumption reduces the types of interactions between the internal and external classes that we must consider. In practice, this assumption holds in most cases because the external classes are often library classes that are developed independent of, and prior to, the development of the applications that use them.

Deterministic test runs. Our technique assumes that a test case covers the same set of statements, and produces the same output, each time it is run on an unmodified program. This assumption guarantees that (1) the coverage information contained in the coverage matrix, obtained by running the original program with the test cases, does not depend on the specific test run, and (2) the execution of a test case that does not traverse affected parts of the code yields the same results for the original and the modified programs. Under this assumption, if our representation of the internal classes is correct (i.e., it correctly models all language constructs addressed and their effects), based on the information in the coverage matrix, we can safely exclude test cases that do not traverse modifications.

One possible threat to this assumption is a change in the execution environment. If the execution environment changes between the test run on the original program and the test run on the modified program, the outcome of a test case may change even if the test execution does not traverse the parts of code that are affected by the modification. Therefore, for our technique to be applicable, the tester must ensure that elements such as the operating system, the Java Virtual Machine, the Java compiler, the external classes, databases and network resources possibly interacting with the program are fixed. This requirement, however, is not overly restrictive—it is a common requirement for testing in general because it guarantees that, in case of a failure, the problem can be reproduced.

Another possible threat to this assumption is the presence of nondeterministic behavior. The assumption holds for sequential programs, which contain only one thread of execution, and for those multithreaded programs in which the interaction among threads does not affect the coverage and the outputs (e.g., an ftp server whose multiple threads are just clones created to handle multiple clients). The assumption, however, does not hold in general for programs that contain multiple threads of execution. For our technique to be applicable to such programs, we must use special execution environments that guarantee the deterministic order in which the instructions in different threads are executed [24]. This threat too, like the previous one, is a requirement for testing in general, and therefore, is not unduly restrictive.

3.2 Representation for Java Software

To adequately handle all Java language constructs, a representation based on simple CFGs, such as the one presented in Section 2, is inadequate. A CFG is suitable for representing the control flow within a single procedure, but cannot accommodate interprocedural control flow (i.e., control flow across procedure boundaries) or features of the Java language such as inheritance, polymorphism and dynamic binding, and exception handling.

We define a representation that, although general enough to represent software written in other object-oriented languages, is tailored to Java. For convenience, we refer to this representation as Java Interclass Graph. A *Java Interclass Graph* (JIG) accommodates the Java language features and can be used by the graph-traversal algorithm to find dangerous entities by comparing the original and modified programs. A JIG extends the CFG to handle five kinds of Java features: (1) variable and object type information; (2) internal or external methods; (3) interprocedural interactions through calls to internal or external methods from internal methods; (4) interprocedural interactions through calls to internal methods from external methods; and (5) exception handling. We next discuss how the JIG represents each of these characteristics and constructs of Java. We also show that the representation is suitable for modeling both complete and partial programs, such as classes, clusters, or components in general, because it accounts for the possible effects of missing parts of the system.

Variable and object types. The graph-traversal algorithm described in Section 2, uses a representation that contains *declaration* nodes to represent declarations in the program. If a change is made in a global declaration, then the edge from the entry node to the corresponding declaration node is marked as dangerous, thus causing all test cases in the test suite to be selected.

To achieve better precision, in our representation we associate the type of a variable that is of primitive type with the name of the variable by augmenting the name with the type information. For example, a variable `a` of type `double`, is identified as `a_double` in our representation. This method for representing scalar variables essentially pushes any change in the variable type down to the location where that variable is referenced, which gives more precise test selection than the use of the declaration node.

The graph-traversal algorithm described in Reference [31] models classes and class hierarchies explicitly. In our representation, instead of explicitly representing class hierarchies, we encode the hierarchy information at each point of instantiation (e.g., at each point in the code where `new` is invoked) using a globally-qualified class name. A *globally-qualified class name* for a class contains the entire inheritance chain from the root of the inheritance tree (i.e., from class `java.lang.Object`) to its actual type.[4] The interfaces that are implemented by the class are also in-

[4]For efficiency, a globally-qualified class name can exclude external classes and interfaces.

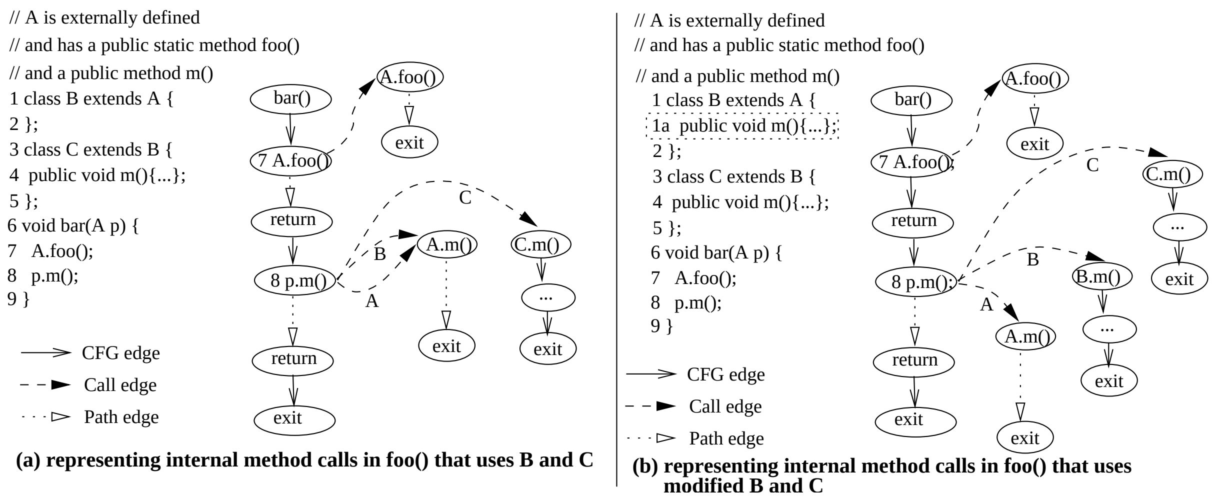


Figure 3: Example of internal method call representation.

cluded in globally-qualified names. If a class implements more than one interface, the names of the interfaces are inserted in the globally-qualified name in alphabetical order. For example, if a class `B` in package `foo` extends a class `A` in the same package, and `A` implements interface `I` in package `bar`, then the globally-qualified name for `B` is `java.lang.Object:bar.I:foo.A:foo.B`.

Using globally-qualified class names, our technique can identify the changes in class hierarchies. This method for representing class hierarchies also pushes the changes in class hierarchies to the locations where the affected classes are instantiated. Therefore, our technique can be very precise in accounting for such changes.

Internal or external methods. A JIG contains a CFG for each internal method in the set of classes under analysis. The CFG differs from the one described in Section 2 in two ways. First, each call site is expanded into a *call* and a *return* node. Second, there is a *path* edge between the call and return node that represents the path through the called method. The graph on the left of Figure 4 illustrates this representation. The node labeled "p.m()" represents a call node; it is connected to the return node with a path edge.

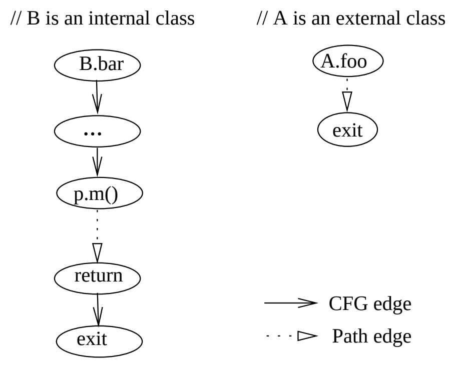


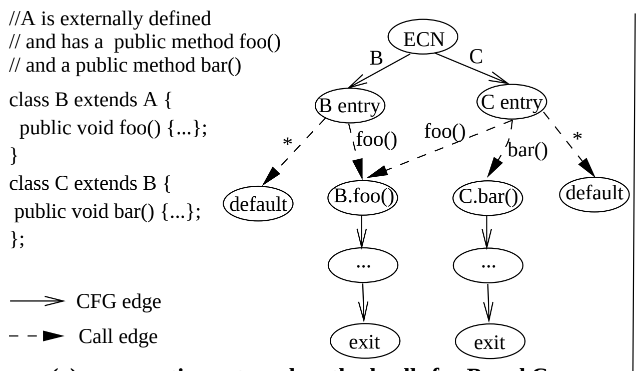
Figure 4: Example of internal method representation (left) and external method representation (right).

A JIG contains a collapsed CFG for each external method that is called by any internal method. Usually, the source code for the external classes is not available, and, even if it were available, we do not want to analyze it. Because we assume that external classes do not change, there is no need to represent and analyze such code. Thus, each collapsed CFG consists of a method entry node and a method exit node along with a path edge from the method entry node to the method exit node. The path edge summarizes the paths through the method. The graph on the right in Figure 4 illustrates this representation.

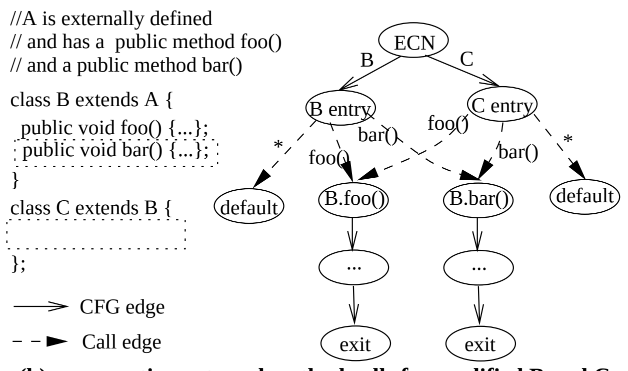
Interprocedural interactions through internal method calls. The JIG represents each call site as a pair of call and return nodes that are connected with a path edge. The call node is also connected to the entry node of the called method with a *call edge*. If the call is not virtual, the call node has only one outgoing call edge. To illustrate, consider Figure 3(a), which shows three classes, `A` (external), `B`, and `C`, a method `bar`, and the corresponding JIG. Class `B` extends class `A` without overriding any method; class `C` extends class `B` and overrides method `m`.

In the example, method `bar` invokes static method `A.foo`; there is no dynamic binding at this call site because `A.foo` is a static method. Thus, the call node (node 7) has only one outgoing call edge connected to the entry node for method `A.foo`.

If the call is virtual, the call node is connected to the entry node of each method that can be bound to the call. Each call edge from the call node to the entry node of a method `m` is labeled with the type of the receiver instance that causes `m` to be bound to the call. In the example, the call to `p.m` in method `bar` is a virtual call. Depending on the dynamic type of `p` (whose static type is `A`), the call to `m` is bound to different methods: if `p`'s type is either `A` or `B`, the call is bound to `A.m`; if `p`'s type is `C`, the call is bound to `C.m`. Consequently, the call node has three outgoing call edges: two of them, labeled "A" and "B", are connected to the entry node for `A.m`; the third edge, labeled "C", is connected to the entry node for `C.m`.



(a) representing external method calls for B and C



(b) representing external method calls for modified B and C

Figure 5: Example of external method call representation.

To represent virtual method calls correctly, we must compute, for each virtual call site, the set of methods to which the call may be bound. Such information can be computed using various type-inferencing algorithms (e.g., [2, 10, 34]) or points-to analysis algorithms (e.g., [23]). The precision of this computation determines the efficiency of the representation. In our technique, we use the class hierarchy analysis [10] to resolve the virtual calls.

Using this representation, our algorithm can identify, by traversing the JIGs constructed for the original and modified programs, the internal method calls that may be affected by a program change. Figure 3(b) shows a modification to class `B` that adds a new method `m()` in `B` (statement 1a). This change affects the method that is invoked at statement 8 when `p`'s type is `B`. By comparing the outgoing edges from the call node associated with statement 8 in the JIG in Figure 3(a) and the outgoing edges from the call node associated with statement 8 in the JIG in Figure 3(b), our algorithm identifies that the target of the edge labeled "B" has changed. Thus, our algorithm identifies this edge as dangerous.

Interprocedural interactions through external method calls. Potential subtle interactions between internal classes and external classes may lead to different behavior in the program, as a consequence of apparently harmless changes in the internal classes. Therefore, in the case of incomplete programs, we must consider the possible effects of the unanalyzed parts of the system. In particular, unforeseen interactions between internal classes and external classes may be caused by calls from external methods to internal methods. To handle this situation, we explicitly represent, in a summarized form, potential calls to internal methods from external methods.

Figure 5(a) provides an example of such representation. External code is represented by a node labeled "ECN" (External Code Node). For each internal class that is accessible from external classes (both classes `B` and `C` in the example), the JIG contains an outgoing edge from the `ECN` node. Each of these edges is labeled with the name of the class it represents and terminates in a class entry node.

A *class entry node* for an internal class A represents an entry point to the internal code through an object of type A, and is connected to the entry of each method that can be invoked by external methods on objects of type A. The only internal methods that can be invoked by external code are those methods that override an external method.[5] Therefore, we must create a class entry node for (1) each class that overrides at least one external methods, and (2) each class that inherits at least one method overriding an external method. For example, in Figure 5(a), class `C` overrides `A.bar` and inherits `B.foo`, which in turn overrides `A.foo`; thus, we must create a class entry node for `C` and connect it to `B.foo` and `C.bar`, which can both be invoked on objects of type `C` by external code through polymorphism and dynamic binding.

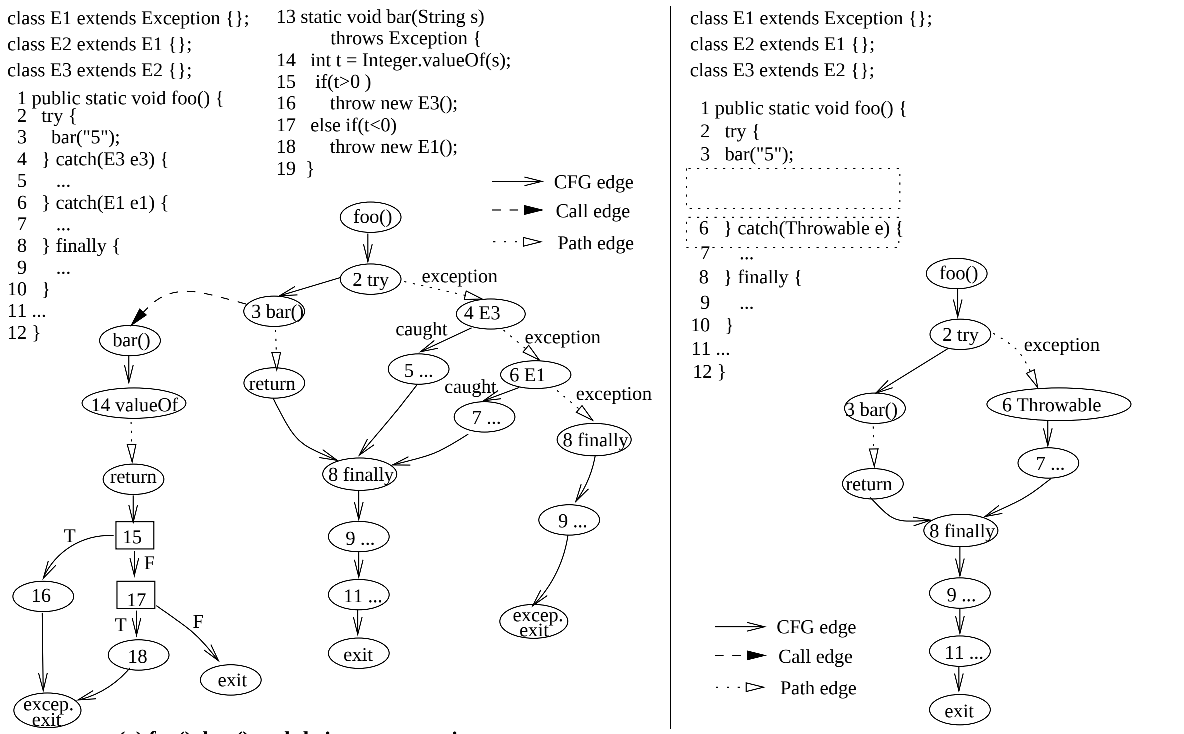
In addition, for each class entry node, we create an outgoing default call edge labeled "*" (see Figure 5(a)), and we connect it to a *default node*. The *default node* for a class `A` represents all methods that can be invoked through an object of a type `A`, but that are externally defined. This representation lets us correctly handle modifications that involve addition or removal of internal methods that override external methods,

Using this representation, our algorithm can identify, by traversing the JIGs constructed for the original and modified programs, the external method calls that may be affected by a program change. Figure 5(b) shows the modification to classes `B` and `C` that deletes `bar()` from `C` and adds a new `bar()` in `B`. This change may affect an external call to these methods when the receiver type is either `B` or `C`. By comparing the outgoing edges from the node labeled "C entry" in the graph in Figure 5(a) with the outgoing edges from node labeled "C entry" in the graph in Figure 5(b), our algorithm identifies that the target of the edge labeled "bar()" has changed. Thus, our algorithm identifies this edge as dangerous.

Exception handling. The JIG uses an approach similar to that described in Reference [33] to model exception-handling constructs in Java code.

A JIG explicitly represents the try block, the catch blocks,

[5]As stated in Section 3.1, we assume that external code has no knowledge of internal methods and, therefore, can call internal methods only through polymorphism and dynamic binding.



```
class E1 extends Exception {};
class E2 extends E1 {};
class E3 extends E2 {};
 1 public static void foo() {
 2   try {
 3     bar("5");
 4   } catch(E3 e3) {
 5     ...
 6   } catch(E1 e1) {
 7     ...
 8   } finally {
 9     ...
10   }
11 ...
12 }
```

```
13 static void bar(String s)
        throws Exception {
14   int t = Integer.valueOf(s);
15   if(t>0 )
16     throw new E3();
17   else if(t<0)
18     throw new E1();
19 }
```

(a) foo(), bar() and their representations

```
class E1 extends Exception {};
class E2 extends E1 {};
class E3 extends E2 {};
 1 public static void foo() {
 2   try {
 3     bar("5");

 6   } catch(Throwable e) {
 7     ...
 8   } finally {
 9     ...
10   }
11 ...
12 }
```

(b) a modified version of foo() and its representation

Figure 6: Example of exception handling representation.

and the finally block in each *try* statement. Figure 6(a) shows an example JIG for a method that contains exception-handling constructs. For each try statement, we create a *try node* in the CFG for the method that contains the statement (node labeled "2 try" in the example). We represent the try block of the try statement using a CFG. There is a CFG edge from the try node to the entry of the CFG of the try block.

We create a *catch node* and a CFG to represent each catch block of the try statement. A catch node is labeled with the type of the exception that is declared by the corresponding catch block. A CFG edge, labeled "caught", connects the catch node to the entry of the catch CFG. A path edge, labeled "exception", connects the try node to the catch node for the first catch block of the try statement. That path edge represents all control paths, from the entry node of the try block, along which an exception can be propagated to the try statement. For example, path edge (2, 4) in the graph in Figure 6(a) represents all control paths that traverse statement 2 and reach statement 16 or 18, or a throw statement in `Integer.valueOf()` that causes `Integer.valueOf()` to propagate an exception. A path edge labeled "exception" connects the catch node for a catch block b_i to the catch node for catch block b_{i+1} that follows b_i. This path edge represents all control paths, from the entry node of the try block, along which an exception is (1) raised, (2) propagated to the try statement, and (3) not handled by any of the catch blocks that precede b_{i+1} in the try statement.

We create a *finally node* and a CFG to represent the finally block of the try statement. A CFG edge connects the finally node to the entry of the CFG. For each CFG that represents the try block or a catch block, a CFG edge connects the exit node of this CFG to the finally node. The exit of the CFG for the finally block is connected to the statement that follows the try statement. If there are exceptions that cannot be caught by any catch block of the try statement, we create a copy of the finally node and of the CFG for the finally block. A path edge labeled "exception" connects the catch node of the last catch block to this finally node. If the try statement is not enclosed in another try statement in the same method, a CFG edge connects the exit of this duplicated finally CFG to the exceptional exit node. An *exceptional exit node* models the effect of an uncaught exception causing exit from the method. If no finally block is present and the try statement is not enclosed in any other try statement in the same method, a path edge labeled "exception" connects the catch node of the last catch block of the try statement to the exceptional exit node.

Using this representation, our algorithm can identify the changes in exception-handling code by traversing the JIGs constructed for the original and the modified programs. Figure 6(b) shows a modified version of `foo()` in which we delete statements 4 and 5 and change the type of exception handled by statement 6. By comparing the outgoing edges from the try node in the graph in Figure 6(a) and the outgoing edges from the try node in the graph in Figure 6(b), our algorithm discovers that the target of the edge labeled "exception" has changed. Thus, our algorithm identifies this edge as dangerous.

```
Procedure      Compare($N$,$N'$)
input          $N$: a node in the JIG for original program $P$
               $N'$: a node in the JIG for modified program $P'$
global output  $\mathcal{E}$: set of dangerous edges for $P$
begin Compare
1.  mark $N$ “$N'$-visited”
2.  foreach edge $e'$ leaving $N'$ do
3.      $e$ = match($N$,$e'$)
4.      if $e$ is null then continue
5.      $C = e$.getTarget()
6.      $C' = e'$.getTarget()
7.      if $\neg$NodesEquiv($C$,$C'$) then
8.          $\mathcal{E} = \mathcal{E} \cup e$
9.      elseif $C$ is not marked “$C'$-visited”
10.         Compare($C$,$C'$)
11.     endif
12. endfor
13. foreach edge $e$ leaving $N$ and
            not matched to any edge leaving $N'$ do
14.     $\mathcal{E} = \mathcal{E} \cup e$
15. endfor
end Compare
```

Figure 7: `Compare` procedure.

3.3 The Traversal Algorithm

Our algorithm that traverses the JIGs and identifies dangerous edges is similar to the algorithm in Reference [29] that traverses the CFG of a procedure. Our algorithm starts the traversal by invoking `Compare()` on the entry node of method `main()`, on the `ECN` node, and on the entry nodes of all methods called *static*.[6] `Compare()` accepts as inputs a node N in the JIG constructed for the original program `P` and a node N' in the JIG constructed for the modified program `P'`; it traverses the JIGs and adds the dangerous edges that it finds to $\mathcal{E}$. `Compare()` first marks N as “N'-visited” (line 1) to avoid comparing N with N' again in a subsequent iteration. `Compare()` then examines each outgoing edge e' from N' and calls `match()` to find an outgoing edge from N that matches e''s label (line 3). `match()` first looks for an outgoing edge from N that has the same label as e'. If `match()` finds such an edge, it returns this edge. Otherwise, `match()` looks for the edge whose label is “*”. If `match()` finds such an edge, it returns this edge. Otherwise, it returns `null`.

After `Compare()` finds the edge e that matches e', it compares C, the target of e, with C', the target of e' (lines 5–7). If C is not equivalent to C', `Compare()` adds e to set $\mathcal{E}$ (line 8). Otherwise, if C has not been marked with “C'-visited”, `Compare()` invokes itself on C and C' to further traverse the graph (line 10).

One way to determine the equivalence of two nodes is to examine the lexicographic equivalence of the text associated with the two nodes [29]. For nodes that represent program statements, we can examine the lexicographic equivalence of the program statements associated with the nodes; for nodes, such as exit nodes, that do not represent program statements, we can examine the lexicographic equivalence of the labels associated with the nodes.

After `Compare()` finishes processing the outgoing edges from N', it searches for outgoing edges from N that have not been matched with any outgoing edge from N' (line 13). These edges appear along paths that have been deleted in `P'`. Thus, `Compare()` adds these edges to set $\mathcal{E}$ (line 14).

[6]The Java compiler creates, for each class containing initializers for static fields, a special method called *static*, which contains all such initializations and is executed when the class is initialized.

3.4 Instrumentation for Test Selection

Given a JIG for a program `P`, we can instrument `P` or modify the execution environment to record the edges covered by each test case (the task of gathering coverage was discussed in Section 2). The coverage information lets the regression-test-selection technique select test cases that cover the dangerous edges identified by the traversal algorithm. However, because some edges (e.g., path edges for exception handling) do not represent actual control flow from one statement to another, we cannot instrument the program or the execution environment to find the test cases that cover such edges. Moreover, because recording the coverage information for each edge can be very expensive, we may want to record coverage information for coarser-grained entities, such as methods, classes, or modules. Thus, some dangerous edges must be mapped to another set of entities whose coverage information is recorded in the coverage matrix.

In general, we need an adaptor that takes a set of dangerous edges from the traversal algorithm and maps them to a set of dangerous entities whose coverage information is recorded in the coverage matrix. The adaptor must be designed together with the instrumenter because it needs to know the entities whose coverage information is being recorded. To get a better trade-off between precision and efficiency, the instrumenter also needs to know which entities are of interest to the adaptor. In the following, we discuss a possible approach for building an adaptor and an instrumenter in the case of instrumentation at the edge and at the method level.

Edge-level instrumentation techniques record, for the internal methods, the CFG edges that are covered by each execution of a program `P`. To determine the virtual call edges that are covered by the execution, we require the instrumenter to also record the receiver type of each virtual method call in internal methods.

In a JIG, edges representing calls from external methods (see Figure 5) and path edges representing the control paths on which exceptions are raised (see Figure 6) need to be mapped to actual CFG edges and nodes, so that the instrumenter can record the test cases that cover such edges.

For an edge e_1 representing a method call from an external method and whose receiver is an instance of an internal class C, if the target of e_1 is the entry of a method, the adaptor maps e_1 to the corresponding entry node. However, if the target of e_1 is the default node created for C (e.g., the nodes labeled “default” in Figure 5), the adaptor maps e_1 to all new-instance statements in the internal classes that create instances of C.

To record the coverage information for exception handling, we require the instrumenter to insert a catch block at the end of each try statement to catch and to re-throw all exceptions that are not caught by other previous catch blocks. Let e_2 be a path edge that represents the control paths on which an exception is raised. If the target of e_2 is a catch node for a catch block b, then e_2 is mapped to the entry of b and the entry of each of the catch blocks that

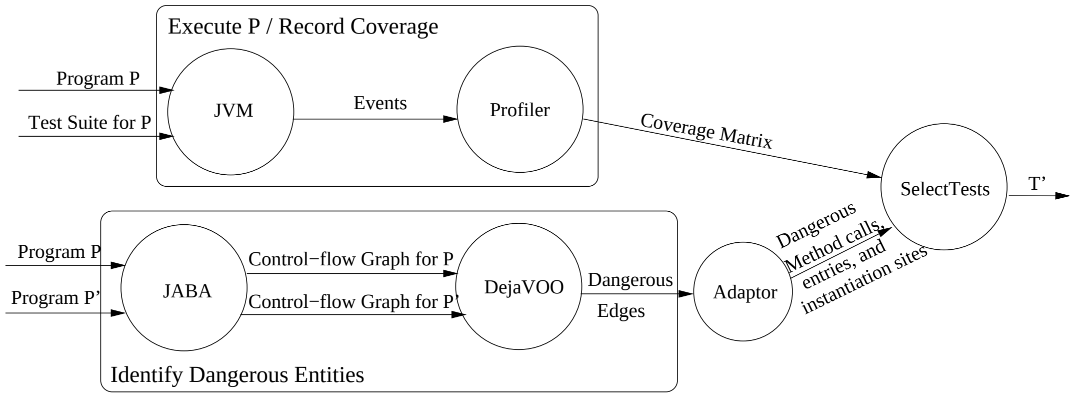


Figure 8: Our regression-test-selection system, RETEST.

appear after b in the try statement. e_2 is also mapped to the entry of the catch block added by the instrumenter.

Method-level instrumentation techniques record the internal methods that are covered by each execution of the program. Using this instrumentation technique, the adaptor maps each dangerous edge in the JIG for a method m to m. For each call edge e, if the target of e is the entry node of an internal method m, the adaptor maps e to m. Otherwise, if the target of e is the entry node of an external method, the adaptor maps e to the method that contains the source of e. However, if e is an edge that represents a call from external methods and e's target is the default node (e.g., the nodes labeled "default" in Figure 5) created for an internal class C, the adaptor maps e to all the constructors (including the default constructor) of C.

Instrumentation at a coarser level of granularity is more efficient than instrumentation at a finer level of granularity. In particular, method-level instrumentation is more efficient than edge-level instrumentation. Also, the coverage matrix computed using method-level instrumentation is smaller than the one computed using edge-level instrumentation. However, using a coverage matrix computed by method-level instrumentation, a test-selection algorithm may select more test cases than using a coverage matrix computed by edge-level instrumentation.

In practice, another viable solution is a hybrid instrumentation approach that, for example, records coverage information for all internal methods, and also records coverage information for those statements, such as exception handling, that are rarely covered (even when the method that contains the statements is executed). Hybrid approaches may yield a good trade-off between precision and efficiency.

4. REGRESSION TEST SELECTION SYSTEM

To investigate empirically the regression-test-selection technique presented in this paper, we implemented a regression-test-selection system named RETEST, which is a specialization of the general regression-test-selection system shown in Figure 2. RETEST consists of three main components: a component named *Profiler*, which gathers dynamic information, a static analysis component named *DejaVOO*, and a test-selection component named *SelectTests*. RETEST also consists of a component, *Adaptor*, which adapts the output produced by DejaVOO to the type of coverage information gathered by the profiler. Figure 8 presents the system architecture of RETEST.

The profiler component in RETEST uses the Java Virtual Machine Profiler Interface (JVMPI) [26] to gather coverage information about `P` when `P` is run using each test case in `T`. However, due to restrictions in JVMPI, the current profiler cannot record information about the execution of individual statements other than class instantiation, method entries, and method calls. The profiler also cannot record information for exception throws and catches. Therefore, the profiler uses a hybrid instrumentation approach: it records class instantiation, method calls, and method entries. At a virtual method call, the profiler also records the receiver type so that it can determine the method that is actually invoked.

The DejaVOO module implements the analysis algorithm described in Section 3, using the Java Architecture for Bytecode Analysis (JABA) [1] to construct JIGs and other necessary information about `P` and `P'`. The output from DejaVOO is a list of dangerous edges that can be either CFG edges or call edges.

The adaptor module treats CFG edges and call edges differently. The adaptor maps each dangerous CFG edge e in a method `m` to the entry of `m`, unless the target of e is a class instantiation or a method call. If this is the case, the adaptor maps e to the node representing the class instantiation or to the call node, respectively. The adaptor maps each call edge (m_i, m_e), which represents a call from an internal method m_i to an external method m_e, to m_i's entry. The adaptor maps each call edge (A *entry*, m_i), which represents a call from an external method to an instance of A that is dynamically bound to an internal method m_i, to m_i's entry. Finally, the adaptor maps each call edge (A *entry*, $default$), which represent a call from an external method to an instance of A that is dynamically bound to an external method, to all the instantiation sites for A.

The current implementation of RETEST is less precise than it could be using the technique described in this paper. This imprecision occurs because the profiler cannot record information for each individual statement. Therefore, if a change

Table 3: Software subjects used for the empirical studies.

Subject	Description	Methods	Versions	Test cases	Method coverage
Siena	Internet-based event notification system	185	7	138	70%
JEdit	Text editor	3495	11	189	75%
JMeter	Web-applications testing tool	109	5	50	67%
RegExp	Regular-expression library	168	10	66	46%

occurs only in statements that are traversed by a small fraction of the test cases that cover the method, RETEST can select more test cases than necessary. For example, if an exception handler that catches infrequently-raised exceptions has changed, each test case going through the method that contains the handler is selected, even though the exception is not raised by a majority of these test cases.

We plan to investigate techniques to enhance our profiler by using the Java Virtual Machine Debug Interface (JVMDI) [25] instead of the JVMPI. The JVMDI allows more detailed examination of programs as they run, and should let us reduce the imprecision described above. We will also investigate the possibility of instrumenting the bytecode using a tool such as BIT (Bytecode Instrumenting Tool)[7] [9], Soot (a Java Optimization Framework),[8] or BCEL (Byte Code Engineering Library),[9] to gather the CFG-edge coverage information. With this approach, we may be able to improve the precision of the test selection.

5. EMPIRICAL EVALUATION

To evaluate our approach for regression test selection, we used RETEST to perform two empirical studies. This section describes the software subjects and the empirical studies that we performed.

5.1 Software Subjects

Our study utilized four software subjects: SIENA, JEDIT, JMETER[10], and REGEXP. Each software subject consists of an original version `P`, several modified versions (V1, ... Vn), and a set of test cases that was used to test `P`. Table 3 shows the subjects and, for each subject, lists the number of methods in the original program, the number of versions, the size of the test suite, and the percentage of methods covered by the test suite.

The first subject for our studies is the Java implementation of the SIENA server [6]. SIENA (Scalable Internet Event Notification Architecture) is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks. We obtained seven successive versions of the system from the authors, along with a test suite that had been used to test the software. We also added test cases to the test suite to increase its coverage (at the method level); the resulting test suite contains 138 test cases, which provide 70% method coverage of the system.

The second subject for our studies is JEDIT, a Java-based text editor. JEDIT is a versatile, customizable text editor, which provides several advanced text-editing features, such as syntax highlighting, regular-expression search and replace, multiple clipboards, and macro recording. We obtained two successive development releases of the software—version 3.0-pre4 and version 3.0-pre5—and, based on the changes between the releases, created 11 versions of the software. The original version of the software contains 3495 methods. We then developed a test suite to exercise various features of the text editor; the test suite consists of 189 test cases and provides 75% method coverage.

The third subject for our studies is Apache JMETER. JMETER is a Java desktop application designed to load test the functional behavior and measure performance; it was originally designed for testing web applications but has since expanded to other test applications. We obtained two successive releases of the system from the code repository and, based on the modifications between the releases, created five versions of the software. We created 50 test cases by considering combinations of features available through the user interface of the system; the test cases provide 67% method coverage of the system.

The final subject for our studies is REGEXP, a GNU library for parsing regular expressions. Like for JEDIT and JMETER, we obtained two successive releases and built separate versions based on the differences between the releases; we created 10 versions of the software. We used the three drivers and the 22 test cases that are provided with the library. The test suite thus contains 66 test cases, which exercise 46% of the methods in the library.

5.2 Studies

To evaluate our technique, we performed two studies; this section presents the results of those studies. For SIENA, we compared each version to the previous versions, whereas for JEDIT, JMETER, and REGEXP, we compared each version to the original version.

Study 1: Test Suite Reduction. The goal of this study was to determine the reduction in the number of test cases that could be achieved using our regression-test-selection technique. For each subject program `P`, and each version `P'`, we used RETEST, shown in Figure 8, to select test cases for regression testing.

Figure 9 shows the results for the four software subjects. For each subject, the figure shows the percentage of test cases that were selected for each version of that subject. The data in the figure illustrate that the reduction in the number of selected test cases varies widely both across and within the subjects. For example, for SIENA, the technique selected less than 2% of the test cases for one version, but between 60% and 90% of the test cases for the remaining six versions. Similarly, for JEDIT, the technique selected fewer than 15% of the test cases for four versions, but over 90% of the test cases for five versions. The extremely low number of test cases selected for some versions, such as version V6

[7]See `http://www.cs.colorado.edu/~hanlee/BIT/`.
[8]See `http://www.sable.mcgill.ca/soot/`.
[9]See `http://bcel.sourceforge.net/`.
[10]Copyright 1999-2001, Apache Software Foundation.

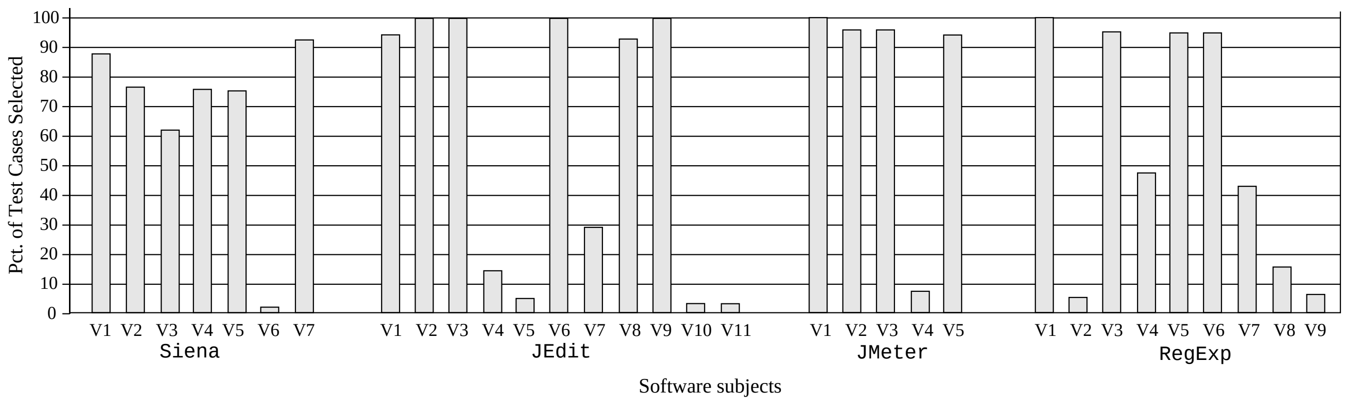


Figure 9: Regression test selection results for our software subjects.

of Siena and version V5, V10, and V11 of JEdit, depends on the fact that, for those versions, the changes are minor, involving few methods, and methods encountered by only a few test cases.

The test reduction illustrated in Figure 9 is similar to results of other studies that have evaluated regression-test-selection techniques for procedural software [12, 30]. The data does not reveal any trends that may be peculiar to the object-oriented paradigm; further experimentation with a bigger and diverse set of subjects will help determine whether such trends exist. The success of code-based regression-test-selection techniques depends not only on the magnitude of changes to the modified software—which, in turn, depends of the frequency of regression testing—but also on the location of those changes and the characteristics of the software. For example, a modification in the startup code of a software causes each test case to be selected. For further example, the characteristics of certain classes of software, such as language parsers, are such that most of the test cases exercise a significant percentage of the code in the software; therefore, for such software, most of the modifications cause a majority of the test cases to be selected. In such cases, code-based regression-test-selection techniques may fail to provide any benefit in terms of reduced regression test suites.

Study 2: Test Selection Granularity. The goal of the second study was to determine whether any additional reduction in the size of the selected test suite could be achieved by selecting dangerous edges instead of dangerous methods. The results of this study will guide our development of profilers in the future.[11]

To perform this study, we used DejaVOO to select dangerous edges in `P`. We then used a data-flow analysis that considers each method M, and determines whether some dangerous edge in M is reached on all paths from the entry point of M. If a dangerous edge is reached over all paths in M, all test cases that enter M will be selected by an edge-level version of Retest. Thus, in this case, no further reduction in the test suite can be achieved by considering changes at the edge level over changes at the method level. However, if there exists some path in M over which a dangerous edge in M is not reached, then, assuming that the test suite covers all edges in `P`, there is at least one test case through M that the edge-level version of Retest will not select. Thus, in this case, additional reduction of the test suite can be achieved by considering regression test selection at the edge level.

The total number of dangerous methods in all four subjects is 51; of these, in 51% of the methods, the dangerous edges are reached along all paths from the entry of the respective methods. Thus, in many cases, performing the test selection at a finer granularity—at the edge level—can reduce the size of the test suite selected. Previous empirical studies that have examined procedural software have also found edge-level selection to differ from method-level selection [5].

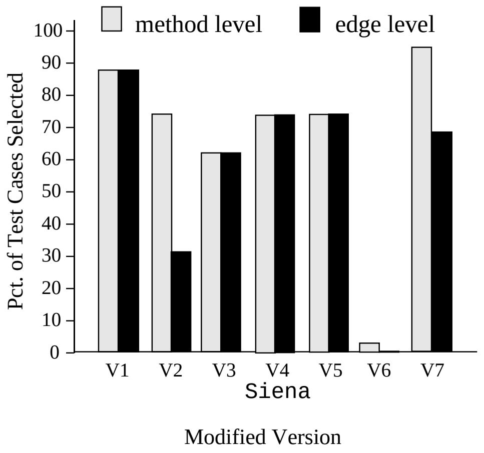


Figure 10: Percentage of test cases that are selected for Siena using method-level and edge-level test selections.

We also wanted to determine the accuracy of our estimates for edge-level test selection. Thus, for Siena, we manually determined the test cases that would be selected by the edge-level version of DejaVOO. The graph in Figure 10 shows the results of this study. For all but three of the

[11]The development of a profiler for recording information at the edge level requires significant effort, and the results of this study can help us decide whether this effort is worthwhile for regression test selection.

versions of Siena, the test suite selected by the method-level version of DejaVOO and the edge-level version of DejaVOO selected the same number of test cases. For those versions that differed—V2, V6, and V7—the graph shows that there can be a significant difference in the size of the reduction. Thus, in cases where running test cases is particularly expensive, the additional overhead in selection required by an edge-level version of DejaVOO may be justified.

6. RELATED WORK

Many researchers have considered the problem of regression test selection for procedural software. A number of regression-test-selection algorithms fit into the general system shown in Figure 2, and are thus related to our work. Ball [3] presents an edge-optimal regression-test-selection algorithm that, under certain conditions, provides more precision than Rothermel and Harrold's algorithm. His algorithm also identifies dangerous edges. Ball also presents additional algorithms based on control flow that are even more precise than edge-based algorithms, at greater computation cost. Vokolos and Frankl present a regression-test-selection algorithm based on text differencing [35]. Their algorithm maintains an association between basic blocks and test cases in `T`, and compares the source files of `P` and `P'` to identify the modified program statements. We can think of this algorithm as selecting dangerous blocks of code in the program. Their algorithm performs the comparison using UNIX `diff` utility, and is based on statements, not control flow; thus, it may select more test cases than the control-flow-based algorithms, at a lesser computation cost. Chen, Rosenblum, and Vo [8] present a regression-test-selection algorithm that detects modified code entities (i.e., dangerous code entities), which are defined as functions or as non-executable components, such as storage locations. The technique selects all test cases associated with changed entities. Because this technique is based on entities that are coarser-grained than those used by statement- or control-flow-based techniques, it may select more test cases than those techniques, with lesser computation cost.

Other researchers have developed regression-test-selection techniques for object-oriented software. In the Introduction, we discussed two techniques: Rothermel, Harrold, and Dedhia's technique for C++ [31] and White and Abdullah's firewall technique [36]. We compared both techniques to our approach: our approach is more precise, can be applied to Java programs, handles exception-handling constructs, can be applied to incomplete programs, and provides a new method for handling polymorphism.

Several techniques have been developed to reduce the effort required to test subclasses (e.g., [7, 11, 13]). These techniques use information associated with a parent class in the design of test suites for derived classes. However, the techniques do not address the problem of regression testing of modified classes.

Kung et al. [17, 18] and Hsia et al. [14] present a technique for selecting regression test cases for class testing. This technique is based on the concept of *firewalls* defined originally by Leung and White for procedural software [21, 20, 37] and later extended to object-oriented software [36]. 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 interactions with C. In such cases, the ORD technique selects test cases that could be omitted from retesting with no ill effects—test cases that our technique does not select. Thus, our technique is more precise than the ORD technique.

7. CONCLUSIONS

In this paper, we have presented the first safe regression-test-selection algorithm that handles the Java language features, that can be applied (under certain conditions) to partial programs, and that is more precise than existing approaches for object-oriented software. We also presented a regression-test-selection system for Java, called Retest, that implements our technique. With Retest, we performed empirical studies to evaluate the effectiveness of our technique. Our empirical studies indicate that the technique can be effective in reducing the size of the test suite but that the reduction varies across subjects and versions. These results are consistent with results reported for C programs, and, for our subject programs, do not show any trends that are peculiar to Java software. Additional studies are required to identify such trends. Our studies also indicate that regression-test-selection at a finer granularity may provide further reductions in the test-suite size. However, additional evaluation is required to determine the granularity of the test selection that provides the best trade-offs in precision and efficiency.

Our future work will include investigating viable alternatives for instrumenting at different levels, gathering additional subjects, and performing empirical studies to evaluate the effectiveness of our technique. We will also perform studies to determine the efficiency of our technique in practice.

8. ACKNOWLEDGMENTS

This work was supported in part by grants to Georgia Tech from Boeing Aerospace Corporation, by National Science Foundation awards CCR-9988294, CCR-0096321, and EIA-0196145, and by the State of Georgia under the Yamacraw Mission. The work was also partially supported by the ESPRIT Project TWO (EP n.28940) and by the Italian Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project. Antonio Carzaniga, David Rosenblum, and Alexander Wolf provided the Siena system and its versions. The anonymous reviewers provided many comments and suggestions that helped improve the presentation of the paper.

9. REFERENCES

[1] Aristotle Research Group, Georgia Institute of Technology. Java Architecture for Bytecode Analysis. 2000.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, Oct. 1996.

[3] T. Ball. On the limit of control flow analysis for regression test selection. In *ACM Int'l Symp. on Softw. Testing and Analysis*, pages 134–142, Mar. 1998.

[4] J. Bible and G. Rothermel. A unifying framework supporting the analysis and development of safe regression test selection techniques. Technical Report 99-60-11, Oregon State University, Dec. 1999.

[5] J. Bible, G. Rothermel, and D. Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection. *ACM Transactions on Software Engineering and Methodology*, 10(2):149–183, Apr. 2001.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.

[7] T. 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 211–222, May 1994.

[9] B. F. Cooper, H. B. Lee, and B. G. Zorn. Profbuilder: A package for rapidly building java execution profilers. Technical report, University of Colorado.

[10] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierachy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[11] S. P. Fielder. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, Apr. 1989.

[12] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the International Conference on Software Engineering*, pages 188–197, Apr. 1998.

[13] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class inheritance structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80, May 1992.

[14] 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.

[15] J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 126–135, Jun. 2000.

[16] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. Firewall regression testing and software maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 1994.

[17] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, Y-S. K im, and Y-K. Song. Developing an object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–87, Oct. 1995.

[18] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Ch en. On regression testing of object-oriented programs. *The Journal of Systems and Software*, 32(1):21–40, Jan. 1996.

[19] D. Kung, J. Gao, P. Hsia, Y. Wen, and Y. Toyoshima. Change impact identification in object-oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance '94*, pages 202–211, Sep. 1994.

[20] 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, Nov. 1990.

[21] H. K. N. Leung and L. J. White. Insights into testing and regression testing global variables. *Journal of Software Maintenance: Research and Practice*, 2:209–222, Dec. 1990.

[22] H. K. N. Leung and L. J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance '91*, pages 201–208, Oct. 1991.

[23] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *Proceedings of the ACM Workshop on Program Analyses for Software Tools and Engineering*, Jun. 2001.

[24] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec. 1989.

[25] Sun Microsystems. Java Virtual Machine Debug Interface. http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/.

[26] Sun Microsystems. Java Virtual Machine Profiler Interface. http://java.sun.com/products/jdk/-1.2/docs/guide/jvmpi/jvmpi.html.

[27] Sun Microsystems. Java2 Platform, API Specification. http://java.sun.com/j2se/1.3/docs/api/.

[28] Sun Microsystems. The Java Foundation Class Abstract Window Toolkit. http://java.sun.com/products/jdk/awt/.

[29] G. Rothermel and M. J. Harrold. A safe, efficient re-

gression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, Apr. 1997.

[30] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, Jun. 1998.

[31] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Journal of Software Testing, Verification, and Reliability*, 10(6):77–109, Jun. 2000.

[32] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, Aug. 1996.

[33] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, pages 849–871, Sep. 2000.

[34] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, Oct. 2000.

[35] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *International Conference on Reliability, Quality, and Safety of Software Intensive Systems*, May 1997.

[36] L. J. White and K. Abdullah. A firewall approach for regression testing of object-oriented software. In *Proceedings of 10th Annual Software Quality Week*, May 1997.

[37] L. J. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 262–270, Nov. 1992.