Proc. of the 16th Int'l. Conf. on Softw. Eng., Sorrento, Italy, May 1994, pages 201-210.

A Framework for Evaluating Regression Test Selection Techniques*

Gregg Rothermel and Mary Jean Harrold
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

Abstract

Regression testing is a necessary but expensive activity aimed at showing that code has not been adversely affected by changes. A selective approach to regression testing attempts to reuse tests from an existing test suite to test a modified program. This paper outlines issues relevant to selective retest approaches, and presents a framework within which such approaches can be evaluated. This framework is then used to evaluate and compare existing selective retest algorithms. The evaluation reveals strengths and weaknesses of existing methods, and highlights problems that future work in this area should address.

1 Introduction

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production[19]. One necessary but expensive maintenance task is *regression testing*, performed on a modified program to instill confidence that changes are correct, and have not adversely affected unchanged portions of the program. An important difference between regression testing and development testing is that during regression testing an established set of tests may be available for reuse. One approach to reusing tests, the *retest all* approach, chooses all such tests, but this strategy may consume excessive time and resources. An alternate approach, *selective retest*, chooses a subset of tests from the old test set, and uses this subset to test the modified program.

Although many techniques for selective retest have been developed[1, 2, 3, 6, 8, 9, 11, 12, 15, 20, 21, 23], there is no established basis for evaluation and comparison of these techniques. Classifying selective retest strategies for evaluation and comparison is difficult because distinct philosophies underlie the existing approaches. *Minimization* approaches[6, 10, 20] assume that the goal of regression testing is to reestablish satisfaction of some structural coverage criterion, and aim to identify a minimal set of tests that must be rerun to meet that criterion. *Coverage* approaches[2, 3, 8, 9, 12, 15, 21, 23], like minimization approaches, rely on coverage criteria, but do not require minimization. Instead, they assume that a second but equally important goal of regression testing is to rerun tests that could produce different output, and they use coverage criteria as a guide in selecting such tests. *Safe* approaches[1, 11, 18] place less emphasis on coverage criteria, and aim instead to select every test that will cause the modified program to produce different output than the original program.

These philosophies lead selective retest methods to distinctly different results in test selection. Despite these differences, we have identified categories in which selective retest approaches can be compared and evaluated. These categories are inclusiveness, precision, efficiency, generality, and accountability. *Inclusiveness* measures the extent to which a method chooses tests that will cause the modified program to produce different output. *Precision* measures the ability of a method to avoid choosing tests that will not cause the modified program to produce different output. *Efficiency* measures the computational cost and automatability, and thus practicality, of a selective retest approach. *Generality* measures the ability of a method to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications. *Accountability* measures a method's support for coverage criteria, that is, the extent to which the method can aid in the evaluation of test suite adequacy. These categories form a framework for evaluation and comparison of selective retest approaches. In this paper, we present this framework, and demonstrate its usefulness by evaluating a representative sample of selective retest techniques.

*This work was partially supported by a grant from Data General Corporation, and by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University.

The main benefit of our framework is that it provides a way to evaluate and compare existing selective retest approaches. Evaluation and comparison of existing techniques is useful for choosing appropriate approaches for particular applications. For example, if very reliable code is essential, a safe technique may be required regardless of cost. On the other hand, if execution time for the regression tests is the most important factor, a minimization technique may be desirable, even though some tests that expose faults may be omitted from the regression test suite. Evaluation and comparison of existing approaches also provides insights into the strengths and weaknesses of current methods, and guidance in choosing areas that future work on selective retest should address. Our framework can also be used to evaluate newly developed selective retest techniques.

In the next section, we provide background, definitions and observations about regression testing. In Section 3 we explore our categorization method, and give more details about the framework for evaluation and comparison of selective retest methods. In Section 4, we use our framework to review and compare several of the current methods for selective retest and in Section 5, we conclude and discuss future work.

2 Regression Testing

Most work on regression testing addresses the following problem.

Problem 1: Given program P, its modified version P', and test set T used previously to test P, find a way, making use of T, to gain sufficient confidence in the correctness of P'.

Solutions to Problem 1 typically consist of the following steps:

1. *Identify the modifications that were made to P.* Some approaches assume the availability of a list of modifications, perhaps created by a cooperating editor that tracks the changes applied to P. Other approaches assume that a mapping of code segments in P to their corresponding segments in P' can be obtained using algorithms that perform slicing[22] or establish graph isomorphisms[11].
2. *Select $T' \subseteq T$, the set of tests to reexecute on P'.* This step may make use of the results of step 1, coupled with *test history information* that records the input, output, and *execution history* for each test. An execution history for a given test lists the statements or code segments exercised by that test. For example, Figure 1 shows test history information for procedure AVG.

Procedure AVG

```
S1.    count = 0
S2.    fread(fileptr,n)
S3.    while (not EOF) do
S4.        if (n<0)
S5.            return(error)
           else
S6.            numarray[count] = n
S7.            count++
           endif
S8.        fread(fileptr,n)
       endwhile
S9.    avg = calcavg(numarray,count)
S10.   return(avg)
```

test number	input	output	execution history
T1	empty file	0	S1,S2,S3,S9,S10
T2	-1	error	S1,S2,S3,S4,S5
T3	1 2 3	2	S1,S2,S3,S4,S6,S7,S8,S3,...,S9,S10

Figure 1: AVG and its test history information.

3. *Retest P' with T', establishing P''s correctness with respect to T'.* Since we are concerned with testing the correctness of the modified code in P', we *retest* P' with each $T_i \in T'$ by keeping all factors other than the code, and possibly the expected output, the same as they were when P was tested with T_i. As tests in T' are rerun, new test history information may be gathered for them.
4. *If necessary, create new tests for P'.* When T' does not achieve the required coverage of P', new tests are needed. These may include functional tests required by specification changes, and/or structural tests required by coverage criteria.
5. *Create T'', a new test set/history for P'.* The new test set includes tests from steps 2 and 4, and old tests that were not selected, provided they remain valid. New test history information is gathered for tests whose histories have changed, if those histories have not yet been recorded.

Step 2 addresses the *selective retest problem.* However, in order to evaluate approaches to selective retest, we must consider them in the context of all steps required to solve Problem 1. For example, the cost and effectiveness of selective retest algorithms are impacted by assumptions about modification information (step 1), and by their ability to support evaluations of adequacy (step 4). We discuss this further in Section 3.

To choose tests from test set T, selective retest algorithms partition T into two subsets: tests to rerun and tests not to rerun. To reduce the time involved in retesting we want to choose only those tests from

T that will produce different output when run on P'; we call such tests modification-revealing.

> **Definition 1:** A test $T_i \in T$ is *modification-revealing* if it produces different outputs in P and P'.

However, we cannot in general find an algorithm that will identify the set of modification-revealing tests in T. For a given program P and a finite nonempty test set T, consider the following decision problem, where we regard P and T as fixed.

$D_{P,T}$: For an arbitrary program P', is there a test $T_i \in T$ that is modification-revealing for P and P'?

For each fixed P and T, $D_{P,T}$ is undecidable. While we do not prove this here, the undecidability follows from a straightforward application of Rice's Theorem[4]. The crux of the difficulty lies in the fact that we cannot predict when P' will halt for a test T_i.

Although we cannot in general find an algorithm that will identify the set of modification-revealing tests in T, we can identify a necessary condition for a test $T_i \in T$ to be modification-revealing. In order for a test $T_i \in T$ to produce different output in P and P', and thus be modification-revealing, it must execute some code modified from P to P'[1][1]. We define tests that execute modified code as modification-traversing.

> **Definition 2:** A test $T_i \in T$ is *modification-traversing* if it executes a new or modified statement in P', or misses a statement in P' that it executed in P.

Although modification-revealing tests are necessarily modification-traversing, not all modification-traversing tests are modification-revealing. For example, consider what happens when statement `S1. count=0` of procedure AVG (Figure 1) is modified. In this case, test $T2$ is modification-traversing because it executes the modified version of S1 in the new version of the procedure. However, test $T2$ is not modification-revealing: it traverses no statement that uses the value computed in $S1$ and thus cannot produce different output in the new version of AVG.

3 A Framework for Evaluating Selective Retest Methods

In this section we describe the five categories that constitute our framework for evaluating selective retest strategies. Assume throughout this section that P is a program, P' is a modified version of P, T is a set of tests for P, S is a selective retest strategy, and T' is a set of tests selected by S given P, P', and T.

Inclusiveness

Inclusiveness measures the extent to which S chooses modification-revealing tests from T for inclusion in T'. We define inclusiveness relative to a particular program, modified program, and test set as follows:

> **Definition 3:** Suppose T contains n modification-revealing tests, and S selects m of these tests. The *inclusiveness of S relative to P, P', and T* is the percentage calculated by the expression $((m/n) * 100)$.

For example, if T contains 50 tests, 8 of which are modification-revealing, and S selects only 2 of the 8 modification-revealing tests, then S is 25% inclusive relative to P, P', and T.

If a method S always selects all modification-revealing tests, we call S safe.

> **Definition 4:** If for all P, P' and T, S is 100% inclusive relative to P, P' and T then S is *safe*.

For an arbitrary choice of S, P, P', and T, there is no algorithm to determine the inclusiveness of S relative to P, P', and T. Such an algorithm would require the ability to determine whether an arbitrary test T_i is modification-revealing. However, we can still draw conclusions about inclusiveness in several useful ways. First, we may be able to prove that S is safe by showing that S selects a known superset of the modification-revealing tests. Second, we may be able to prove that S is *not* safe by finding a case where S misses a modification-revealing test. Third, we may be able to compare methods $S1$ and $S2$ to each other if we can show that the methods select subsets Q and R of the modification-revealing tests, such that Q is demonstrably a subset (superset) of R. Finally, we can experiment to approximate the inclusiveness of S relative to a particular choice of P, P', and T. We run S on P, P', and T, generating set T'. Then, we run P' on each test in T to determine which tests in T are modification-revealing, and then compare the modification-revealing tests to those in T'. (Since we cannot determine whether P' will halt when run on test $T_i \in T$, we can only estimate the subset of T that is modification-revealing.) We can also perform such experiments on a group W of programs, modified programs, and test sets to measure S's inclusiveness relative to W by taking the average of the relative

[1] We associate information about declarations and unexecutable initialization statements with a program's entry. Thus, for changes in these types of statements, any test is modification-traversing.

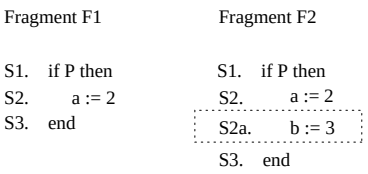

Figure 2: Fragments illustrating statement addition.

inclusivenesses of S for each member of W. Then we could compare S's performance relative to W to that of other methods. Such experimentation could provide estimates of S's inclusiveness.

Inclusiveness and safety can be significant measures. If S is safe then we know that S will select every modification-revealing test, while if S is not safe it may omit tests that can expose faults. Furthermore, we hypothesize that given selective retest strategies $S1$ and $S2$, if $S1$ is more inclusive than $S2$, then $S1$ has a greater ability to expose faults than $S2$, because it selects more modification-revealing tests.

When we evaluate S's inclusiveness, we must consider the effects of new and deleted code. When new code is added to P, T may already contain tests that are modification-revealing with respect to this new code. For example, consider the program fragments in Figure 2. In the absence of other code changes, any test that executes statement $S2$ in fragment $F1$ necessarily executes statement $S2a$ in fragment $F2$, and may (depending on subsequent statements encountered) produce different output. Similarly, when code is deleted from P, T may contain tests that are modification-revealing with respect to this deleted code. For example, consider the program fragments in Figure 3. In the example on the left, statement $S1$ in fragment $F1$ is deleted, yielding fragment $F2$. In the absence of other modifications, any test that traversed $S1$ in $F1$ will produce different output in $F2$. In the example on the right, two statements have been deleted from fragment $F3$, yielding fragment $F4$. In the absence of other modifications, any test in which both P and Q are true produces different output in $F4$. If S does not account for the effects of new and/or deleted statements, we may be able to find examples of such statements that prove that S is not safe.

Precision

Precision measures the extent to which a selective retest strategy omits tests that are non-modification-revealing. We define precision relative to a particular program, modified program and test set, as follows:

Definition 5: Suppose T contains n non-modification-revealing tests, and S selects m of these tests. The *precision of S relative to P, P', and T* is the percentage calculated by the expression $((m/n) * 100)$.

For example, if T contains 50 tests, 44 of which are non-modification-revealing with respect to P', and S omits 33 of these 44 tests, then S is 75% precise relative to P, P', and T.

As with inclusiveness, there is no algorithm to determine, for an arbitrary choice of S, P, P', and T, the precision of S relative to P, P', and T. However, we can still measure precision in several ways. First, we may be able to compare the precision of methods $S1$ and $S2$ if we can show that they select subsets Q and R, respectively, of the non-modification-revealing tests such that Q is demonstrably a subset (superset) of R. Second, as with inclusiveness, we may be able to demonstrate that S is *not* precise by finding a case in which S selects a test that is non-modification-revealing. Third, we can use experimentation to compare relative precisions. Finally, we may be able to show that S is precise by proving that S omits a superset of the non-modification-revealing tests.

Precision is useful because it measures the extent to which S *avoids* selecting tests that cannot produce different program output.

If the precision of a strategy relative to every P, P' and T is 100%, we say the strategy is *precise*. A precise strategy always selects *only* modification-revealing tests, while an imprecise strategy selects some tests that cannot produce different output.

Efficiency

We measure the efficiency of a selective retest method in terms of its space and time requirements. Space efficiency is affected by the test history and program analysis information a method must store. Where time is concerned, a selective retest strategy is more economical than a retest-all strategy if the cost of selecting T' is less than the cost of running the tests in T - T'[13]. Thus, efficiency varies with the size of the test set that a method selects, as well as with the computational cost of that method. Methods for quantitatively evaluating algorithms are well understood and will not be discussed here. However, we discuss several factors that must be considered when evaluating the efficiency of selective retest algorithms.

One factor influencing the computational expense of a selective retest method is whether or not the method must calculate information on program modifications. If a method must determine which program components have been modified, deleted, and added,

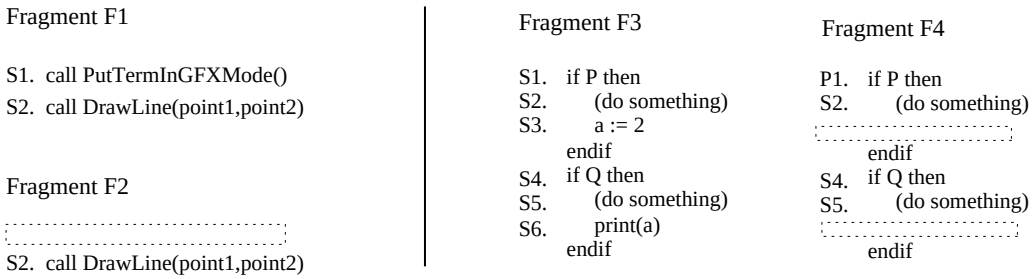

Figure 3: Fragments illustrating statement deletion.

or construct a mapping between components of P and P', that method may require more computational resources than a method that does not calculate such information. A second factor impacting a method's computational expense is its automatability.

When evaluating efficiency, we consider two distinct phases in the regression testing life cycle: the *preliminary phase* and the *critical phase*. The preliminary phase of regression testing begins after a product is released. During this phase, programmers work on the next version of the code, adding enhancements and fixing bugs. Testers may still be testing code during this phase, particularly at the unit level, but final tests, and in particular integration and system tests, await the inclusion of the final "fix". Eventually the release period nears, code is "frozen", and a test version of the release is constructed. At this point, the critical phase of regression testing begins. During this phase, regression testing is the dominating activity, and its time is limited by the deadline for product release. It is in this critical phase that minimization of cost is most important for regression testing.

It is possible for selective retest methods to take advantage of the phases of the regression testing life cycle. For example, although test history and program analysis information may be gathered during the critical phase while tests are run, it is economically preferable to gather the bulk of that information during the preliminary phase. Our evaluation favors methods whose principal expense occurs in the preliminary phase of regression testing over those that expect to complete most of their work during the critical phase.

Generality

The generality of a selective retest method is its ability to function in a wide and practical range of situations. Ideally, selective retest algorithms should function in the presence of arbitrarily complex code modifications. Also, although we could apply different methods in different settings, we prefer methods that handle all types of language constructs, and large classes of programs, at the interprocedural as well as intraprocedural levels. The need for information on program modifications is also a generality issue since requiring knowledge of modifications may impose unreasonable restrictions.

Accountability

Studies suggest that structural test coverage criteria increase the effectiveness of testing[14]. If a program is initially tested with such a criterion, then after modifications it is desirable to confirm that the criterion remains satisfied. Alternatively, if a program is not initially tested using a coverage criterion, it is still possible to apply a criterion at regression test time, ensuring that all new or modified portions of the code have been covered properly[7].

We use the term *accountability* to refer to the extent to which a selective retest method promotes the use of structural coverage criteria. Selective retest methods may promote this use by identifying unsatisfied program components, or selecting tests that maximize the coverage achievable. Both coverage and minimization methods facilitate such efforts.

4 A Comparison of Regression Test Selection Techniques

In this section, we use our framework to evaluate a representative sample of existing regression test strategies; other strategies are evaluated in [17].

4.1 Minimization methods

Minimization approaches to selective retest have been described by Fischer, Raji, and Chruscicki[6], Hartmann and Robson[10], and Sherlund and Korel[20]. In this section we examine Hartmann and Robson's method, since it extends Fischer, Raji, and Chruscicki's work, and exemplifies the strengths and weaknesses of minimization approaches.

Hartmann and Robson's selective retest method uses systems of linear equations to express relation-

ships between tests and program segments (basic blocks). These equations are obtained from matrices that track the segments of code reached by test cases, and the segments reachable from other segments. The solution to such a system of equations identifies a minimal set of tests T' such that each segment reachable from a changed segment is exercised by at least one test in T'. Dataflow information is used to ensure that only tests that traverse affected uses are selected.

Inclusiveness: Hartmann and Robson's approach, like other minimization approaches, is not safe. If several tests exercise a particular modified statement and all of these tests exercise a particular affected statement, only one such test is selected, unless the others are selected for coverage elsewhere. Some of the tests that are omitted may produce different output if executed. For example, suppose statement S1 in procedure AVG (Figure 1) is erroneously modified, to "`count=1`". Tests T1 and T3 both traverse S1 and reach affected statement S9, which uses the value of `count`. Hartmann and Robson's method selects only one of these tests, omitting the other. However, only test T3 exposes this fault and if T1 is chosen instead, the fault will not be detected. Hartmann and Robson's approach also may omit tests that can reveal faults caused by non-dataflow dependencies, such as tests reaching S2 from S1 in the fragments on the left in Figure 3.

Precision: Hartmann and Robson's method omits tests that are non-modification-traversing by ignoring tests that do not execute changed segments. The method also uses dataflow information to further increase precision.

Efficiency: Since Hartmann and Robson's method is a minimization method, it returns small test sets and thus reduces the time required to run regression tests. The method is also fully automatable. However, due to the calculations required for solving systems of linear equations, the approach may be data and computation intensive on large programs. Moreover, the computation required by the method is performed on the new version of the program, and thus cannot be accomplished until program changes are complete, when the regression testing process has entered the critical phase. Finally, the method requires knowledge of the correspondence between segments in P and P'. This correspondence must also be computed after testing has entered the critical phase.

Generality: Hartmann and Robson's method is defined and implemented for "C," and can handle all "C" structures. The method depends only on identifying code segments, so it could be implemented for any procedural language. The method may also be extended to handle interprocedural test selection, by treating entire routines rather than basic blocks as segments. However, this extension reduces precision by admitting tests that are non-modification-traversing, such as tests that traverse a modified procedure but do not actually traverse modified code in the procedure. More importantly, Hartmann and Robson's method is defined only for situations where code modifications do not alter control flow. Thus, the method does not handle addition, deletion, or modification of predicate statements. The authors suggest that future work to handle changes in control flow will force reanalysis of the changed program, which is expensive.

Accountability: Hartmann and Robson's approach is motivated by the desire to achieve structural coverage of a program at a basic block level. The approach establishes such coverage by reusing as many existing tests as possible, without selecting tests that are redundant in terms of coverage.

4.2 Coverage methods

A majority of existing selective retest methods are most aptly described as coverage methods. These include approaches proposed by Bates and Horwitz[2], Benedusi, Cimitile, and De Carlini[3], Gupta, Harrold, and Soffa[7], Harrold and Soffa[8, 9], Leung and White[12], Ostrand and Weyuker[15], Taha, Thebaut and Liu[21], and Yau and Kishimoto[23]. In this section we apply our framework to the methods proposed by Harrold and Soffa, and Bates and Horwitz, since these methods let us illustrate some important facets of the use of our framework.

Harrold and Soffa's method

Harrold and Soffa[8, 9] present a test selection method based on dataflow testing techniques. Their approach identifies changed definition-use pairs in a program, and selects tests that exercise these pairs.

Inclusiveness: Harrold and Soffa's method specifically selects *all* tests that cover affected pairs, thereby selecting a superset of the set selected by Hartmann and Robson's method. Nevertheless, the approach may omit modification-revealing tests in at least three ways, and thus is not safe. First, the approach may omit tests that exercised statements deleted from P. Second, by relying solely on data dependencies as a guide in test selection, the method misses tests that may be exposed by other forms of dependencies. Hence, for code changes such as that depicted on the left in Figure 3, the method does not select any tests.

Finally, Harrold and Soffa's method may omit tests that execute modified output statements that contain no variable uses, although these statements may cause the program to produce different output.

Precision: Because Harrold and Soffa's approach only selects tests that traverse new or modified definition-use pairs, all tests selected necessarily traverse new or modified statements. Thus, the method selects only modification-traversing tests. Moreover, by selecting tests that exercise definition-use pairs, Harrold and Soffa's approach is capable of greater precision than methods that select all modification-traversing tests. On the other hand, the effects of aliasing and dynamic memory usage may cause loss of precision, some of which can be mitigated by the use of algorithms for identifying alias information[16].

Efficiency: Harrold and Soffa's approach requires storage and/or calculation of dataflow information, but dataflow calculation is at worst an $O(n^2)$ operation that is well understood and is accomplished by many compilers. However, the approach also requires knowledge of program modifications, and typically assumes that these modifications will be provided through a program development environment. To be efficient, such environments must handle incremental updates of dataflow information as changes are applied to programs. In doing so these environments incur additional computation and storage costs.

Generality: Harrold and Soffa's approach is fairly general, because it requires only control flow graphs and test execution histories. Also, the method handles structural program changes. Moreover, Harrold and Soffa show how to apply dataflow methods to interprocedural test selection, at the cost of added interprocedural analysis, facilitating larger-scale applicability. On the other hand, the assumption of a programming environment that tracks changes restricts the applicability of the approach.

Accountability: Harrold and Soffa's method is highly accountable, lending direct support for dataflow coverage criteria.

Bates and Horwitz's method

Bates and Horwitz[2] present a test selection method based on program dependence graph adequacy criteria. The program dependence graph encodes both control and data dependence information for a procedure[5]. Bates and Horwitz use slicing algorithms to group statements in P and P' into *execution classes* such that a test that executes any statement in an execution class executes all statements in that class. Next, they identify *affected statements*, which

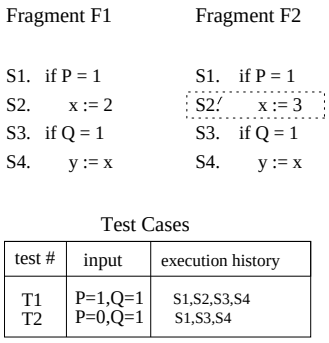

Figure 4: Fragments distinguishing modified and affected statements.

are statements that may exhibit different behavior in P', by comparing slices of corresponding points in P and P'. Finally, they select for retest all tests that exercise any statement in the same execution class as an affected statement.

Inclusiveness: Bates and Horwitz's method successfully identifies tests that traverse modified statements, because all modified statements are identified as "affected". Moreover, by using execution classes the method is able to identify tests that traverse new statements. However, Bates and Horwitz's method does not necessarily account for deleted statements, and thus is not safe. For example, in both cases presented in Figure 3, their method selects no tests.

Precision: The technique of selecting tests through affected statements, which ensures selection of tests of new statements, causes Bates and Horwitz's method to admit tests that are non-modification-traversing. For example, when applied to Fragment $F2$ of Figure 4, the method correctly selects test $T1$, but also selects test $T2$, because $T2$ executes a statement ($S4$) that is affected by the change to $S2$. As we have discussed, test $T2$ is non-modification-traversing, and cannot cause $F2$ to produce different output than $F1$. Furthermore, since Bates and Horwitz's method relies upon slices along data and control edges in the program dependence graph, its precision is adversely impacted by assumptions required in the presence of aliasing and dynamic memory allocation. As with dataflow-based methods, some precision loss can be prevented by using algorithms for identifying alias information[16].

Efficiency: Bates and Horwitz's method may require a large number of program slices. The method computes a *control slice* for every statement in P and

every statement in P'. It then computes a backward slice on each statement in P that has a corresponding statement in P', and each statement in P' that has a corresponding statement in P. Moreover, since slice comparisons must be done with respect to statements in P', the costs of slicing on P' are incurred after modifications are complete, when testing has entered the critical phase. Finally, the method requires prior knowledge of changes, in the form of a mapping of statements in P to their modified versions in P'. This mapping must be computed after modifications are complete, hence in the critical phase.

Generality: Bates and Horwitz's approach supports all types of program modifications, but is presented only for a restricted set of language constructs. Furthermore, no method for interprocedural testing is suggested.

Accountability: Like other coverage methods, Bates and Horwitz's method is highly accountable, aiding in the satisfaction of program dependence graph coverage criteria.

4.3 Safe methods

Only three safe methods for selective regression testing exist. These are the methods of Laski and Szermer[11], Rothermel and Harrold[18], and Agrawal, Horgan, Krauser, and London[1]. We discuss the first two approaches.

Laski and Szermer's method

Laski and Szermer[11] present a method for identifying the modifications to a program, which has applications to regression testing. Laski and Szermer's algorithm computes control dependence information for a procedure and its changed version, and then computes the control *scope* of each decision statement in the procedures via transitive closure on control dependence. This information is used to identify *clusters*, which are single-entry, single-exit subsets of flow-graph nodes, and to establish isomorphisms between the flow graphs of P and P'. While establishing isomorphisms, the method identifies new, deleted, and modified clusters. Clusters are like parameterless procedures; unit and integration testing can be performed on them. Tests that do not execute new, modified, or deleted clusters are omitted.

Inclusiveness: Laski and Szermer's method is safe, because it identifies all modification-traversing tests for retest. The approach handles both new and deleted code properly, selecting all tests that exercise new statements, and all that formerly exercised deleted statements.

Precision: Laski and Szermer's algorithm can identify clusters that are larger than necessary. Since the method selects all tests known to exercise changed clusters, unnecessarily large clusters can result in selection of tests that are non-modification-traversing. For example, Laski and Szermer's algorithm identifies a cluster consisting of nodes S3 through S10 for procedure AVG in Figure 1, but no smaller clusters within that cluster. If a new line, `S5'. print("Improper data in input file.")`, is added to procedure AVG just before line S5, Laski and Szermer's method selects tests T1, T2 and T3 for retest, because all three tests exercise the modified cluster enclosing the new line. However, only test T2 actually exercises the new statement; tests T1 and T3 are non-modification-traversing.

Efficiency: Laski and Szermer's algorithm for computing scope is $O(n^3)$, and does not require undue storage, so the method is reasonably efficient. The method is also fully automatable. However, the algorithm does compute *all* corresponding program parts, which we shall see involves more work than necessary in the context of regression test selection. Also, computation of corresponding parts is done after modifications have been completed, when testing is in the critical phase.

Generality: Since Laski and Szermer's algorithm works on control flow graphs, it is fairly general. Moreover, the method handles all forms of program modifications. However, interprocedural test selection is not addressed.

Accountability: Laski and Szermer's algorithm provides no particular support for any coverage criterion. However, by identifying changed clusters, the algorithm yields information about portions of the program on which coverage must be reestablished, which could be used to advantage.

Rothermel and Harrold's method

Rothermel and Harrold[18] present a safe regression test selection method based on control dependence graphs. Control dependence graphs encode control dependence information for a procedure. Rothermel and Harrold's method constructs control dependence graphs for P and P', and instruments tests to report the *regions* (groups of statements sharing common control conditions) executed by the tests. Any test that executes a region of code that contains a changed statement is selected for retest.

Inclusiveness: Since Rothermel and Harrold's approach selects every modification-traversing test, it is safe. The approach also handles new and deleted code.

Precision: Rothermel and Harrold's approach selects only modification-traversing tests, and thus is more precise than methods that choose non-modification-revealing tests. However, the method makes no use of dataflow or other information that could reduce the size of selected test sets further.

Efficiency: The running time of Rothermel and Harrold's method is bounded by the time it takes to construct control dependence graphs, which is $O(n^2)$, so the method is efficient. Moreover, much of the computation required by Rothermel and Harrold's method, such as construction of the control dependence graph for P and collection of test history information, may be completed during the preliminary regression testing phase. The only work that must be done during the critical phase of regression testing is the construction of the control dependence graph for P', and the execution of a tree walk on P and P'. Furthermore, since the goal of Rothermel and Harrold's method is to identify tests, the method can examine fewer parts of programs than Laski and Szermer's method, for which the goal is to identify corresponding program components. This improvement is due to the fact that when Rothermel and Harrold's algorithm walks control dependence graphs looking for differing regions, it does not need to explore subgraphs within a changed region; it takes advantage of the fact that all tests entering the region are modification-traversing and does not look at enclosed regions. Rothermel and Harrold's approach is fully automatable. Finally, since Rothermel and Harrold's method requires test histories listing just the regions executed by a test, its space requirements are more modest than methods requiring test histories on a per statement basis.

Generality: Rothermel and Harrold's method applies to all programs in procedural languages, because control dependence graphs may be constructed directly from control flow graphs. The method also applies to all types of program modifications. Furthermore, Rothermel and Harrold offer a way to apply their method interprocedurally, that promises increased savings for selective retest at the integration and system test level.

Accountability: By identifying changed regions of code, Rothermel and Harrold's method identifies the areas of the code in which coverage needs to be verified, but does not aid with any particular coverage criteria.

5 Conclusion

In this paper we have presented a framework for evaluating regression test selection techniques that analyzes techniques in terms of inclusiveness, precision, efficiency, generality, and accountability. We have illustrated the application of this framework by using it to evaluate several regression test selection strategies.

Our evaluations indicate that despite differing underlying philosophies, selective retest methods may be more clearly compared and understood when our framework is employed. Parts of our framework (generality and accountability in particular) are largely qualitative. More quantitative comparisons using experimentation would be useful. Nevertheless, as presented the framework helps identify strengths and weaknesses of various approaches, and can guide the choice of a selective retest method for practical application.

Moreover, using our framework, we have identified several areas in which selective retest approaches need improvement. In particular, our evaluations show that a majority of current approaches are concerned with coverage or minimization rather than safety. Accountability is an important issue since coverage criteria are useful for improving test adequacy. However, in many practical situations the most important concern is a method's safety. In particular, where tests are functional and specification-based, as is typically the case for system and integration testing, safety may be more important than coverage. Testing professionals are reluctant, in practice, to discard any tests that may expose errors. Thus, effective safe approaches are needed. Existing safe approaches would benefit most from improvements in precision and accountability.

As the evaluations also suggest, one major drawback for most current methods is their need for information on program modifications or mappings between "old" and "new" sections of code. This need leads techniques to either assume that knowledge of modifications will be provided by an incremental editor, or that some algorithm will be used to calculate a mapping. The former assumption adds requirements to the production environment that may not be easily satisfied. The latter assumption increases the cost of the method and forces more work to be done during the critical phase of regression testing.

Finally, few solutions have been offered to the problem of interprocedural regression test selection. Only two approaches analyzed in this paper, and only one not analyzed[12], address this issue. Interprocedural testing is precisely the area where selective retest methods may make the greatest impact. A method

that saves a few tests of a single procedure does not offer a very high payoff, but when methods are applied to large modules, savings are multiplied. Moreover, in practice most regression testing currently consists of integration and system tests, which necessarily involve groups of procedures. Thus, work concentrating on interprocedural regression testing is needed.

Acknowledgement

We would like to thank Lori Clarke for her help in revising this paper. We are also grateful to Dave Jacobs and to the anonymous reviewers, who made many helpful suggestions for improvement.

References

[1] H. Agrawal, J. Horgan, E. Krauser, and S. London. Incremental Regression Testing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 348–357, September 1993.

[2] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[3] P. Benedusi, A. Cimitile, and U. De Carlini. Post-maintenance testing based on path change analysis. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 352–61, October 1988.

[4] M. Davis and E. Weyuker. *Computability, Complexity, and Languages.* Academic Press, Boston, MA, 1993.

[5] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–49, July 1987.

[6] K.F. Fischer, F. Raji, and A. Chruscicki. A methodology for retesting modified software. In *Proceedings of the National Telecommunications Conference B-6-3*, pages 1–6, November 1981.

[7] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, November 1992.

[8] M.J. Harrold and M.L. Soffa. An incremental approach to unit testing during maintenance. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 362–367, October 1988.

[9] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–67, December 1989.

[10] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–8, January 1990.

[11] J. Laski and W. Szermer. Identification of program modifications and its applications in software maintentance. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 282–90, November 1992.

[12] H.K.N. Leung and L.J. White. A study of integration testing and software regression at the integration level. In *Proceedings of the Conference on Software Maintenance - 1990*, pages 290–300, November 1990.

[13] H.K.N. Leung and L.J. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance - 1991*, pages 201–8, October 1991.

[14] E.F. Miller. Exploitation of software test technology. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, page 159, June 1993.

[15] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–47, September 1988.

[16] H. Pande, B. G. Ryder, and W. Landi. Interprocedural def-use associations in C programs. In *Proceedings of the Fourth ACM Symposium on Testing, Analysis and Verification (TAV4)*, October 1991.

[17] G. Rothermel and M.J. Harrold. A comparison of regression test selection techniques. Technical Report 114, Clemson University, Clemson, SC, April 1993.

[18] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, September 1993.

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

[20] B. Sherlund and B. Korel. Modification oriented software testing. In *Conference Proceedings: Quality Week 1991*, pages 1–17, May 1991.

[21] A.B. Taha, S.M. Thebaut, and S.S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, pages 527–34, September 1989.

[22] W. Yang. Identifying syntactic differences between two programs. *Software—Practice and Experience*, 21(7):739–55, July 1991.

[23] S.S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *Proceedings of the COMPSAC '87: The Eleventh Annual International Computer Software and Applications Conference*, pages 272–77, October 1987.