Scaling Regression Testing to Large Software Systems

Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, Georgia

{orso|clarenan|harrold}@cc.gatech.edu

ABSTRACT

When software is modified, during development and maintenance, it is *regression tested* to provide confidence that the changes did not introduce unexpected errors and that new features behave as expected. One important problem in regression testing is how to select a subset of test cases, from the test suite used for the original version of the software, when testing a modified version of the software. Regression-test-selection techniques address this problem. Safe regression-test-selection techniques select every test case in the test suite that may behave differently in the original and modified versions of the software. Among existing safe regression testing techniques, efficient techniques are often too imprecise and achieve little savings in testing effort, whereas precise techniques are too expensive when used on large systems. This paper presents a new regression-test-selection technique for Java programs that is safe, precise, and yet scales to large systems. It also presents a tool that implements the technique and studies performed on a set of subjects ranging from 70 to over 500 KLOC. The studies show that our technique can efficiently reduce the regression testing effort and, thus, achieve considerable savings.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*;

General Terms: Algorithms, Experimentation, Verification

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

1. INTRODUCTION

As software evolves, *regression testing* is applied to modified versions of the software to provide confidence that the changed parts behave as intended and that the changes did not introduce unexpected faults, also known as *regression faults*. In the typical regression testing scenario, D is the developer of a software product P, whose latest version has been tested using a test suite T and then released. During maintenance, D modifies P to add new features and to fix faults. After performing the changes, D obtains a new version of the software product, P', and needs to regression test it before committing the changes to a repository or before release.

One important problem that D must face is how to select an appropriate subset T' of T to rerun on P'. This process is called *regression test selection* (RTS hereafter). A simple approach to RTS is to rerun all test cases in T on P', that is, to select $T' = T$. However, rerunning all test cases in T can be expensive and, when there are limited changes between P and P', may involve unnecessary effort. Therefore, RTS techniques (e.g., [2, 4, 6, 7, 9, 15, 17, 20, 22, 23]) use information about P, P', and T to select a subset of T with which to test P', thus reducing the testing effort. An important property for RTS techniques is safety. A *safe* RTS technique selects, under certain assumptions, every test case in the test suite that may behave differently in the original and modified versions of the software [17]. Safety is important for RTS techniques because it guarantees that T' will contain all test cases that may reveal regression faults in P'. In this paper, we are interested in safe RTS techniques only.

Safe RTS techniques (e.g., [6, 7, 15, 17, 20, 22] differ in efficiency and precision. Efficiency and precision, for an RTS technique, are generally related to the level of granularity at which the technique operates [3]. Techniques that work at a high level of abstraction—for instance, by analyzing change and coverage information at the method or class level—are more efficient, but generally select more test cases (i.e., they are less precise) than their counterparts that operate at a fine-grained level of abstraction. Conversely, techniques that work at a fine-grained level of abstraction—for instance, by analyzing change and coverage information at the statement level—are precise, but often sacrifice efficiency.

In general, for an RTS technique to be cost-effective, the cost of performing the selection plus the cost of rerunning the selected subset of test cases must be less than the overall cost of rerunning the complete test suite [23]. For regression-testing cost models (e.g., [10, 11]), the meaning of the term *cost* depends on the specific scenario considered. For example, for a test suite that requires human intervention (e.g., to check the outcome of the test cases or to setup some machinery), the savings must account for the human effort that is saved. For another example, in a cooperative environment, in which developers run an automated regression test suite before committing their changes to a repository, reducing the number of test cases to rerun may result in early availability of updated code and improve the efficiency of the development process. Although empirical studies show that existing safe RTS techniques can be cost-effective [6, 7, 19],

SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

such studies were performed using subjects of limited size. In our preliminary studies we found that, in many cases, existing safe techniques are not cost-effective when applied to large software systems: efficient techniques tend to be too imprecise and often achieve little or no savings in testing effort; precise techniques are generally too expensive to be used on large systems. For example, for one of the subjects studied, it took longer to perform RTS than to run the whole test suite.

In this paper, we present a new RTS algorithm for Java programs that handles the object-oriented features of the language, is safe and precise, and still scales to large systems. The algorithm consists of two phases: partitioning and selection. The partitioning phase builds a high-level graph representation of programs P and P' and performs a quick analysis of the graphs. The goal of the analysis is to identify, based on information on changed classes and interfaces, the parts of P and P' to be further analyzed. The selection phase of the algorithm builds a more detailed graph representation of the identified parts of P and P', analyzes the graphs to identify differences between the programs, and selects for rerun test cases in T that traverse the changes. Although the technique is defined for the Java language, it can be adapted to work with other object-oriented (and traditional procedural) languages.

We also present DejaVOO: a tool that we developed and that implements our RTS technique.

Finally, we present a set of empirical studies performed using DejaVOO on a set of Java subjects ranging from 70 to over 500 KLOC. The studies show that, for the subjects considered, our technique is considerably more efficient than an existing precise technique that operates at a fine-grained level of abstraction (89% on average for the largest subject). The studies also show that the selection achieves considerable savings in overall regression testing time. For the three subjects, our technique saved, on average, 19%, 36%, and 63% of the regression testing time.

The main contributions of this paper are:

- The definition of a new technique for RTS that is effective and can scale to large systems.
- The description of a prototype tool, DejaVOO, that implements the technique.
- A set of empirical studies that show and discuss the effectiveness and efficiency of our technique. These are the first studies that apply safe RTS to real systems of these sizes.

2. TWO-PHASE RTS TECHNIQUE

The basic idea behind our technique is to combine the effectiveness of RTS techniques that are precise but may be inefficient on large systems (e.g., [7, 17, 18]) with the efficiency of techniques that work at a higher-level of abstraction and may, thus, be imprecise (e.g., [9, 22]). We do this using a two-phase approach that performs (1) an initial high-level analysis, which identifies parts of the system to be further analyzed, and (2) an in-depth analysis of the identified parts, which selects the test cases in T to rerun.

We call these two phases partitioning and selection. In the *partitioning* phase, the technique analyzes the program to identify hierarchical, aggregation, and use relationships among classes and interfaces [13]. Then, the technique uses the information about these relationships, together with information about which classes and interfaces have syntactically changed, to identify the parts of the program that may be affected by the changes between P and P'. (Without loss of generality, we assume information about syntactically changed classes and interfaces to be available. This information can be easily gathered, for example, from configuration-management systems, IDEs that use versioning, or by comparison of the two versions of the program.) The output of this phase is a subset of the classes and interfaces in the program (hereafter, referred to as the *partition*). In the *selection* phase, the technique takes as input the partition identified by the first phase and performs edge-level test selection on the classes and interfaces in the partition. Edge-level test selection selects test cases by analyzing change and coverage information at the level of the flow of control between statements (see Section 2.2 for details). To perform edge-level selection, we leverage an approach previously defined by some of the authors [7].

Because of the partitioning performed in the first phase, the low-level, expensive analysis is generally performed on only a small fraction of the whole program. Although only a part of the program is analyzed, the approach is still—under certain assumptions—safe because (1) the partitioning identifies all classes and interfaces whose behavior may change as a consequence of the modifications to P, and (2) the edge-level technique we use on the selection phase is safe [7]. The assumptions for safety are discussed in Section 2.3.

In the next two sections, we illustrate the two phases in detail, using the example provided in Figures 1 and 2. The example consists of a program P (Figure 1) and a modified version of P, P' (Figure 2). The differences between the two programs are highlighted in the figures. Note that, for ease of presentation, we align and use the same line numbering for corresponding lines of code in P and P'. Also for ease of presentation, in the rest of the paper we use the term *type* to indicate both classes and interfaces and the terms *super-type* and *sub-type* to refer to type-hierarchical relation (involving two classes, a class and an interface, or two interfaces).

2.1 Partitioning

The first phase of the approach performs a high-level analysis of P and P' to identify the parts of the program that may be affected by the changes to P. The analysis is based on purely syntactic changes between P and P' and on the relationships among classes and interfaces in the program.

2.1.1 Accounting for Syntactic Changes

Without loss of generality, we classify program changes into two groups: statement-level changes and declaration-level changes. A *statement-level change* consists of the modification, addition, or deletion of an executable statement. These changes are easily handled by RTS: each test case that traverses the modified part of the code must be re-executed. Figures 1 and 2, at line 9, show an example of statement-level change from P to P'. Any execution that exercises that statement, will behave differently for P and P'.

A *declaration-level change* consists of the modification of a declaration. Examples of such changes are the modification of the type of a variable, the addition or removal of a method, the modification of an inheritance relationship, the change of type in a `catch` clause, or the change of a modifiers list (e.g., the addition of modifier "synchronized" to a method). These changes are more problematic for RTS than statement-level changes because they affect the behavior of

Program P

```
1:  public class SuperA {
2:     int i=0;
3:     public void foo() {
4:        System.out.println(i);
5:     }
6:  }
7:  public class A extends SuperA {
8:     public void dummy() {
9:        i--;
10:       System.out.println(-i);
11:    }
12: }
13: public class SubA extends A {}
14: public class B {
15:    public void bar() {
16:       SuperA a=LibClass.getAnyA();
17:       a.foo();
18:    }
19: }
20: public class SubB extends B {}
21: public class C {
22:    public void bar(B b) {
23:       b.bar();
24:    }
25: }
```

Figure 1: Example program P.

Program P'

```
1:  public class SuperA {
2:     int i=0;
3:     public void foo() {
4:        System.out.println(i);
5:     }
6:  }
7:  public class A extends SuperA {
8:     public void dummy() {
9:        i++;
10:       System.out.println(-i);
11:    }
12a:   public void foo() {
12b:      System.out.println(i+1);
12c:   }
12d:}
13: public class SubA extends A {}
14: public class B {
15:    public void bar() {
16:       SuperA a=LibClass.getAnyA();
17:       a.foo();
18:    }
19: }
20: public class SubB extends B {}
21: public class C {
22:    public void bar(B b) {
23:       b.bar();
24:    }
25: }
```

Figure 2: Modified version of program P, P'.

the program only indirectly, often in non-obvious ways. Figures 1 and 2 show an example of a declaration-level change: a new method foo is added to class A, in P' (lines 12a–12c). In this case, the change indirectly affects the statement that calls $a.foo$ (line 17). Assume that *LibClass* is a class in the library and that $LibClass.getAnyA()$ is a static method of such class that returns an instance of $SuperA$, A, or $SubA$. After the static call at line 16, the dynamic type of a can be $SuperA$, A, or $SubA$. Therefore, due to dynamic binding, the subsequent call to $a.foo$ can be bound to different methods in P and in P': in P, $a.foo$ is always bound to $SuperA.foo$, whereas in P', $a.foo$ can be bound to $A.foo$ or $SuperA.foo$, depending on the dynamic type of a. This difference in binding may cause test cases that traverse the statement at line 17 in P to behave differently in P'.

Declaration-level changes have generally more complex effects than statement-level changes and, if not suitably handled, can cause an RTS technique to be imprecise, unsafe, or both. We will show how our technique suitably handles declaration-level changes in both phases.

2.1.2 Accounting for Relationships Between Classes

In this section, we use the example in Figures 1 and 2 to describe, intuitively, how our partitioning algorithm works and the rationale behind it. To this end, we illustrate different alternative approaches to partitioning, discuss their shortcomings, and motivate our approach.

One straightforward approach for partitioning based on changes is to select just the changed types (classes or interfaces). Assume that the change at line 9 in Figures 1 and 2 is the only change between P and P'. In this case, any test case that behaves differently when run on P and P' must necessarily traverse statement 9 in P. Therefore, the straightforward approach, which selects only class A, would be safe and precise: in the second phase, the edge-level analysis of class A in P and P' would identify the change at statement 8 and select all and only test cases traversing it.

However, such a straightforward approach does not work in general for declaration-level changes. Assume now that the only change between P and P' is the addition of method foo to A (12a–12c in P'). As we discussed above, this change leads to a possibly different behavior for test cases that traverse statement 17 in P, which belongs to class B. Therefore, all such test cases must be included in T', the set of test cases to rerun on P'. Conversely, any test case that does not execute that statement can be safely excluded from T'.

Unfortunately, the straightforward approach would still select only class A. The edge-level analysis of A would then show that the change between A in P and A in P' is the addition of the overriding method $A.foo$, a declaration-level change that does not affect directly any other statement in A. Therefore, the only way to select test cases that may be affected by the change would be to select all test cases that instantiate class A[1] because these test cases may execute $A.foo$ in P'. Such an approach is clearly imprecise: some test cases may instantiate class A and never traverse $a.foo$, but the approach would still select them. Moreover, this selection is also unsafe. If, in the second-phase, we analyze only class A, we will miss the fact that class $SubA$ inherits from A. Without this information, we will not select test cases that traverse $a.foo$ when the dynamic type of a is $SubA$. Because these test cases may also behave differently in P and P', not selecting them is unsafe.

Because polymorphism and dynamic binding make RTS performed only on the changed types both unsafe and imprecise, a possible improvement is to select, when a type C has changed, the whole type hierarchy that involves C (i.e., all super- and sub-types of C. Considering our example, this strategy will select a partition that contains classes $SuperA$, A, and $SubA$. By analyzing $SuperA$, A, and $SubA$, the edge-level technique would (1) identify the inheritance

[1]In this case, static calls are non-relevant because dynamic binding can occur only on actual instances.

```
algorithm buildIRG
input: program P
output: IRG G for P

begin buildIRG
 1: create empty IRG G
 2: for each class and interface e ∈ P do
 3:    create node n_e
 4:    G_N = G_N ∪ {n_e}
 5: end for
 6: for each class and interface e ∈ P do
 7:    get direct super-type of e, s
 8:    G_IE = G_IE ∪ {⟨n_e, n_s⟩}
 9:    for each type r ∈ P that e references do
10:       G_UE = G_UE ∪ {⟨n_e, n_r⟩}
11:    end for
12: end for
13: return G
end buildIRG
```

Figure 3: Algorithm for building an IRG.

relationships correctly, (2) discover that a call to $a.foo$ may be bound to different methods in P and P' if the type of a is A or $SubA$, and (3) consequently select for rerun all test cases that instantiate A, $SubA$, or both. Thus, such a partitioning would lead to safe results.

Although considering whole hierarchies solves the safety issue, the approach is still imprecise. Again, some test cases may instantiate class A or $SubA$ and never invoke $a.foo$. Such test cases behave identically in P and P', but they would still be selected for rerun.

To improve the precision of the selection, our partitioning technique considers, in addition to the whole hierarchy that involves a changed type, all types that explicitly reference types in the hierarchy. For our example, the partition would include $SuperA$, A, $SubA$, and B. By analyzing these four classes, an edge-level selection technique would be able to compute the hierarchy relationships, as discussed above, and also to identify the call site to $a.foo$ in $B.bar$ as the point in the program where the program's behavior may be affected by the change (details on how this is actually done are provided in Section 2.2). Therefore, the edge-level technique can select all and only the test cases that call $a.foo$ when the dynamic type of a is either A or $SubA$. This selection is safe and as precise as the most precise existing RTS techniques (e.g., [2, 7, 17]).

It is important to note that no other type, besides the ones in the partition, must be analyzed by the edge-level technique. Because of the way the system is partitioned, any test case that behaves differently in P and P' must necessarily traverse one or more types in the partition, and would therefore be selected. Consider, for instance, class C of our example, which is not included in the partition. If a test case that exercises class C shows a different behavior in P and P', it can only be because of the call to $B.bar$ in $C.bar$. Therefore, the test case would be selected even if we consider only B.

In summary, our partitioning technique selects, for each changed type (class or interface), (1) the type itself, (2) the hierarchy of the changed type, and (3) the types that explicitly reference any type in such hierarchy. Note that it may be possible to reduce the size of the partition identified by the algorithm by performing additional analysis (e.g., by distinguishing different kinds of use relationships among classes). However, doing so would increase the cost of the partitioning and, as the studies in Section 3 show, our current approach is effective in practice. In the next section, we present the algorithm that performs the selection.

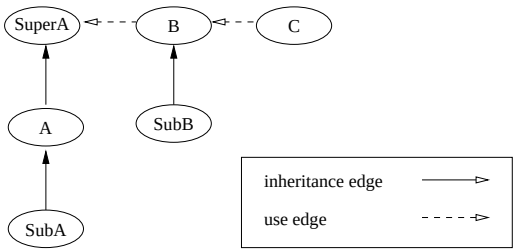


Figure 4: IRG for program P of Figure 1.

2.1.3 Partitioning Algorithm

Before describing the details of the partitioning algorithm, we introduce the program representation on which the algorithm operates: the interclass relation graph. The *Interclass Relation Graph (IRG)* for a program is a triple $\{N, IE, UE\}$:

- N is the set of nodes, one for each type.
- $IE \subset N \times N$ is the set of inheritance edges. An *inheritance* edge between a node for type e_1 and a node for type e_2 indicates that e_1 is a direct sub-type of e_2.
- $UE \subset N \times N$ is the set of use edges. A *use* edge between a node for type e_1 and a node for type e_2 indicates that e_1 contains an explicit reference to e_2.[2]

Figure 3 shows the algorithm for building an IRG, `buildIRG`. For simplicity, in defining the algorithms, we use the following syntax: n_e indicates the node for a type e (class or interface); G_N, G_{IE}, and G_{UE} indicate the set of nodes N, inheritance edges IE, and use edges UE for a graph G, respectively.

Algorithm `buildIRG` first creates a node for each type in the program (lines 2–5). Then, for each type e, the algorithm (1) connects n_e to the node of its direct super-type through an inheritance edge (lines 7–8), and (2) creates a use edge from each n_d to n_e, such that d contains a reference to e (lines 9–11). Figure 4 shows the IRG for program P in Figure 1. The IRG represents the six classes in P and their inheritance and use relationships.

Figures 5, 6, and 7 show our partition algorithm, `computePartition`. The algorithm inputs the set of syntactically-changed types C and two IRGs, one for the original version of the program and one for the modified version of the program. As we mentioned above, change information can be easily computed automatically.

First, algorithm `computePartition` adds to partition $Part$ all hierarchies that involve changed types. For each type e in the changed-type set C, the algorithm adds to $Part$ e itself (line 3) and all sub- and super-types of e, by calling procedures `addSubTypes` and `addSuperTypes` (lines 4 and 5). If s, the super-type of e in P, and s', the super-type of e in P', differ (lines 6–8), the algorithm also adds all super-types of s' to the partition (line 9). This operation is performed to account for cases in which type e is moved to another inheritance hierarchy. In these cases, changes to e may affect not only types in e's old hierarchy, but also types in e's new hierarchy. Consider our example program in Figure 1 and its corresponding IRG in Figure 4. Because the only changed type is class A, at this point in the algorithm $Part$ would contain classes A, $SubA$, and $SuperA$.

[2]The fact that e_1 contains an explicit reference to e_2 means that e_1 *uses* e_2 (e.g., by using e_2 in a cast operation, by invoking one of e_2's methods, by referencing one of e_2's field, or by using e_2 as an argument to `instanceof`).

```
algorithm computePartition
input: set of changed types C,
       IRG for P, G
       IRG for P', G'
declare: set of types Tmp, initially empty
output: set of types in the partition, Part

begin computePartition
 1: Part = ∅
 2: for each type e ∈ C do
 3:    Part = Part ∪ {e}
 4:    Part = Part ∪ addSubTypes(G, e)
 5:    Part = Part ∪ addSuperTypes(G, e)
 6:    n_s = n ∈ G, ⟨n_e, n⟩ ∈ G_IE
 7:    n_s' = n ∈ G', ⟨n_e, n⟩ ∈ G'_IE
 8:    if s ≠ s' then
 9:       Part = Part ∪ {s'}
10:       Part = Part ∪ addSuperTypes(G', s')
11:    end if
12: end for
13: for each type p ∈ Part do
14:    for each edge v = ⟨n_d, n_p⟩ ∈ G_UE do
15:       Tmp = Tmp ∪ {d}
16:    end for
17: end for
18: Part = Part ∪ Tmp
19: return Part
end computePartition
```

Figure 5: Algorithm `computePartition`.

Second, for each type p currently in the partition, `computePartition` adds to a temporary set Tmp all types that reference p directly (lines 13–17). In our example, this part of the algorithm would add class B to T.

Finally, the algorithm adds the types in Tmp to $Part$ and returns $Part$.

Procedure `addSubTypes` inputs an IRG G and a type e, and returns the set of all types that are sub-types of e. The procedure performs, through recursion, a backward traversal of all inheritance edges whose target is node n_e. Procedure `addSuperTypes` also inputs an IRG G and a type e, and returns the set of all types that are super-types of e. The procedure identifies super-types through reachability over inheritance edges, starting from e.

Complexity. Algorithm `buildIRG` makes a single pass over each type t, to identify t's direct super-class and classes referenced by t. Therefore, the worst-case time complexity of the algorithm is $O(m)$, where m is the size of program P. Algorithm `computePartition` performs reachability from each changed type. The worst case time complexity of the algorithm is, thus, $|C|O(n^2)$, where C is the set of changed types and n is the number of nodes in the IRG for P (i.e., the number of classes and interfaces in the program). However, this complexity corresponds to the degenerate case of programs with n types and an inheritance tree of depth $O(n)$. In practice, the depth of the inheritance tree can be approximated with a constant, and the overall complexity of the partitioning is linear in the size of the program. In fact, our experimental results show that our partition algorithm is very efficient in practice; for the largest of our subjects, JBoss, which contains over 2,400 classes and over 500 KLOC, our partitioning process took less than 15 seconds to complete.

2.2 Selection

The second phase of the technique (1) computes change information by analyzing the types in the partition identified by the first phase, and (2) performs test selection by matching the computed change information with coverage information. To perform edge-level selection, we use an approach previously defined by some of the authors [7]. In the following we provide an overview of this part of the technique. Reference [7] provides additional details.

```
procedure addSubTypes
input: IRG G,
       type e
output: set of all sub-types of e, S
begin addSubTypes
19: S = ∅
20: for each node n_s ∈ G, ⟨n_s, n_e⟩ ∈ G_IE do
21:    S = S ∪ {s}
22:    S = S ∪ addSubTypes(G, s)
23: end for
24: return S
end addSubTypes
```

Figure 6: Procedure `addSubTypes`.

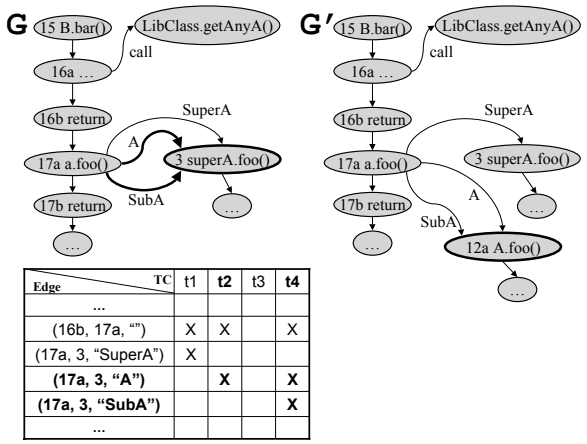
```
procedure addSuperTypes
input: IRG G,
       type e
output: set of all super-types of e, S
begin addSuperTypes
25: S = ∅
26: while ∃n_s ∈ G, ⟨n_e, n_s⟩ ∈ G_IE do
27:    S = S ∪ {s}
28:    e = s
29: end while
30: return S
end addSuperTypes
```

Figure 7: Procedure `addSuperTypes`.

2.2.1 Computing Change Information

To compute change information, our technique first constructs two graphs, G and G', that represent the parts of P and P' in the partition identified by the partitioning phase. To adequately handle all Java language constructs, we defined a new representation: the Java Interclass Graph. A *Java Interclass Graph (JIG)* extends a traditional control flow graph, in which nodes represent program statements and edges represent the flow of control between statements. The extensions account for various aspects of the Java language, such as inheritance, dynamic binding, exception handling, and synchronization. For example, all occurrences of class and interface names in the graph are fully qualified, which accounts for possible changes of behavior due to changes in the type hierarchy. The extensions also allow for safely analyzing subsystems if the part of the system that is not analyzed is unchanged (and unaffected by the changes), as described in detail in Reference [7].

For the sake of space, instead of discussing the JIG in detail, we illustrate how we model dynamic binding in the JIG using our example programs P and P' (see Figures 1 and 2). The top part of Figure 8 shows two partial JIGs, G and G', that represent method $B.bar$ in P and P'. In the graph, each call site (lines 16 and 17) is expanded into a *call* and a *return* node (nodes 16a and 16b, and 17a and 17b). Call and return nodes are connected by a *path edge* (edges (16a,16b) and (17a,17b)), which represents the path through the called method. Call nodes are connected to the entry node(s) of the called method(s) with a *call edge*. If the call is static (i.e., not virtual), such as the call to $LibClass.getAnyA()$, the call node has only one outgoing call edge, labeled *call*. 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 considered, the call to $a.foo$ at node 17a is a virtual call. In P, although a can have dynamic type $SuperA$, A, or $SubA$, the call is



Edge \ TC	t1	t2	t3	t4
...				
(16b, 17a, "")	X	X		X
(17a, 3, "SuperA")	X			
(17a, 3, "A")		**X**		**X**
(17a, 3, "SubA")				**X**
...				

Figure 8: Edge-level regression test selection.

always bound to method $SuperA.foo$ (there is only one implementation of method foo, in $SuperA$, which is inherited by A and $SubA$). Therefore, G contains three call edges from node 17a to the entry of $SuperA.foo$, labeled $SuperA$, A, and $SubA$. In P', if the dynamic type of a is $SuperA$, the call is bound to $SuperA.foo$, whereas if the dynamic type of a is A or $SubA$, the call is bound to $A.foo$. Therefore, G' contains one call edge from node 17a to the entry of $SuperA.foo$, labeled $SuperA$, and two call edges from node 17a to the entry of $A.foo$, labeled A and $SubA$.

After constructing the JIGs, the technique traverses them to identify *dangerous edges*: edges that, if traversed, may lead to a different behavior in P and P'. For example, for P and P' in Figures 1 and 2, the edge that represents the flow of control from the statement at line 8 to the statement at line 9 is a dangerous edge. Any input that causes P to traverse that edge may—in this specific case, will—cause a different behavior in P and in P'. Similar to Rothermel and Harrold's technique [17], our technique identifies dangerous edges by performing a synchronous traversal of G and G'. Whenever the targets of like-labeled edges in G and G' differ, the edge is added to the set of dangerous edges.

To illustrate, consider again G and G' in Figure 8. The algorithm that performs the synchronous traversal of the graphs starts from node 15 in G and G', matches the edges from 15 to 16a, and compares their target nodes. Because the two nodes match (i.e., they represent identical instructions), the traversal continues from these nodes. The algorithm continues through like paths in the two graphs by traversing like-labeled edges until a difference in the target nodes of such edges is detected. Therefore, the algorithm successfully matches the three pairs of nodes, in G and G', labeled $LibClass.getAnyA()$, 16b, and 17a. At this point, assuming that the next edge considered is the outgoing call edge with label $SubA$ (edge (17a,3,"SubA")), the algorithm would compare the target nodes for such edge in G and G' and find that they differ: in G, the target node is $SuperA.foo()$, whereas in G', the target node is $A.foo()$. The algorithm would thus add this edge to the set of dangerous edges. For analogous reasons, the algorithm would add edge (17a,3,"A") to the set of dangerous edges. Conversely, edge (17a,3,"SuperA") is not dangerous because its target node, both in G and in G', has not changed. Subsequent traversals find no additional dangerous edges and, at the end of the synchronous walk, the set of dangerous edges for P consists of edges (17a,3,"SubA") and (17a,3, "A").

In the next section, we describe how dangerous edges are matched with coverage information to select for rerun all test cases in T that traverse edge (17a,3,"SubA") or edge (17a,3,"A"), that is, all test cases that execute the call at statement 17 with a's dynamic type being A or $SubA$.

2.2.2 Performing Test Selection

When testing a program P, testers measure the coverage achieved by a test suite T to assess the adequacy of T. Coverage is usually computed for program entities, such as statements or edges. For each test case t in T, information is recorded about which entities in P are executed by t. Such coverage information can be automatically collected using one of the many coverage tools available and can be conveniently represented as a *coverage matrix*, with one row per entity and one column per test case. A mark in a cell (i, j) represents the fact that test case t_j covers entity e_i. The bottom part of Figure 8 shows a partial coverage matrix for program P, whose JIG G is shown in the upper part of the same figure. The matrix shows that test cases $t1$, $t2$, and $t4$ cover edge (16b,17a,""), $t1$ covers (17a,3,"SuperA"), $t2$ and $t4$ cover (17a,3,"A"), and $t4$ covers (17a,3,"SubA"). Test case $t3$ does not cover any of the edges shown.

In our approach, we use coverage at the edge-level because that is the level at which we also compute change information, as described in the previous section. Given the set of dangerous edges and the coverage matrix for program P and test suite T, our technique performs a simple lookup and selects for rerun all test cases that traverse at least one dangerous edge. For the example in Figure 8, where the dangerous edges are (17a,3,"A") and (17a,3,"SubA"), our approach would select test cases $t2$ and $t4$. It is worth noting that the coverage for the test cases that do not traverse any dangerous edge does not have to be recomputed: it does not change because such test cases execute exactly in the same way in both P and P'. In the case in which relative positions in the code change, coverage can be mapped from P to P' based on change information [1, 21], or it can be recomputed when free cycles are available.

Two things are worth noting about this part of the technique. First, coverage information can be efficiently gathered and many testers gather it anyway, so we are imposing little or no extra overhead on the testing process. Second, the approach does not necessarily require this specific kind of coverage and could be adapted to work with coverage of other entities, such as statements.

2.3 Assumptions For Safety

To be safe, 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. Two overall assumptions are that the code must be compilable (i.e., that the programs we analyze are syntactically correct) and that the test cases in T can be rerun individually (otherwise, there would be no reason to perform regression-test selection). The other assumptions are related to the determinism of test-case execution, the execution environment, and the use of reflection within the code.

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 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 and, thus, allows for safely excluding test cases that do not traverse modifications. Note that this assumption does not involve all multithreaded programs, but only those programs for which the interaction among threads affects the coverage and the outputs. In such cases, for our technique to be applicable, we must use special execution environments that guarantee the deterministic order in which the instructions in different threads are executed [12].

Execution environment. Changes in the execution environment could modify the program behavior in ways that our technique would not be able to identify. Therefore, we assume that the execution environment is the same across versions. In particular, we assume that the same Java Virtual Machine is used and that there are no changes in the libraries used by the program under test.

Use of reflection. In Java, reflection provides runtime access to information about classes' fields and methods, and allows for using such fields and methods to operate on objects. Reflection can affect the safety of regression-test-selection techniques in many ways. For example, using reflection, a method may be invoked on an object without performing a traditional method call on that object. For another example, a program may contain a predicate whose truth value depends on the number of fields in a given class; in such a case, the control flow in the program may be affected by the (apparently harmless) addition of unused fields to that class. Although some uses of reflections can be handled through analysis, others require additional, user-provided information. In our work, we assume that such information is available and can be leveraged for the analysis. In particular, in the case of dynamic class loading, we assume that the classes that can be loaded (and instantiated) by name at a specific program point either can be inferred from the code (in the case in which a `cast` operator is used on the instance after the object creation) or are specified by the user.

3. EMPIRICAL EVALUATION

Our goal is to empirically investigate effectiveness and efficiency of the technique presented in this paper when applied to medium and large systems. To this end, we developed a prototype tool, DejaVOO, that implements the technique, and used it to perform an empirical study on a set of subjects. The study investigates three research questions:

RQ1: What percentage of the program under test does our partitioning technique select, and how does this affect the overall RTS costs?

RQ2: How much do we gain, in terms of precision, with respect to a technique that operates only at a high-level of abstraction?

RQ3: What overall savings can our technique achieve in the regression testing process?

3.1 The Tool: DejaVOO

The diagram in Figure 9 shows the high-level architecture of DejaVOO, which consists of three main components: InsECT, DEI, and Selector.

InsECT *(Instrumentation, Execution, and Coverage Tool)* is a modular, extensible, and customizable instrumenter and

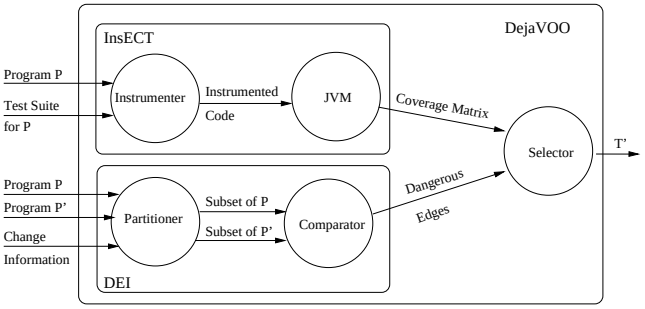


Figure 9: High-level architecture of the DejaVOO regression-test-selection system

coverage analyzer that we developed in Java [5]. Within DejaVOO, we use InsECT to gather edge-coverage information for program P when run against its test suite T.

DEI *(Dangerous Edges Identifier)* and Selector implement the two-phase technique presented in this paper. The *Partitioner* performs Phase 1 of the technique: it inputs program P, program P', and information about changed types, and computes the partitions of P and P' to be further analyzed. We make no assumptions on how the change information is gathered, as long as it is expressed in terms of syntactically changed types (classes and interfaces).

To identify inheritance and use relations between types, the *Partitioner* uses the Byte Code Engineering Library (BCEL)[3] to inspect classes' and interfaces' bytecode. The *Partitioner* also takes advantage of externally provided information (through a configuration file) for some uses of reflection, such as calls to $java.lang.Class.newInstance()$ or calls to several classes in the $java.lang.reflect$ package. If such information is not provided for a point in the program where the use of reflection requires it, the *Partitioner* issues a warning and reports the point in the program for which the information is missing.

The *Comparator* performs the first part of Phase 2 of the technique: it builds the JIGs for the parts of P and P' to be analyzed, compares them, and identifies dangerous edges.

The *Selector* implements the final part of Phase 2. It performs a simple lookup in the coverage matrix to select all test cases that traverse any dangerous edge and reports the list of such test cases.

3.2 Variables and Measures

3.2.1 Independent Variables

The independent variable in this study is the particular RTS technique used. We considered four techniques.

RetestAll. This technique simply selects all test cases in T for rerun on P'. This is our control technique.

HighLevel. This technique identifies changes at the class and interface level and selects all test cases that instantiate changed classes or classes that may be affected by the changes. We use this technique as a representative of the RTS approaches based on an efficient analysis performed at a high level of abstraction, such as the firewall-based techniques [9, 22].

EdgeLevel. This technique identifies changes at the edge level and selects test cases based on coverage information at the same level. This technique is analogous to Phase 2 of our approach performed on the whole program. We use this

[3] `http://jakarta.apache.org/bcel/`

technique as a representative of the RTS approaches based on a precise and expensive analysis (e.g., [7, 18]).

TwoPhases. This is the two-phase technique presented in this paper.

3.2.2 Dependent Variable and Measures

Our dependent variables of interest are (1) effectiveness of the technique in terms of savings in testing effort, and (2) technique efficiency in terms of overhead imposed on the testing process. Because the techniques we examine are safe, and thus select all test cases that may reveal regression faults, fault detection capability of the selected test cases is not a variable of interest. We use two measures for technique effectiveness: reduction in test-suite size and reduction in test-execution time. We use one measure for technique efficiency: analysis time.

Reduction in test-suite size. One method used to compare RTS techniques considers the degrees to which the techniques reduce test-suite size for given modified versions of a program. Using this method, for each RTS technique R that we consider and for each (version, subsequent-version) pair (P_i,P_{i+1}) of program P, where P_i is tested by test suite T, we measure the percentage of T selected by R to test P_{i+1}.

Reduction in test-execution time. To further evaluate savings, for each RTS technique, we measure the time required to execute the selected subset of T (i.e., T') on P_{i+1}. As discussed in the introduction, other cost factors may be relevant when assessing the savings achieved by a regression testing technique (e.g., the amount of human effort saved). However, we limit ourselves to the reduction in test-suite size and in test-execution time because they are good indicators and, most important, can be measured accurately.

Analysis time. One way to measure the overall savings OS achieved by an RTS technique when applied to version P_{i+1} of a program is given by the formula

$$OS = TimeT - TimeT' - TimeA$$

where $TimeT$ is the time to execute the entire test suite on P_{i+1}, $TimeT'$ is the time to execute the selected subset of T on P_{i+1}, and $TimeA$ is the time to perform RTS, or analysis time [10]. Analysis time is thus an important indicator of the efficiency of an RTS technique. To compare efficiency of different techniques, for each RTS technique, we measure the time required to perform RTS on P_{i+1}.

3.3 Experiment Setup

3.3.1 Subject programs

As subjects for our study we utilized several releases of each of three medium-to-large programs: JABA, DAIKON, and JBOSS, summarized in Table 1. The table shows, for each subject, the size as number of non-commented lines of code ($KLOC$), the number of classes and interfaces ($Types$), the number of versions ($Versions$), the number of test cases (TC), and the time required to run the entire regression test suite ($Time\ TS$). All values, except the number of versions, are computed, for each program, as averages across versions.

JABA[4] (Java Architecture for Bytecode Analysis) is a framework for analyzing Java programs developed within the Aristotle Research Group at Georgia Tech. DAIKON[5] is a tool that performs dynamic invariant detection. JBOSS[6], the largest of our subjects, is a fully-featured Java application server. For all three programs, we extracted from their CVS repositories five consecutive versions from one to a few days apart. By doing so, we simulate a possible way in which regression testing would occur in practice, before any commit to the repository. For all three programs, we used the test suites used internally by the programs' developers, which were also available through CVS.

Subject	*KLOC*	*Types*	*Versions*	*TC*	*Time TS*
Jaba	70	525	5	707	54 mins
Daikon	167	824	5	200	74 mins
JBoss	532	2,403	5	639	32 mins

Table 1: Subjects programs for the empirical study.

3.3.2 Experiment Design

To implement technique HighLevel, we modified DEJAVOO so that, after identifying the partition in Phase 1, it skips Phase 2 and instead selects all test cases through the partition. As an implementation of EdgeLevel, we used a prototype tool, created at Georgia Tech, that implements the RTS technique presented by some of the authors in Reference [7]. Finally, as an implementation of TwoPhases, we used DEJAVOO, discussed above.

For each version P_i, we ran the entire regression test suite for P_i and measured the time elapsed. This corresponds to our control technique, RetestAll. We also performed a second run of each version to collect coverage information. (We measured time and coverage on different runs to avoid for the instrumentation overhead to affect the time measurements.) Then, for each (program, modified-version) pair (P_i, P_{i+1}), we performed RTS using the three techniques considered. For each technique, we measured selected test-suite size, selected test-suite execution time, and analysis time, as discussed in Section 3.2.2. These measures served as the data sets for our analysis. We collected all data on a dedicated 2.80 GHz Pentium4 PC, with 2 GB of memory, running GNU/Linux 2.4.23.

3.4 Threats to Validity

Like any empirical study, this study has limitations that must be considered when interpreting its results. We have considered the application of the RTS techniques studied to only three programs, using one set of test data per program, and cannot claim that these results generalize to other workloads. However, the systems and versions used are real, large software systems, and the test suites used are the ones actually used by the developers of the considered systems.

Threats to internal validity mostly concern possible errors in our algorithm implementations and measurement tools that could affect outcomes. To control for these threats, we validated the implementations on known examples and performed several sanity checks. One sanity check that we performed involved checking that all the test cases that produce different outputs for the modified programs are actually selected by our tool. Another sanity check involved making sure that techniques EdgeLevel and TwoPhases selected exactly the same test cases. We also spot checked, for many of the changes considered, that the results produced by the two phases of our approach were correct.

3.5 Results and Discussion

In this section, we present the results of the study and discuss how they address the three research questions that we are investigating.

[4] http://www.cc.gatech.edu/aristotle/Tools/jaba.html
[5] http://pag.lcs.mit.edu/daikon/
[6] http://www.jboss.org/

Subject	*Total Types*	*Chg. Classes*	*Types in Partition*	*EdgLv (sec)*	*TwoPh (sec)*
Jaba v2	525	13	350 (66.7%)	63	69 (10)
Jaba v3	525	3	151 (28.8%)	63	48 (10)
Jaba v4	525	3	343 (65.3%)	62	70 (9)
Jaba v5	525	2	141 (26.9%)	62	46 (9)
Daikon v2	816	171	515 (63.1%)	540	185 (5)
Daikon v3	817	27	261 (32.0%)	420	63 (3)
Daikon v4	839	226	554 (66.0%)	420	246 (6)
Daikon v5	849	9	390 (45.9%)	420	186 (6)
JBoss v2	2,410	23	463 (19.2%)	420	70 (10)
JBoss v3	2,407	21	71 (3.0%)	480	18 (10)
JBoss v4	2,406	23	67 (2.8%)	480	17 (10)
JBoss v5	2,386	95	550 (23.1%)	660	133 (13)

Table 2: Savings in test selection time using TwoPhases.

Subject	*Testsuite Size*	*HighLevel Selection*	*TwoPhases Selection*
Jaba v2	707	606 (85.7%)	502 (71.0%)
Jaba v3	707	606 (85.7%)	202 (28.6%)
Jaba v4	707	606 (85.7%)	432 (61.1%)
Jaba v5	707	707 (100.0%)	707 (100.0%)
Daikon v2	200	183 (91.5%)	168 (84.0%)
Daikon v3	200	168 (84.0%)	168 (84.0%)
Daikon v4	200	183 (91.5%)	168 (84.0%)
Daikon v5	200	168 (84.0%)	40 (20.0%)
JBoss v2	639	432 (67.6%)	282 (44.1%)
JBoss v3	639	284 (44.4%)	11 (1.7%)
JBoss v4	639	242 (37.9%)	238 (37.3%)
JBoss v5	639	501 (78.4%)	452 (70.7%)

Table 3: Results of selection for HighLevel and TwoPhases.

RQ1. The first part of RQ1 concerns the effectiveness of our partitioning technique in selecting only small fractions of the program to be further analyzed. To address this part of RQ1, we analyzed the size of the partitions computed by TwoPhases for each version of each subject. The second part of RQ1 aims to investigate whether the identification of small partitions actually results in savings for the overall test-selection process. To address this second part of the question, we compared the analysis time for technique TwoPhases with the analysis time for technique EdgeLevel (which performs the same kind of edge-level analysis as TwoPhases, but on the whole program).

Table 2 shows the data that we analyzed. The table contains one row for each subject and version. Version number v_i for program P indicates data for (program, modified-version) pair (P_{i-1}, P_i). For each subject and version P_i, we report: the number of types in P_i (*Total Types*); the number of classes that syntactically changed between P_{i-1} and P_i (*Chg Classes*);[7] the number of types in the partition selected by Phase 1 of our technique (*Types in Partition*), both as an absolute number and as a percentage over the total number of types (in parentheses); the time required to perform edge-level RTS on the whole system (*EdgLv*); and the time required to perform our two-phase RTS (*TwoPh*), in the format ⟨*totaltime*⟩ (⟨*partitioning time*⟩).

Table 2 shows that the percentage of types selected ranges from 2.8%, for JBoss $v4$, to 66.7%, for Jaba $v2$, with an average of 36.9%. The table also shows that the partitioning performed efficiently: for the largest subject, JBoss, it computed the partition in 13 seconds, in the worst case. This result is encouraging because it shows that, at least for the cases considered, we can safely and efficiently avoid the analysis of a considerable part of the system. Note that the results show no direct correlation between the number of changes in the program and the size of the partition identified. Although more studies are required to confirm this observation, based on previous experience, we conjecture that the size of the partition is related more to the location of the changes than to the number of changes.

For the second part of RQ1, the results in Table 2 show that the two-phase analysis achieves overall savings in analysis time in most, but not all cases. For two versions of Jaba, $v2$ and $v4$, the time to perform the two-phase selection is higher than the time to perform selection on the whole system. However, we note that: (1) the additional cost is on the order of a few seconds and is thus compensated by the savings obtained for the other two versions of Jaba; and (2) the time for the analysis of the whole Jaba program is low, which penalizes the almost fixed overhead imposed by the partitioning. We observe that the larger the system, and the longer the analysis time, the more negligible is the partitioning time. In fact, our data show a clear trend in this direction, with average savings in test selection time that grow with the size of the subject: 6.8% for Jaba, 62% for Daikon, and 89% for JBoss. (These percentages are computed by averaging the ratio of the test-selection time for TwoPhases to the test-selection time for EdgeLevel over the five versions of each subject.) This trend is promising and shows that our technique may scale to even larger systems.

RQ2. The results in Table 2 show that the use of our two-phase technique can result in savings in test selection time. However, performing the second phase of the technique at the edge-level is not necessarily cost-effective. A technique that operates at a high-level of abstraction and performs test-case selection at the class/interface level is more efficient and may provide similar results in terms of selection. This is the question addressed by RQ2. To answer this question, we compared techniques HighLevel and TwoPhases in terms of the number of test cases selected. The results of the comparison are presented in Table 3. The table reports, for each subject and version P_i, the size of P_i's test suite (*Testsuite Size*), constant across a subject's versions, the number of test cases selected by technique HighLevel (*HighLevel Selection*), and the number of test cases selected by technique TwoPhases (*TwoPhases Selection*). The number of test cases is reported both as an absolute number and as a percentage over the total number of test cases (in parentheses).

The results in the table show that, in most cases, technique TwoPhases selects considerably fewer test cases than HighLevel. There are only two cases in which the two techniques select the same number of test cases, for Jaba $v5$ and Daikon $v3$. In all other cases, TwoPhases is more effective in reducing the number of test cases to be rerun: TwoPhases selects 21% fewer test cases than HighLevel on average, and 64% fewer in the best case, for Daikon $v5$. In the next section, we evaluate how this reduction affects the cost of the overall regression-testing process.

RQ3. With RQ3, we want to investigate what overall savings can be achieved in the regression-testing process by using technique TwoPhases. To this end, we compare the overall time to perform regression testing using TwoPhases and using RetestAll, our control technique. The overall time for a technique includes the analysis time to perform the test selection (this time is zero for RetestAll) and the time to rerun the selected test cases. Although not explicitly addressed by RQ3, we also add to the comparison technique HighLevel, to

[7]For the subjects and versions considered, there were no changed interfaces.

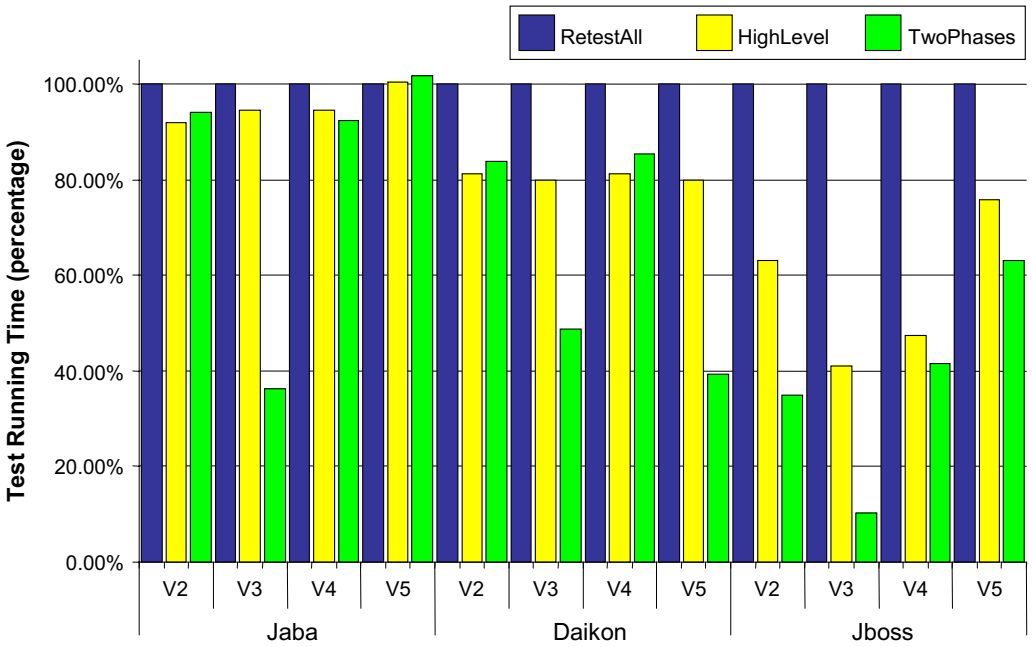


Figure 10: Overall time for regression testing using techniques RetestAll, HighLevel, and TwoPhases.

investigate whether there are cases in which the efficiency of the analysis compensates for the higher number of test cases to rerun (see Table 3).

Figure 10 shows a chart with the results of the comparison. For each subject and version P_i, we show three bars. The height of each bar indicates, from left to right, the time required to regression test P_i using technique RetestAll, HighLevel, and TwoPhases. The time is expressed as a percentage of the time required to rerun all the test cases

In all except one case, the use of both HighLevel and TwoPhases results in savings in the overall regression-testing time. The only exception is JABA $v5$, for which all test cases are selected (due to a change in the main path of the program) and, thus, the overall time is increased by the overhead of the test selection. However, the additional cost is low: 0.3% for HighLevel and 1.7% for TwoPhases. In all other cases, the savings in testing time range from 3.1% to 58.9%, with an average of 18%, for HighLevel, and from 5.9% to 89.7%, with an average of 42.8%, for TwoPhases. As for the comparison between HighLevel and TwoPhases, in 5 out of 15 cases, the techniques achieve almost the same savings. In some of these cases, HighLevel performs marginally better than TwoPhases because it selects a similar number of test cases, but it does it more efficiently. In the remaining 10 cases, and overall, TwoPhases outperforms HighLevel. If we consider, for each subject, the average across versions of the difference in savings, TwoPhases takes 14.3%, 16.2%, and 32.5% less time than HighLevel to regression test JABA, DAIKON, and JBOSS, respectively. The savings achieved by TwoPhases, with respect to RetestAll, for the three subject are 18.9%, 35.7%, and 62.5%, respectively. These data show that our technique can achieve considerable savings in terms of regression-testing time.

4. RELATED WORK

To date, a number of techniques have been developed for regression testing of procedural software (e.g., [2, 6, 7, 9, 10, 15, 17, 18, 20, 22, 23]). The technique from Ren and colleagues [15] is an approach for identifying both which test cases are affected by code changes and which changes affect each test case. Unlike our approach, their technique mostly focuses on unit test cases. Although it would be interesting to compare the cost-effectiveness of their technique to our two-phase approach, such a comparison is outside the scope of this paper.

Most of the other approaches listed above are based on identifying differences between the old and new versions of the program and on performing selection by matching such differences with coverage information for the test cases [2, 6, 17, 18, 20]. The second phase of our technique uses a similar approach for selection. However, unlike these approaches, our technique (1) handles object-oriented features, and (2) does not have to analyze the whole system, but only the partition identified in the first phase.

Rosenblum and Weyuker [16] and Harrold and colleagues [8] studied how coverage data can be used to predict the cost-effectiveness of regression-test-selection approaches. Ostrand and Weyuker studied the distribution of faults in several versions of a large industrial software system [14]. This line of work is complementary to our approach, in that their predictors could be used to inform regression testing (e.g., to avoid performing selection when it is highly unlikely to produce any savings).

Other techniques are defined for object-oriented software and are more directly related to the technique presented in this paper [7, 9, 18, 22]. White and Abdullah's technique [22] constructs a *firewall* to enclose the set of classes affected by the changes; such classes are the only ones that need to be retested. Hsia and colleagues' technique [9] is also based on the concept of class firewall defined originally by White and Leung [23], but leverages a different representation of the program. Both techniques are limited in that they do not handle certain object-oriented features (e.g., exception handling) and perform analysis only at the class level, which can be imprecise (see Table 3). Moreover, the techniques are not implemented and, thus, there is no empirical evidence of their effectiveness or efficiency.

Rothermel and colleagues extend Rothermel and Harrold's technique for RTS of C programs to C++ [18]. This technique handles only a subset of object-oriented features and requires the analysis of the whole system, which may be inefficient for large systems.

Harrold and colleagues [7] propose an RTS technique for Java. This technique handles all language features, but requires low-level analysis of the entire program under test.

Although not defined for object-oriented programs, the hybrid technique discussed by Bible and colleagues [3] is related to our two-phase technique. Similar to our approach, their technique combines a coarser-grained analysis with a finer-grained analysis, so as to apply the latter only where needed. However, their coarse-grained analysis is more expensive than our first phase because it still computes differences between the old and new programs at the method level for the whole program. Conversely, our partitioning computes simple dependences and postpones the expensive analysis to the second phase, which is performed only on a small part of the program.

5. CONCLUSIONS

In this paper, we have presented a new technique for regression test selection of Java software that is designed to scale to large systems. The technique is based on a two-phase approach. The first phase performs a fast, high-level analysis to identify the parts of the system that may be affected by the changes. The second phase performs a low-level analysis of these parts to perform precise test selection.

The paper also presents an empirical study performed on three medium-to large-sized subjects to investigate the efficiency and the effectiveness of the technique. The results of the study are encouraging: For the subjects considered, the technique (1) produces considerable savings in regression testing time (62.5% on average for the largest subject considered), and (2) scales well (in fact, its cost-effectiveness improves with the size of the program under test).

The empirical results also led us to an interesting research direction to further improve our RTS technique. The studies show that there are cases in which the second phase of the technique could be skipped in favor of a selection at the partition level (technique HighLevel). Future research could investigate heuristics (e.g., based on the size of the partition or in the location of the changes) for identifying those cases.

We are currently working on improving the efficiency of the tool and performing a controlled experiment with a larger set of versions. We are also investigating various ways to further improve the efficiency of the technique. Finally, we are investigating ways to leverage knowledge about the changes in the program to assess the adequacy of existing test cases in covering the changed parts of the code.

Acknowledgments

This work was supported in part by NSF awards CCR-0205422, CCR-0306372, and SBE-0123532. Michael Ernst provided access to Daikon's CVS. Sebastian Elbaum and the anonymous ISSTA reviewers provided insightful comments on a previous version of this paper.

6. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria, Sep. 2004.

[2] T. Ball. On the limit of control flow analysis for regression test selection. *Proceedings of the ACM-SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–142, Mar. 1998.

[3] J. Bible, G. Rothermel, and D. S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test selection techniques. *ACM TOSEM*, 10(2):149–183, Apr. 2001.

[4] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8), Aug. 1997.

[5] A. Chawla and A. Orso. A generic instrumentation framework for collecting dynamic information. In *Online Proceeding of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, Boston, MA, USA, Jul. 2004.

[6] Y. Chen, D. Rosenblum, and K. Vo. Testtube: A system for selective regression testing. *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[7] M. J. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi. Regression Test Selection for Java Software. *Proceedings of OOPSLA*, pages 312–326, Oct. 2001.

[8] M. J. Harrold, G. Rothermel, D. S. Rosenblum, and E. J. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27(3):248–263, Mar. 2001.

[9] 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.

[10] 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.

[11] A. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance (ICSM 02)*, pages 204–213, Oct. 2002.

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

[13] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[14] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 55–64, Jul. 2002.

[15] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of java programs. Technical Report DCS-TR-551, Department of Computer Science, Rutgers University, Apr. 2004.

[16] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Transactions on Software Engineering*, 23(3):146–156, Mar. 1997.

[17] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM TOSEM*, 6(2):173–210, Apr. 1997.

[18] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Journal of Software Testing, Verification, and Reliability*, pages 77–109, Jun. 2000.

[19] G. Rothermel and M.J.Harrold. Empirical studies of a safe regression test selection technique. *IEEE TSE*, pages 401–419, Jun. 1998.

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

[21] Z. Wang, K. Pierce, and S. McFarling. BMAT – a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2, May 2000.

[22] 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.

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