IEEE Transactions on Software Engineering, V.22, no. 8, August 1996, pages 529-551

Analyzing Regression Test Selection Techniques[1]

Gregg Rothermel and Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
395 Dreese Lab, 2015 Neil Avenue
Columbus, OH 43210-1277
{grother,harrold}@cis.ohio-state.edu
August 10, 1998

Abstract

Regression testing is a necessary but expensive maintenance activity aimed at showing that code has not been adversely affected by changes. Regression test selection techniques reuse tests from an existing test suite to test a modified program. Many regression test selection techniques have been proposed; however, it is difficult to compare and evaluate these techniques because they have different goals. This paper outlines the issues relevant to regression test selection techniques, and uses these issues as the basis for a framework within which to evaluate the techniques. We illustrate the application of our framework by using it to evaluate existing regression test selection techniques. The evaluation reveals the strengths and weaknesses of existing techniques, and highlights some problems that future work in this area should address.

Keywords: software maintenance, regression testing, selective retest, regression test selection.

1 Introduction

Estimates indicate that software maintenance activities account for as much as two-thirds of the cost of software production[36]. 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 suite of tests may be available for reuse. One regression testing strategy, the *retest-all* approach, reruns all such tests, but this strategy may consume excessive time and resources. *Regression test selection techniques*, in contrast, attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite.[2]

Although some regression test selection techniques select tests based on information collected from program specifications[28, 40] most techniques select tests based on information about the code of the program and the modified version[1, 2, 3, 5, 7, 10, 11, 13, 15, 16, 17, 18, 20, 19, 24, 25, 27, 28, 31, 33, 35, 34, 37, 38, 39, 41, 42, 45]. These code-based techniques pursue three distinct goals. Coverage techniques locate program components that have been modified or affected by modifications, and select tests in T that exercise those components. Minimization techniques work like coverage techniques, but select minimal sets of tests through

[1]This work was partially supported by grants from Microsoft, Incorporated and Data General Corporation, and by NSF under Grants CCR-9109531 and CCR-9357811 to Clemson University.

[2]A second important task for regression testing is to find ways in which the existing test suite is not adequate for testing a modified program, and indicate where new tests might be needed. In this work, however, we are concerned only with the process of reusing existing tests. We discuss this further in Section 2.

modified or affected program components. Safe techniques select every test in T that can expose one or more faults in P'. Given this abundance of regression test selection techniques, if we wish to choose a technique for practical application, we need a way to compare and evaluate the techniques.

Differences in underlying goals lead regression test selection techniques to distinctly different results in test selection. Despite these philosophical differences, we have identified categories in which regression test selection techniques can be compared and evaluated. These categories are inclusiveness, precision, efficiency, and generality. *Inclusiveness* measures the extent to which a technique chooses tests that will cause the modified program to produce different output than the original program, and thereby expose faults caused by modifications. *Precision* measures the ability of a technique to avoid choosing tests that will not cause the modified program to produce different output than the original program. *Efficiency* measures the computational cost, and thus, practicality, of a technique. *Generality* measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex code modifications, and realistic testing applications. These categories form a framework for evaluation and comparison of regression test selection techniques. In this paper we present this framework, and demonstrate its usefulness by applying it to the code-based regression test selection techniques that we cited above.

The main benefit of our framework is that it provides a way to evaluate and compare existing regression test selection techniques. Evaluation and comparison of existing techniques helps us choose appropriate techniques for particular applications. For example, if we require very reliable code, we may insist on a safe selective retest technique regardless of cost. On the other hand, if we must reduce testing time we may choose a minimization technique, even though in doing so we may fail to select some tests that expose faults in the modified program. Evaluation and comparison of existing techniques also provides insights into the strengths and weaknesses of current techniques, and guidance in choosing areas that future work on regression test selection should address.

In the next section, we provide background material on regression testing in general, and on the regression test selection problem in particular. In Section 3 we discuss theoretical issues that provide motivation for our framework. In Section 4 we present our framework for comparing and evaluating regression test selection techniques. In Section 5, we use our framework to review and compare existing techniques. In Section 6, we conclude and discuss future work.

2 Background

Let P be a program, let P' be a modified version of P, and let S and S' be the specifications for P and P', respectively. $P(i)$ refers to the output of P on input i, $P'(i)$ refers to the output of P' on input i, $S(i)$ refers to the specified output for P on input i, and $S'(i)$ refers to the specified output for P' on input i. Let T be a set of tests (a *test suite*) created to test P. A *test* is a 3-tuple, <identifier, input, output>, in which *identifier* identifies the test, *input* is the input for that execution of the program, and *output* is the specified output, $S(input)$, for this input. For simplicity, in the sequel we refer to a test $\langle t, i, S(i)\rangle$ by its identifier t, and refer to the outputs $P(i)$ and $S(i)$ of test t for input i as $P(t)$ and $S(t)$, respectively.

Research on regression testing spans a wide variety of topics. Dogsa and Rozman[9], Hoffman and

Brealey[22], Hoffman[21], Brown and Hoffman[6], and Ziegler, Grasso, and Burgermeister [46] focus on test environments and automation of the regression testing process. Lewis, Beck, and Hartmann[30] investigate automated capture-playback mechanisms and test suite management. Hartmann and Robson[20], Taha, Thebaut, and Liu[39], Harrold, Gupta, and Soffa[14], and Wong et al.[43] address test suite management. Binkley[4] presents an algorithm that constructs a reduced-size version of the modified program for use in regression testing. Leung and White[26] discuss regression testability metrics. Most recent research on regression testing, however, concerns *selective retest techniques.*

Selective retest techniques reduce the cost of testing a modified program by reusing existing tests and identifying the portions of the modified program or its specification that should be tested. Selective retest techniques differ from the *retest-all* technique, which reruns all tests in the existing test suite. Leung and White[29] show that a selective retest technique is more economical than the retest-all technique if the cost of selecting a reduced subset of tests to run is less than the cost of running the tests that the selective retest technique lets us omit.

A typical selective retest technique proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P'.
2. Test P' with T', to establish the correctness of P' with respect to T'.
3. If necessary, create T'', a set of new functional or structural tests for P'.
4. Test P' with T'', to establish the correctness of P' with respect to T''.
5. Create T''', a new test suite and test history for P', from T, T', and T''.

In performing these steps, a selective retest technique addresses four problems. Step 1 addresses the *regression test selection problem*: the problem of selecting a subset T' of T with which to test P'. Step 3 addresses the *coverage identification problem*: the problem of identifying portions of P' or S' that require additional testing. Steps 2 and 4 address the *test suite execution problem*: the problem of efficiently executing tests and checking the results for correctness. Step 5 addresses the *test suite maintenance problem*: the problem of updating and storing test information. Although each of these problems is significant, we restrict our attention to the regression test selection problem. We further restrict our attention to code-based regression test selection techniques, which rely on analysis of P and P' to select tests.

3 Regression Test Selection for Fault Detection

All code-based regression test selection techniques attempt to select a subset T' of T that will be helpful in establishing confidence that P' was modified correctly and that P's functionality has been preserved where required. In this sense, all code-based test selection techniques are concerned, among other things, with locating tests in T that expose faults in P'. Thus, it is appropriate to evaluate the relative abilities of the techniques to choose tests from T that detect faults.

A test t detects a fault in P' if it causes P' to fail: in that case we say t is *fault-revealing for* P'. A program P fails for t if, when P is tested with t, P produces an output that is incorrect according to S. There is no effective procedure by which to find the tests in T that are fault-revealing for P'[32]. Under certain conditions, however, a regression test selection technique can select a superset of the set of tests in T

that are fault-revealing for P'. Under those conditions, such a technique omits no tests in T that can reveal faults in P'.

Consider a second subset of the tests in T: the modification-revealing tests. A test t is *modification-revealing for P and P'* if and only if it causes the outputs of P and P' to differ. Given two assumptions, we can find the tests in T that are fault-revealing for P' by finding the tests in T that are modification-revealing for P and P'. The assumptions are as follows:

> **P-Correct-for-T Assumption.** For each test t in T, when P was tested with t, P halted and produced the correct output.
>
> **Obsolete-Test-Identification Assumption.** There is an effective procedure for determining, for each test in t, whether t is obsolete for P'. Test t is *obsolete* for program P' if and only if t either specifies an input to P' that, according to S', is invalid for P', or t specifies an invalid input-output relation for P'.[3]

To find tests in T that are fault-revealing for P', we run our procedure for identifying obsolete tests in T, and remove them from T. We know that every test remaining in T terminated and produced correct output for P, and is supposed to produce the same output for P'. Thus, the only tests left in T that can be fault-revealing for P' are those that are modification-revealing for P and P', and we can find those fault-revealing tests simply by finding the modification-revealing tests. Also, some tests identified as obsolete may still involve legal inputs to P' that, if used to test P', reveal faults in P'. However, we may find these tests simply by running P' with those inputs, and setting a time bound b such that, if any test exceeds that bound, we assume it is fault-revealing for P'.[4]

Unfortunately, even when the P-Correct-for-T Assumption and the Obsolete-Test-Identification Assumption hold, there is no effective procedure for precisely identifying the nonobsolete tests in T that are modification-revealing for P and P'[32]. Thus, we consider a third subset of T: the modification-traversing tests. A test $t \in T$ is *modification-traversing for P and P'* if and only if it (a) executes new or modified code in P', or (b) formerly executed code that has since been deleted from P.[5]

The modification-traversing tests are useful to consider because with an additional assumption, a nonobsolete test t in T can only be modification-revealing for P and P' if it is modification-traversing for P and P'. The additional assumption is as follows:

[3] If we cannot effectively determine test obsolesence, we cannot effectively judge test correctness. Thus, this assumption is necessary if we intend to employ test reuse, whether selective or not.

[4] This procedure does not, of course, precisely identify the fault-revealing tests from among the obsolete tests in T: it may select a test which, if we increased b, would terminate without being fault-revealing. However, this procedure does conservatively approximate the set of fault-revealing tests from among those obsolete tests, omitting no fault-revealing tests. Notice further that we could apply a similar procedure to all tests in T, running them with a time bound b to discover, conservatively, which tests are fault-revealing for P'; however, in running all tests in T we are doing the very thing that selective retest aims to avoid.

[5] To capture more formally the notion of executing new, modified, or deleted code, in Reference [32] we define the concept of an execution trace $ET(P(t))$ for t on P to consist of the sequence of statements in P that are executed when P is tested with t. We then say that two execution traces $ET(P(t))$ and $ET(P'(t))$, representing the sequences of statements executed when P and P', respectively, are tested with t, are nonequivalent if they have different lengths, or if when we compare their elements from first to last, we find some pair of elements that are lexically nonidentical. We then say that t is modification-traversing for P and P' if and only if $ET(P(i)) \not\equiv ET(P'(i))$.

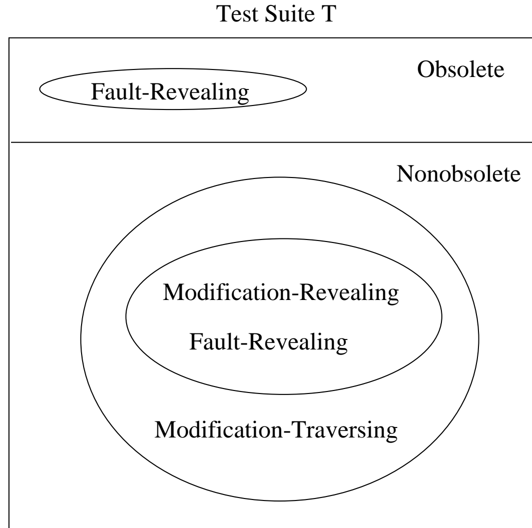

Figure 1: Relationship between classes of tests.

> **Controlled Regression Testing Assumption.** When P' is tested with t, we hold all factors that might influence the output of P', except for the code in P', constant with respect to their states when we tested P with t.

When the Controlled Regression Testing Assumption holds (with our first two assumptions), we can identify the tests in T that are fault-revealing for P and P' by identifying the nonobsolete tests in T that are modification-traversing for P and P', and then using the procedure described above to determine, from among the obsolete tests for T, those tests that are fault-revealing for P and P'. When all three of our assumptions hold, this process locates the tests in T that are fault-revealing for P'. Figure 1 illustrates the relationship that holds between the sets of obsolete, fault-revealing, modification-revealing, and modification-traversing tests in T when the assumptions hold.

Reference [32] gives an algorithm that precisely identifies the tests t in T that are modification-traversing for P and P'. Unfortunately, that algorithm has an exponential worst-case running time. Moreover, we show that unless $P = NP$, we cannot expect to find an efficient algorithm to precisely identify the tests in T that are modification-traversing for P and P', because the problem of precisely identifying those tests is PSPACE-hard. However, even though the problem of precisely identifying the tests in T that are modification-traversing for P and P' is intractable in general, we show that there are algorithms that can conservatively identify those tests. In doing so, for cases where the three assumptions hold, these algorithms select all nonobsolete tests in T that are fault-revealing for P'.

For brevity, we henceforth refer to tests that are "fault-revealing for P'," "modification-revealing for P and P'," and "modification-traversing for P and P'," simply as "fault-revealing," "modification-revealing," and "modification-traversing," respectively.

The Controlled Regression Testing Assumption is not always practical. For example, if we port P to another system, creating P', we cannot use controlled regression testing to test P' on the new system: controlled regression testing demands that we hold all factors other than code constant, including the system. When we do not employ controlled regression testing, selection of the modification-traversing tests may omit modification-revealing tests. For instance, in the porting example, P' may fail on tests that are not modification-traversing for P and P' if the new system has less memory available for dynamic allocation than the old system. There are also other factors that affect the viability of the assumption in practice: nonondeterminism in programs, time-dependencies, and interactions with the external environment can all be difficult (although not necessarily impossible) to incorporate into the testing process in a way that allows us to employ controlled regression testing.[6] However, even when we cannot employ controlled regression testing, the modification-traversing tests may constitute a useful test suite. We conjecture that when a testing budget is limited, and a subset of T must be chosen, tests that execute changed code are better candidates for re-execution than tests that are not modification-traversing.

The three classes of tests that we describe in this section can contribute to the specification of a framework for evaluating and comparing regression test selection techniques, for several reasons. First, testing professionals are reluctant to discard tests that may expose faults. As we have shown, given certain assumptions, the relationships among the three classes provide a way to analytically evaluate test selection techniques in terms of their abilities to select and avoid discarding fault-revealing tests. Despite the fact that many regression test selection techniques aim to select tests to satisfy some test adequacy measure, it is both reasonable and important to evaluate those methods in terms of their abilities to detect faults.

Second, even for cases where the Controlled Regression Testing Assumption cannot be satisfied, the three classes can still serve to distinguish existing regression test selection techniques. Most code-based regression test selection techniques attempt to identify tests that execute changed components of P'. Many techniques attempt to be more precise, eliminating, from those tests that execute changed components, some tests that clearly cannot cause P and P' to produce different output. Thus, it is useful to compare test selection techniques in terms of their abilities to identify these classes of tests.

Third, in practice, for large software systems, test suites are functional, and the goal of testing is not coverage of code components, but testing of functional behavior. When test suites are built primarily to provide code coverage, it may make little sense to select all tests that pass through a component, because only one such test will provide that coverage. But when test suites are functional it seems particularly important not to omit tests from T' that may reveal faults in P', even though they may exercise code components already exercised by other tests.

[6] It is worth worrying, however, about the ramifications for software quality if we do not deterministically test a particular timing effect, environmental interaction, or dependency. In that case, in any particular testing session, some important program behavior may go untested, due simply to the vagaries of the environment or of the order in which operations occur. If we cannot test such behavior deterministically, it is not clear how we can ensure that we test it at all. Rather than concluding that the Controlled Regression Testing Assumption is not useful in practice, perhaps we should focus on finding ways to make controlled regression testing possible for cases like these.

4 Framework for Analyzing Regression Test Selection Techniques

In this section we present our framework for analyzing regression test selection techniques. The framework consists of four categories: inclusiveness, precision, efficiency, and generality.

4.1 Inclusiveness

Let M be a regression test selection technique. *Inclusiveness* measures the extent to which M chooses modification-revealing tests from T for inclusion in T'. We define inclusiveness relative to a particular program, modified program, and test suite, as follows:

> **Definition 1:** Suppose T contains n tests that are modification-revealing for P and P', and suppose M selects m of these tests. The *inclusiveness of M relative to P, P', and T* is (1) the percentage given by the expression $(100(m/n))$ if $n \neq 0$ or (2) 100% if $n = 0$.

For example, if T contains 50 tests of which 8 are modification-revealing for P and P', and M selects 2 of these 8 tests, then M is 25% inclusive relative to P, P', and T. If T contains no modification-revealing tests then every test selection technique is 100% inclusive relative to P, P', and T.

If M always selects all modification-revealing tests we say M is safe, as follows:

> **Definition 2:** If for all P, P', and T, M is 100% inclusive relative to P, P', and T, M is *safe*.

For an arbitrary choice of M, P, P', and T, there is no algorithm to determine the inclusiveness of M relative to P, P', and T[32]; however, we can still draw useful conclusions about inclusiveness. First, we can prove that M is safe by showing that M selects a known superset of the modification-revealing tests. For example, if M selects all modification-traversing tests then for controlled regression testing M is safe. Second, we can prove that M is *not* safe by finding a case for which M omits a modification-revealing test. Third, we can compare test selection techniques $M1$ and $M2$ to each other in terms of inclusiveness by showing that the techniques select subsets Q and R of the modification-revealing tests, and by showing that Q is a subset or superset of R. Finally, we can experiment to approximate the inclusiveness of M relative to a particular choice of P, P', and T. Such experimentation involves running M on P, P', and T to generate set T'. Then we run P' on each test in T to determine which tests in T are modification-revealing. Finally, we compare these tests with the modification-revealing tests in T'.[7]

Inclusiveness and safety are significant measures. If M is safe then M selects every nonobsolete test in T that is fault-revealing for P', whereas if M is not safe it may omit tests that expose faults. Furthermore, we hypothesize that if $M1$ and $M2$ are regression test selection techniques, and $M1$ is more inclusive than $M2$, then $M1$ has a greater ability to expose faults than $M2$.

All code-based regression test selection techniques consider the effects of modified code; however, to evaluate a technique's inclusiveness we must also consider the effects of new and of deleted code. When P' is created by adding new code to P, T may already contain tests that are modification-revealing because of that

[7] Because it is not possible to determine whether P' halts when run on test $t \in T$, such experimentation can only approximate the inclusiveness of M.

```
Fragment F1              Fragment F1'

S1.  if P then           S1.  if P then
S2.    a := 2            S2.      a := 2
S3.  end                 S2a.     b := 3
                         S3.  end
```

Figure 2: Code fragments that illustrate addition of code.

```
Fragment F2                              Fragment F3              Fragment F3'

S1.  call PutTermInGFXMode()             S1.  if P then           S1.  if P then
S2.  call DrawLine(point1,point2)        S2.    (do something)    S2.    (do something)
                                         S3.    a := 2
                                              endif                    endif
Fragment F2'                             S4.  if Q then           S4.  if Q then
                                         S5.    (do something)    S5.    (do something)
S2.  call DrawLine(point1,point2)        S6.    print a
                                              endif                    endif
```

Figure 3: Code fragments that illustrate deletion of code.

code. For example, consider the code fragments shown in Figure 2. In the absence of other modifications, any test that executes statement $S2$ in fragment $F1$ necessarily executes statement $S2a$ in fragment $F1'$ and may be modification-revealing depending on statements it subsequently encounters. Similarly, when P' is created by deleting code from P, T may contain tests that are modification-revealing because of this deleted code. For example, consider the code fragments shown in Figure 3. In the example on the left, statement $S1$ in fragment $F2$ is deleted, yielding fragment $F2'$. In the absence of other modifications, any test that executes $S1$ in $F2$ may produce different output in $F2'$. In the example on the right, two statements are deleted from fragment $F3$, yielding fragment $F3'$. In the absence of other modifications, any test in which both P and Q are true may produce different output in $F3'$. If M does not account for the effects of new and deleted code, we can find examples of code additions or deletions that prove that M is not safe.

4.2 Precision

Let M be a regression test selection technique. *Precision* measures the extent to which M omits tests that are non-modification-revealing. We define precision relative to a particular program, modified program, and

test suite, as follows:

> **Definition 3:** Suppose T contains n tests that are non-modification-revealing for P and P' and suppose M omits m of these tests. The *precision of M relative to P, P', and T* is (1) the percentage given by the expression $(100(m/n))$ if $n \neq 0$, or (2) 100% if $n = 0$.

For example, if T contains 50 tests of which 44 are non-modification-revealing for P and P', and M omits 33 of these 44 tests, then M is 75% precise relative to P, P', and T. If T contains no non-modification-revealing tests, then every test selection technique is 100% precise relative to P, P', and T.

As with inclusiveness, there is no algorithm to determine, for an arbitrary choice of M, P, P', and T, the precision of M relative to P, P', and T[32]; however, we can draw useful conclusions about precision. First, we can compare test selection techniques $M1$ and $M2$ to each other in terms of precision by showing that the techniques select subsets Q and R of the non-modification-revealing tests, and by showing that Q is a subset or superset of R. Second, we can prove that M is not precise by finding a case for which M selects a test that is non-modification-revealing. Third, we can use experimentation to compare relative precisions. Finally, we could prove that M is precise if we could show that M omits a superset of the non-modification-revealing tests.

Precision is useful because it measures the extent to which M avoids selecting tests that cannot produce different program output. In general, when we compare test selection techniques in terms of precision, we can identify the techniques that promote the least unnecessary testing. In particular, when we compare safe test selection techniques in terms of precision, we can identify techniques that come closest to the goal of selecting exactly the modification-revealing tests.

When we evaluate a test selection technique's precision, it is useful to consider procedure `structchange` and modified version `structchange`′ shown in Figure 4.[8] The changes in `structchange`′ create a syntactically different but semantically equivalent modified version: the changes do not affect the output of the program for any inputs. If M selects any tests for this pair of procedures, it is imprecise.

Figure 5 shows another procedure, `pathological`, and a modified version, `pathological`′, that can be useful when we evaluate a test selection technique's precision. Each `while` construct in the two versions first increments the value of `x`, and then tests the incremented value to determine whether to enter or exit its loop. Notice that for a test of input value "0," both versions output value "1", whereas for a test of input value "−2" `pathological` outputs "1" and `pathological`′ outputs "3". If M selects the test of input value "0" it is imprecise.

4.3 Efficiency

We measure the efficiency of regression test selection techniques in terms of their space and time requirements. Where time is concerned, a test selection technique is more economical than the retest-all technique if the cost of selecting T' is less than the cost of running the tests in T-T'[29]. Space efficiency primarily depends on the test history and program analysis information a technique must store. Thus, both space and time

[8] Figure 4 is based on an example suggested by Weibao Wu (private communication).

```
structchange()

S1.  read(x)
S2.  if (x <= 0)
S3.    if (x = 0)
S4.      print(x+2)
S5.      exit
       end
     end
S6.  print(x+3)
S7.  exit
```

```
structchange'()

S1'.  read(x)
S2'.  if (x <= 0)
S3'.    if (x = 0)
S4'.      print(x+2)
S5'.      exit
        else
S6'.      print(x+3)
S7'.      exit
        end
      end
S8'.  print(x+3)
S9'.  exit
```

Figure 4: Procedures `structchange` and `structchange'`.

efficiency depend on the size of the test suite that a technique selects, and on the computational cost of that technique. We rely on standard algorithm analysis techniques to obtain theoretical measurements of efficiency. Where empirical evidence about a technique's efficiency is available, we also rely on that. In the rest of this section, we discuss factors that should be considered when performing such evaluations.

The first factor to consider when we evaluate a test selection technique's efficiency is the phase of the lifecycle in which the technique performs its activities. We distinguish two phases in a typical regression testing life cycle: a preliminary phase and a critical phase. The *preliminary phase* of regression testing begins after release of some version of the software. During this preliminary phase, programmers enhance and correct the software. When corrections are complete, the *critical phase* of the regression testing life cycle begins. In this critical phase, regression testing is the dominating activity; its time is limited — at times severely — by the deadline for product release. It is in this critical phase that cost minimization is most important for regression testing.

Regression test selection techniques can exploit the phases of the regression testing life cycle. For example, a technique that requires test history and program analysis information during the critical phase can achieve a lower critical phase cost by gathering some of that information during the preliminary phase. When we evaluate test selection techniques for efficiency, we differentiate between costs that are incurred during the preliminary phase of regression testing and costs that are incurred during the critical phase.[9]

[9]There are various ways in which this two-phase process may fit into the overall software maintenance process. A *big bang* process performs all modifications, and when these are complete turns to regression testing. An *incremental* process performs

```
pathological(x)
   {
S1.  while (++x < 0)
     {
S2.    while (++x < 0)
       {
S3.      while (++x < 0)
         {}
       }
     }
S4.  printf("%d", x);
   }
```

```
pathological'(x)
   {
S1'.  while (++x < 0)
      {
S2'.    while (++x < 0)
        {}
S3'.    while (++x < 0)
        {}
S4'.    while (++x < 0)
        {}
      }
S5'.  printf("%d", x);
   }
```

Figure 5: Procedures `pathological` and `pathological`$'$.

A second factor to consider when we evaluate a test selection technique's efficiency is its automatability. Human effort is expensive; techniques that require excessive human interaction are impractical. The relative costs of human effort versus machine time may create cases in which, even though test selection requires more time than the retest-all method, test selection is preferable. For example, suppose we require two hours of analysis to determine that we can save one hour of testing. If the analysis is fully automated, the testing requires intensive human interaction, and we can afford the analysis time, then test selection is preferable. Finally, even a test selection method that is efficient in the critical phase may be impractical, if it requires excessive human interaction during the preliminary phase.

A third factor that impacts a test selection technique's efficiency is the extent to which the technique must calculate information on program modifications. A technique that must determine every program component that has been modified in, deleted from, or added to, P, or calculate a correspondence between P and P', may be more expensive than a technique that calculates modification information as needed. A technique that calculates information as needed may find that only a partial correspondence need be calculated, saving work in comparison to a technique that first calculates a complete correspondence.

Regression test selection techniques that require modification information obtain it by one of two approaches. The first approach is to use an incremental editor, that tracks modifications as maintenance programmers perform them. In this case, the cost of obtaining the information is incurred during the preliminary phase of regression testing, instead of in the critical phase. A second approach to obtaining

regression testing at intervals throughout the maintenance life cycle, with each testing session aimed at the product in its current state of evolution. Preliminary phases are typically shorter for the incremental model than with the big bang model; however, for both models, both phases exist, and can be exploited.

modification information is by using a "differencing" algorithm that computes a *correspondence* between P and P', that shows which components in P' are new, which components in P have been deleted, which components in P correspond to which components in P', and which, of these corresponding components, have been modified. One such differencing algorithm is proposed by Yang[44]; this algorithm requires time $O(|P| * |P'|)$ to compute a correspondence between P and P'. An $O(max(|P|, |P'|)^3)$ algorithm is proposed by Laski and Szermer[24]. More efficient comparison methods, such as the UNIX[10] `diff` utility, may not be precise enough for computing correspondences at the intraprocedural level. However, coarser-grained interprocedural test selection algorithms that only need to know, for example, which procedures in P have been modified, can use methods such as `diff`. In this case, a correspondence between P and P' at the procedure level can be calculated in time $O(max(|P|, |P'|) * log(max(|P|, |P'|))$, where the second multiplicand represents the cost of performing table lookups in a directory or configuration management database to locate corresponding procedures.

A final factor affecting the efficiency of test selection techniques concerns the techniques' ability to handle cases in which P' is created by multiple modifications of P. A technique that depends on analyses of programs and processes one modification at a time may be forced to reanalyze or incrementally update analysis or test history information after considering each modification. Such reanalysis can be expensive and can significantly impact a technique's cost. An approach of this sort may suffice in the context of an incremental regression testing process; however, in a big bang process, this expense may be prohibitive.

4.4 Generality

The generality of a test selection technique is its ability to function in a wide and practical range of situations. We describe several factors that must be considered when evaluating a technique's generality.

First, to be practical, a test selection technique should function for some identifiable and practical class of programs. For example, a technique that is defined only for procedures constructed of `if`, `while`, and assignment statement constructs is not practical as defined.

Second, a test selection technique should handle realistic program modifications. For example, a technique that does not handle modifications that alter flow of control in programs is not, in general, practical.

Third, a technique that depends for its success on assumptions about testing or maintenance environments is less general than a technique that requires no such assumptions. For example, a technique that requires that initial testing be performed using dataflow testing criteria is less general than a technique that places no requirements on initial testing. Similarly, a technique that requires an incremental editor to track code modifications is less general than a technique that has no such requirement.

Fourth, a technique that depends on the availability of particular program analysis tools is less general than a technique that does not depend on such tools. For example, a technique that requires collection of test trace information is less general than a technique that does not require this information, because the instrumentation required to collect such traces may be excessively intrusive for certain testing applications. For similar reasons, a method that requires traces on a per function basis is more general than a technique

[10]UNIX is a registered trademark licensed exclusively by Novell, Inc..

that requires traces on a per statement basis.

Finally, a technique may support intraprocedural or interprocedural test selection.[11] In practice, regression testing is often performed at the interprocedural level on subsystems or programs. Furthermore, empirical evidence suggests that test selection at the intraprocedural level may not offer savings sufficient to justify its cost[32].

We could define generality more quantitatively, as we have defined inclusiveness and precision. We could then use experimentation to measure generality with respect to classes of programs and modifications. In this research, however, we have found qualitative comparisons sufficiently informative.

4.5 Tradeoffs

Test selection techniques face tradeoffs where the foregoing criteria are concerned.

First, among safe techniques, increases in precision are typically obtained by increases in analysis. Thus, increasing precision can decrease efficiency. Among techniques that are not safe, increasing either precision or inclusiveness can decrease efficiency. When decreased efficiency drives the cost of analysis above a certain level, it may render the cost of selective regression testing greater than the cost of the retest-all technique.

Second, most of the factors that we have identified as affecting generality also have implications for inclusiveness, efficiency or precision. For example, dataflow-based techniques that do not calculate alias information must make conservative assumptions to handle programs that contain aliases. Attempting to increase the generality of these methods by extending them to handle aliasing decreases their efficiency. Similarly, techniques that handle multiple modifications one at a time incur penalties in efficiency if they must reanalyze programs after considering each modification; if such techniques do not perform reanalysis, however, they can incur penalties in precision.

4.6 Other Definitions of Inclusiveness and Precision

In this paper we define inclusiveness and precision in terms of the modification-revealing tests; however, other definitions may also be useful. For example, we may define inclusiveness and precision in terms of the modification-traversing tests. In this case, inclusiveness measures a technique's ability to select all modification-traversing tests, and precision measures a technique's ability to omit tests that are not modification-traversing. To distinguish these definitions from definitions based on the modification-revealing tests we can use the terms *mr-inclusiveness*, *mr-precision*, and *mr-safety* for the latter, and *mt-inclusiveness*, *mt-precision*, and *mt-safety* for the former.

Similary, we can define inclusiveness and precision in terms of other test selection criteria, such as one used by dataflow test selection techniques. Dataflow techniques attempt to identify tests that exercise new or affected definition-use associations in P'. To evaluate and compare dataflow techniques' relative abilities

[11] Most intraprocedural test selection techniques may be used interprocedurally in a naive fashion, by applying them to all pairs of procedures in the program and its modified version. However, this simplistic approach to interprocedural test selection can be unnecessarily costly[32]. We judge a method interprocedural if it addresses interprocedural test selection by a method that goes beyond this naive approach.

to select tests that exercise affected definition-use pairs (du-pairs), and omit tests that do not, we can use *du-pair-inclusiveness*, *du-pair-precision*, and *du-pair-safety.*

Under all such definitions of inclusiveness or precision the definitions of efficiency and generality that we have presented in this section continue to apply.

Despite the existence of alternative definitions of inclusiveness and precision, we believe that it is particularly important to evaluate test selection techniques in terms of their mr-inclusiveness and mr-precision, because as the discussion in the preceding section shows, those categories support analytical comparisons of methods in terms of their abilities to reveal faults in modified programs.

5 An Analysis of Regression Test Selection Techniques

In this section, we discuss existing regression test selection techniques, and use our framework to analyze them. To illustrate our findings with respect to inclusiveness and precision, we use diagrams like the one shown in Figure 6. In these diagrams, the outer box delimits the set of all nonobsolete tests in T. The three circles, labeled "M", "N", and "D", represent sets of tests that execute modified code in P', new code in P', or code in P that has been deleted from P', respectively. The three circles intersect, because some tests execute two or three classes of code changes. All tests in the sets represented by the circles are modification-traversing; those not in those sets are non-modification-traversing. The dash-filled area in the figure represents the set of modification-revealing tests. Because each category of modification-traversing tests contains some tests that are modification-revealing and some that are not, the dashed area contains, and omits, portions of each area formed by the intersections of the circles.

To depict the inclusiveness and precision of a particular selective retest technique, we shade portions of these diagrams. The shaded area shows the sets of modification-traversing and modification-revealing tests that a technique admits or omits. For example, Figure 7 depicts the inclusiveness and precision of (A) the retest-all technique, and (B) an optimum technique. Because the retest-all technique selects all tests, we shade the entire diagram on the left. An optimum technique, on the other hand, selects exactly the modification-revealing tests; thus, in the diagram on the right we shade only the area delimiting modification-revealing tests.

In a few cases, we use larger shaded areas to indicate that particular techniques are more or less inclusive than other techniques for particular categories of tests. In each of these cases, however, we point out this relationship in the text when we describe the diagram. With the exception of the few cases where relative sizes of shaded areas matter, no importance should be attached to the sizes of the regions or shaded areas in the graphs. Although the diagrams are similar to Venn diagrams, they are not Venn diagrams.

To evaluate time efficiency we have used, where available, the worst-case timing analyses presented by the authors of the papers that present the techniques. In most cases where such analyses were not available, we have performed them ourselves; in some cases where algorithms are not presented in sufficient detail, our analyses are well-considered estimates. To standardize the set of symbols we use in our timing analyses, we use $|P|$, $|P'|$, and $|T|$ to refer to the sizes of P, P', and T, respectively, where by size of a program P we mean the number of statements in the program. To augment our timing analyses, in the few cases where

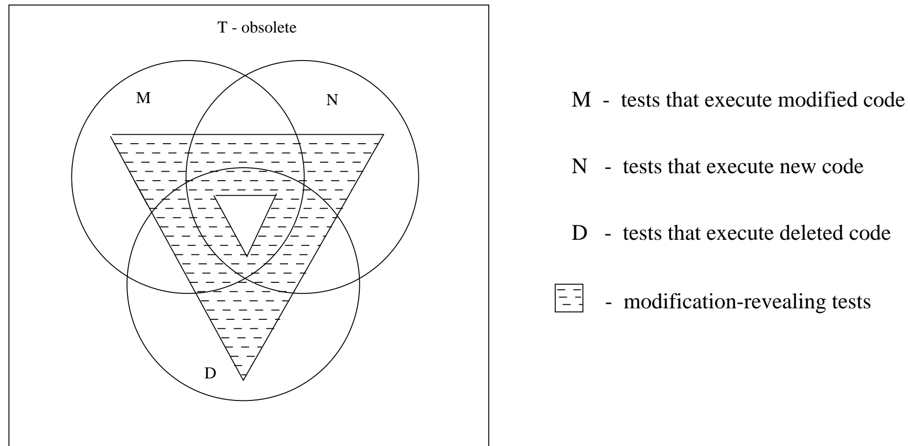

Figure 6: Base diagram for depiction of inclusiveness and precision.

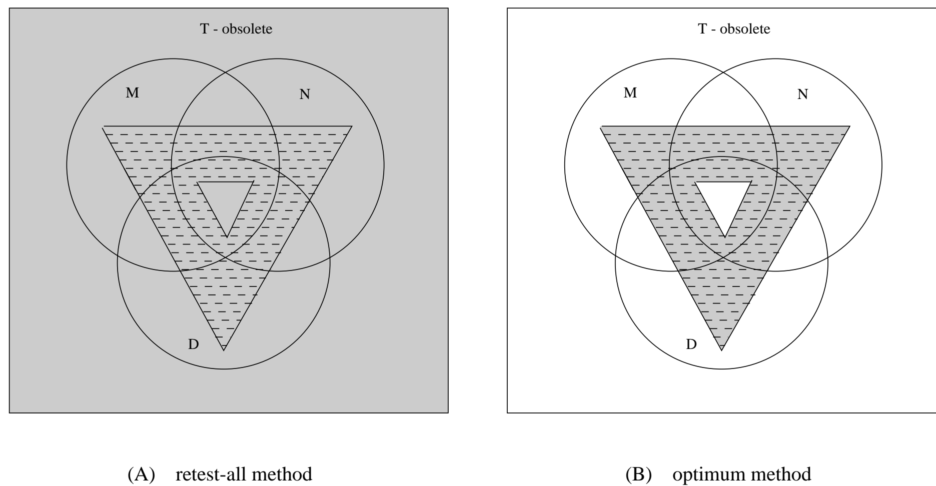

Figure 7: Inclusiveness and precision of retest-all and optimum techniques.

empirical results on the cost of techniques were available, we have discussed those results.

To evaluate space efficiency — a topic seldom discussed in the papers on test selection techniques — we have confined ourselves to mentioning cases in which techniques may require space exponential in the size of their input.

5.1 Linear Equation Techniques

Fischer[10] presents a selective retest technique that uses systems of linear equations to select test suites that yield segment coverage of modified code. Lee and He[25] propose a similar technique. Fischer, Raji, and Chruscicki[11] extend Fischer's earlier work to incorporate information on variable definitions and uses. Hartmann and Robson[18, 19, 20] extend and implement Fischer, Raji, and Chruscicki's technique. Linear equation techniques use systems of linear equations to express relationships between tests and program segments.[12] The techniques obtain systems of equations from matrices that track program segments reached by test cases, segments reachable from other segments, and (optionally) definition-use information about the segments. The intraprocedural techniques use a 0-1 integer programming algorithm to identify a subset T' of T that, if P' contains no modifications that affect control flow,[13] ensures that every segment that is statically reachable from – and optionally every segment that can statically reach – a modified segment is exercised by at least one test in T' that also exercises the modified segment. An interprocedural variant of the techniques treats subroutines as segments; this approach monitors subroutine coverage rather than statement coverage, and supports test selection for programs in which modifications have affected control flow. Both the intraprocedural and interprocedural techniques can use variable definition and use information instead of or in conjunction with control flow information.

In the references just cited, linear equation techniques are presented and discussed as minimization techniques. However, the techniques need not necessarily perform minimization: they can instead select between one and the maximum number of tests traversing the respective coverage entity.

Inclusiveness. Minimization techniques omit modification-revealing tests. If several tests execute a particular modified segment and all of these tests reach a particular affected code segment, minimization techniques select only one such test unless they select the others for coverage elsewhere. Consider, for example, the code fragments and test cases shown in Figure 8, where fragment $F4'$ represents a modified version of fragment $F4$ in which statement $S5$ is erroneously modified. Tests $t3$ and $t4$ both execute statements $S5$ and $S5'$. Test $t3$ causes a divide by zero exception in $S5'$, whereas test $t4$ does not. Minimization techniques select only one of these tests and omit the other; if they select $t4$ they lose an opportunity to expose the fault that $t3$ exposes. Because minimization techniques can omit this modification-revealing test, they are not safe.

[12]Program segments are defined variously in the literature on linear equation techniques. Fischer[10] defines a segment as a single-entry, single-exit block of code whose statements are executed sequentially; by this definition, segments are equivalent to basic blocks. Fischer later applies the term to procedures or functions, in which statements might not be executed sequentially. Hartmann and Robson define segments for C procedures and programs as either particular groups of statements in procedures, or as entire functions, respectively[18]. In all cases, segments are portions of code through which test execution can be tracked, that serve as test requirements or entities to be tested. In our discussion, we use the single term "segment" to refer to all these types of segments, unless it is necessary to distinguish among them.

[13]Modifications that affect control flow include not only changes in predicate statements, but also (for example) changes in assignment statements that alter variables used subsequently in predicate statements.

Fragment F4

```
S1. y = (x-1) * (x+1)
S2. if ( y=0 )
S3.   return(error)
S4. else
S5. return( 1 / y )
```

Fragment F4'

```
S1'. y = (x-1) * (x+1)
S2'. if ( y=0 )
S3'.   return(error)
S4'.else
S5'. return( 1 / (y - 3) )
```

Test Cases

test #	input	execution history
t1	x = 1	S1,S2,S3
t2	x = -1	S1,S2,S3
t3	x = 2	S1,S2,S5
t4	x = 0	S1,S2,S5

Figure 8: Code fragments that illustrate the lack of safety of minimization.

As we have noted, linear equation techniques need not strive for minimization. In this case, although we do not prove this, we believe that the techniques select safe test suites. In their intraprocedural variant, where the techniques only handle situations where code modifications do not alter control flow, minimization techniques can revert to selecting all tests through procedures where such modifications have occured. Interprocedural variants of the techniques have no problem with alterations in control flow, because they just identify modified procedures or functions irrespective of the type of modification.

Precision. Applied to modified procedures for which control flow is not affected, intraprocedural linear equation techniques omit non-modification-traversing tests by ignoring tests that do not execute changed segments. However, when control flow is affected, linear equation techniques are not defined at the intraprocedural level. We interpret this as requiring the user to revert to retest all when control flow is affected. The use of dataflow information can further increase the precision of the techniques. At the interprocedural level, linear equation techniques can select tests that traverse modified procedures, but do not traverse any modified code in those procedures. Such tests are not modification-traversing, so selecting them leads to a loss in precision.

Figure 9 depicts the inclusiveness and precision of linear equation techniques, represented in Diagram (A) as minimization techniques, and in Diagram (B) as interprocedural nonminimization techniques. Diagram (A) illustrates minimization's lack of safety, by leaving areas unshaded within all test categories. Outside of the circles, the shaded area depicts the non-modification-traversing tests selected by the techniques assuming that when procedures contain modified control flow they must select all tests through the procedures. Diagram (B) shows the inclusiveness and precision of linear equation techniques when they do not attempt to minimize the set of tests selected. In this case, the shaded circular areas signify the techniques' selection of all modification-traversing tests, whereas the shaded areas outside the circles signifies the techniques' selection of tests that are not modification-traversing.

Efficiency. Linear equation techniques are automatable. When the techniques operate as minimization techniques, they return small test suites and thus reduce the time required to run the selected tests. However, Fischer states that due to the calculations required to solve systems of linear equations the techniques may be data and computation intensive on large programs[10]. In fact, the underlying problem is NP-hard[12], and all known 0-1 integer programming algorithms may take exponential time. Despite this possible worst-

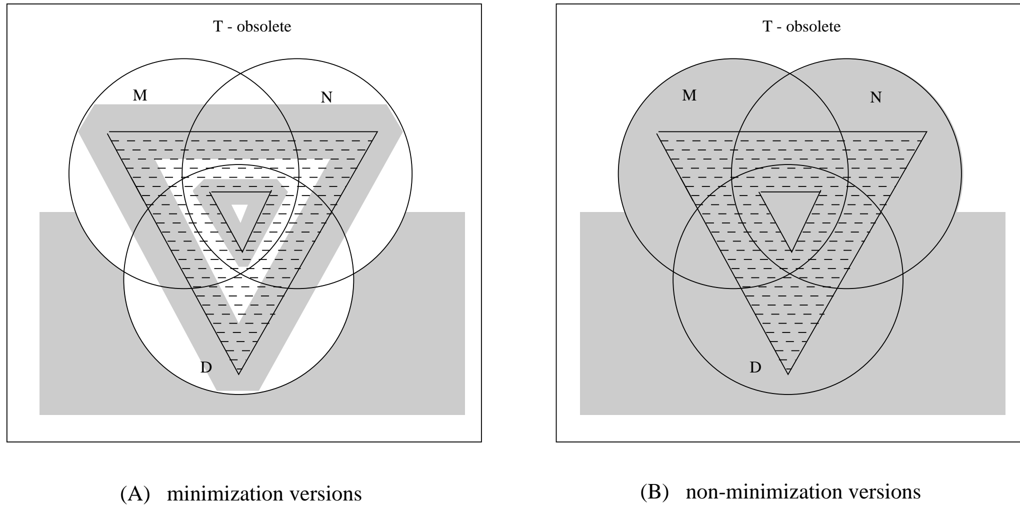

Figure 9: Inclusiveness and precision of linear equation based techniques.

---

case behavior, 0-1 integer programming algorithms exist that can obtain solutions, in practice, in times that may be acceptable. For example, Crowder, Johnson, and Padberg[8] report experimental results in which ten large-scale problems are solved, each in less than an hour. Hartmann and Robson report that their interprocedural technique, by treating functions rather than smaller code segments as the basic entities for coverage, achieves performance gains over intraprocedural techniques. Nevertheless, the references on linear-equation-based test selection techniques do not provide empirical data sufficient to let us evaluate the cost of the techniques in practice.

Both intraprocedural and interprocedural techniques require computation of a correspondence between segments in P to segments in P', and of which segments have been modified, after testing has entered its critical phase. The references on linear equation methods do not specify a method by which correspondence and change information should be computed. We assume, however, as discussed in Section 4, that the required information can be computed for the intraprocedural techniques in time $O(|P| * |P'|)$. For the interprocedural technique, the information can be computed in time $O(max(|P|, |P'|) * log(max(|P|, |P'|))$. The techniques handle multiple modifications in a single application of the algorithm.

Linear equation techniques require transitive closure operations on relations of size $O(|P|)$ to determine static reachability between segments; such operations require worst-case time $O(|P|^3)$. However, these operations can be completed during the preliminary phase of regression testing.

Generality. Although presented only for Fortran and C, linear equation techniques can be implemented for any procedural language. Both intraprocedural and interprocedural versions of the technique are defined. The intraprocedural technique is defined only for modifications that affect flow of control. The interprocedural technique handles all types of program modifications. The techniques are independent of underlying

coverage criteria but are aimed, in general, for use with control flow or dataflow testing criteria. The techniques require tools for solving 0-1 integer programming problems, and for collecting test trace information at either the function level, or some intraprocedural segment level.

5.2 The Symbolic Execution Technique

Yau and Kishimoto[45] present a selective retest technique that uses input partitions and data-driven symbolic execution to select and execute regression tests. Initially, the technique analyzes code and specifications to derive the input partition for a modified program. Next, the technique eliminates obsolete tests, and generates new tests to ensure that each input partition class is exercised by at least one test. Given information on where code has been modified, the technique determines edges in the control flow graph for the new program from which modified code is reachable. The technique then performs data-driven symbolic execution, using the symbolic execution tree to symbolically execute all tests. When tests are discovered to reach edges from which no modifications are reachable, they need not be executed further. Tests that reach modifications are symbolically executed to termination. The technique selects all tests that reach new or modified code. However, the technique also symbolically executes these selected tests, obviating the need for their further execution.

Inclusiveness. The symbolic execution technique selects all tests that execute new or modified code in the modified program. The technique also selects tests that reach blocks in the original program in which statements are deleted. However, if entire blocks of code are deleted by removing control statements, the symbolic execution technique does not detect tests affected by such deletions because it selects only tests that reach modified or new code actually present in the modified program. Because such tests may be modification-revealing, the symbolic execution technique is not safe.

Precision. The symbolic execution technique selects only tests that reach new or changed blocks of code. In most cases, this approach omits non-modification-traversing tests; however, the presence of a new block of code does not necessarily render tests through that block modification-traversing. For example, in the programs shown in Figure 4, tests that take the false branch from $S3$ in `structchange` enter a new block of code when they take that branch in `structchange`′; however, these tests execute identical sequences of statements in the two program versions, and are thus non-modification-traversing.

Figure 10 depicts the inclusiveness and precision of the symbolic execution technique. The shaded area indicates the technique's selection of all tests that execute modified or new code. The area corresponding to tests that execute only deleted statements is partially unshaded, indicating the technique's omission of some such tests. Because the technique admits some non-modification-traversing tests, some areas outside the circles are shaded.

Efficiency. Yau and Kishimoto state that their symbolic execution technique is computationally expensive and may produce unmanageably large amounts of data. The symbolic expressions built during execution can be more complex than the program from which they are derived. In fact, the symbolic execution tree built by the technique can have a size (and require a time to build) that is exponential in the size of the modified program; this may result in symbolic expressions that are exponential in the size of the modified

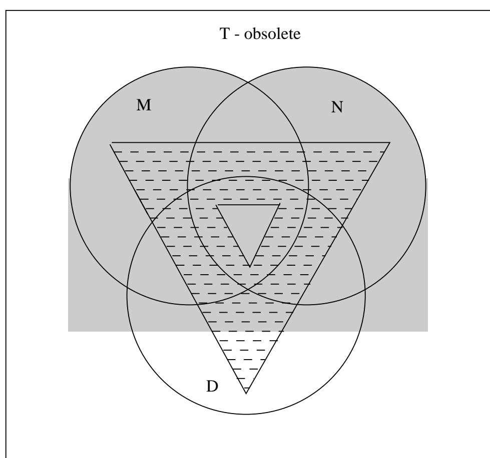

Figure 10: Inclusiveness and precision of the symbolic execution technique.

program. Finally, it is possible that for certain modifications the algorithm will not terminate. This can occur, for example, if a loop predicate is modified such that the modification causes the program to loop forever on one of its test inputs.

The symbolic execution technique requires prior calculation of the location of new or modified code in the modified program; as discussed in Section 4 this can require time $O(|P| * |P'|)$. The technique handles multiple modifications in a single application of the algorithm.

Generality. The symbolic execution technique handles modifications that affect control flow, but does not handle code deletions. The technique applies to both modified procedures and programs. Although the technique is presented as part of a testing approach that makes use of input partition testing, the test selection algorithm itself does not depend on the use of any specific testing criteria. Nevertheless, the technique has limited generality because of its cost: the authors conclude that, due to the computational expense of symbolic execution, their technique is feasible only for numerical programs that are not inordinately complex. The technique does not require test traces.

5.3 The Path Analysis Technique

Benedusi, Cimitile, and De Carlini[3] present a selective retest technique based on path analysis. Their technique takes as input the set of program paths in P' expressed as an algebraic expression, and manipulates that expression to obtain a set of cycle-free *exemplar paths*: acyclic paths from program entry to program exit. The technique then compares exemplar paths from P to exemplar paths from P', and classifies paths as new, modified, canceled, or unmodified. Next, the technique analyzes tests to see which exemplar paths they traverse in P. The technique selects all tests that traverse modified exemplar paths.

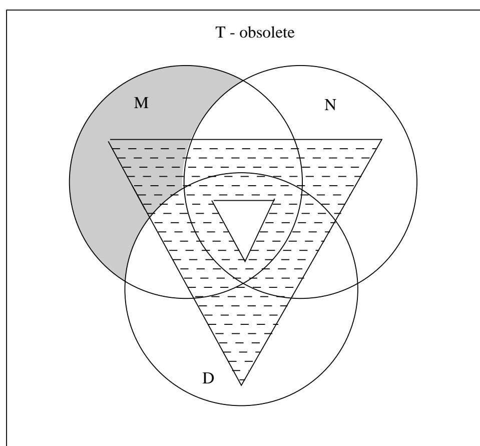

Figure 11: Inclusiveness and precision of the path analysis technique.

Inclusiveness. The path analysis technique omits tests that traverse canceled paths, and omits tests that traverse new paths (paths that contain new blocks of code). In either case, the omitted tests may be modification-revealing. For example, the technique omits a test of `pathological` that uses input "-2". Because such a test is modification-revealing for `pathological` and `pathological`$'$, the path analysis technique is not safe.

Precision. The path analysis technique selects only tests that execute modified exemplar paths; such tests are necessarily modification-traversing. Thus, the technique omits non-modification-traversing tests. However, the technique does not omit any non-modification-revealing tests that execute modified exemplar paths.

Figure 11 depicts the inclusiveness and precision of the path analysis technique. The shaded area indicates the technique's selection of all tests that traverse paths that contain only modifications. The areas corresponding to other tests are unshaded, indicating the technique's omission of such tests.

Efficiency. The path analysis technique does not need to compute a correspondence between a program and its modified version – at least, not in the same sense in which other techniques compute a correspondence. Instead, the technique compares exemplar paths to locate modifications.[14] The technique also handles multiple modifications in a single application of the algorithm. However, the path analysis technique is computationally expensive. The technique calculates and stores exemplar paths for P, P', and each test in T. The number of exemplar paths in P and P', on which both computation and data usage depends, may be exponential in $|P|$ or $|P'|$.

Generality. The path analysis technique assumes the use of a programming environment in which low-

[14]The computation and comparison of exemplar paths, however, can be thought of as a computation of a correspondence.

level program designs are depicted by language-independent algebraic representations. Such an environment facilitates construction of algebraic expressions that may be manipulated to yield a set of exemplar paths. The technique does not handle test selection for additions or deletions of code. The technique does not support interprocedural regression testing beyond the approach of analyzing all procedures in a program. The technique does not require the use of any particular coverage criteria or test generation technique, but does require a tool for collecting traces at the statement level.

5.4 Dataflow Techniques

Several selective retest techniques are based on dataflow analysis and testing techniques. Dataflow test selection techniques identify definition-use pairs that are new in, or modified for, P', and select tests that exercise these pairs. Some techniques also identify and select tests for definition-use pairs that have been deleted from P. Two overall approaches have been suggested. Incremental techniques process a single change, select tests for that change, incrementally update dataflow information and test trace information, and then repeat the process for the next change. Nonincremental techniques process a multiply-changed program considering all modifications simultaneously. The dataflow regression testing techniques described by Gupta, Harrold, and Soffa[13], Harrold and Soffa[15, 16, 17], Taha, Thebaut and Liu[39], and Ostrand and Weyuker[31], are sufficiently alike to justify treating them together.

Inclusiveness. Dataflow techniques consider tests only in association with definition-use pairs. As a result, they can omit modification-revealing tests in several ways. For example, for code deletions like the one depicted on the left in Figure 3, dataflow techniques do not select any tests. Similarly, if a test executes a new or modified output statement that contains no variable uses, dataflow techniques might not select this test even though the statement may be modification-revealing for the old and new versions of the program. Both incremental and nonincremental techniques can omit tests in these ways. Thus, dataflow techniques are not safe.

Precision. By selecting only tests that that execute new, modified or deleted definition-use pairs, dataflow techniques typically omit non-modification-traversing tests. However, the presence of a definition or use in a new block of code does not always render tests through that block modification-traversing. For example, for the program of Figure 4, definition-use pair ($S1'$:x,$S6'$:x) is new, but tests that exercise this pair are non-modification-traversing. Dataflow techniques that attempt to identify tests through such pairs are imprecise for cases such as this. On the other hand, by requiring selected tests to exercise new or modified definition-use pairs, dataflow techniques omit tests, such as tests that reach a modified definition but reach no use of the defined variable, that are modification-traversing but non-modification-revealing.

Figure 12 illustrates the inclusiveness and precision of dataflow techniques. Because the techniques achieve an increase in precision by selecting only tests that exercise definition-use pairs, part of the area of the diagram that corresponds to modification-traversing tests that are non-modification-revealing is unshaded. However, because the techniques miss some modification-revealing tests, areas within the modification-revealing test area are also unshaded, and because the techniques can select some non-modification-traversing tests, some of the area outside the circles is shaded.

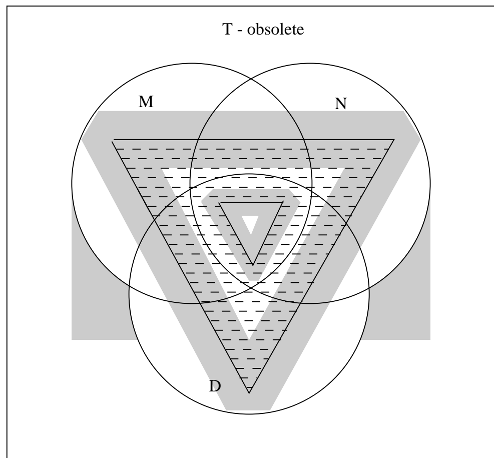

Figure 12: Inclusiveness and precision of dataflow techniques.

Efficiency. Dataflow techniques require initial calculation and storage of dataflow information. Incremental dataflow test selection techniques must perform incremental dataflow analysis and update the dataflow information. The worst-case cost of test selection for such techniques per modification is $O(|T| * |P'|^2)$[39]. The worst-case cost for nonincremental techniques is slightly larger than this. For nonincremental techniques, reanalysis of P' requires $O(|P'|^2)$ time, and computation of a correspondence between the versions can be computed in time $O(|P| * |P'|)$. However, comparison of definition-use associations can require $O((max(|P|, |P'|))^2 * log(max(|P|, |P'|))^2)$ time, because the number of definition-use associations in a program P may be quadratic in $|P|$. Thus the worst-case cost of nonincremental techniques is $O(T * (max(|P|, |P'|))^2 * log(max(|P|, |P'|))^2)$.

Generality. Dataflow techniques require only control flow graphs and test execution histories, and thus can be applied to procedural programs generally. The techniques function for all varieties of program changes except those that do not alter definition-use associations. Taha, Thebaut, and Liu's technique, and Ostrand and Weyuker's technique, apply to intraprocedural regression test selection; Harrold and Soffa's technique applies to interprocedural test selection. The techniques assume the initial use of dataflow test selection criteria, and require tools for static dataflow analysis and for collecting test traces at the basic block level. The incremental approach also requires incremental dataflow analysis tools.

5.5 Program Dependence Graph Techniques

Bates and Horwitz[2] present test selection techniques based on the program dependence graph (PDG) criteria: all-PDG-nodes and all-PDG-flow-edges. PDG techniques use slicing to group PDG components (nodes or flow edges) in P and P' into *execution classes*, such that a test that executes any component in

```
Fragment F4          Fragment F4'

S1. if P = 1         S1. if P = 1
S2.    x := 2        S2'.   x := 3
S3. if Q = 1         S3. if Q = 1
S4.    y := x        S4.    y := x
```

Test Cases

test #	input	execution history
t1	P=1,Q=1	S1,S2,S3,S4
t2	P=0,Q=1	S1,S3,S4

Figure 13: Code fragments that distinguish modified and affected statements.

---

an execution class executes all components in that class. Next, the techniques identify components that may exhibit different behavior in P' than in P (*affected components*) by comparing slices of corresponding components in P and P'. Finally, the techniques select all tests that exercise components that are in the same execution class as an affected component.

Inclusiveness. Although we do not prove this, we believe that PDG techniques identify all tests that execute new or modified code. In the presence of multiple changes, the techniques may fail to recognize that some test t will reach a particular component c in P', due to the presence of some other changed statement in the slice from c. In that case, however, they select t as necessary with respect to some other component. Nevertheless, PDG techniques can omit tests that exercised statements that are deleted from P. For example, in both cases presented in Figure 3, the techniques select no tests. Thus, PDG techniques are not safe.

Precision. The technique of selecting tests through affected nodes causes the all-PDG-nodes technique to select non-modification-traversing tests. For example, applied to the code fragments of Figure 13, the all-PDG-nodes technique correctly selects test $t1$, but also selects non-modification-traversing test $t2$ because $t2$ executes statement $S4$, which is affected by the change to $S2$. This problem does not occur with the all-PDG-flow-edges criteria. However, neither technique is precise with respect to modifications of the sort depicted in Figure 4, because both techniques select tests that reach new PDG components and tests that reach $S6$ (or edge $(S6', S7')$) in `structchange'`, due to equivalence of slices back from $S8$ (or edge $(S8, S9)$).

Figure 14 depicts the inclusiveness and precision of PDG techniques. Because both techniques admit non-modification-traversing tests, some areas outside the circles are shaded. Because both techniques miss some tests through deleted statements, areas of modification-revealing tests remain unshaded.

Efficiency. PDG techniques compute control slices for every node or flow edge in the PDG for P and every node or flow edge in the PDG for P'. They then compute backward slices on each node or flow edge in P that has a corresponding node or flow edge in P', and each node or flow edge in P' that has a corresponding node or flow edge in P. Thus, the all-PDG-nodes techniques compute $O(|P| + |P'|)$ slices, and the all-PDG-flow-edges techniques compute $O(|P|^2 + |P'|^2)$ slices. Each slice can require time quadratic in procedure size. Adding in the cost of performing set operations on test suites of size t, all-PDG-nodes techniques have worst-

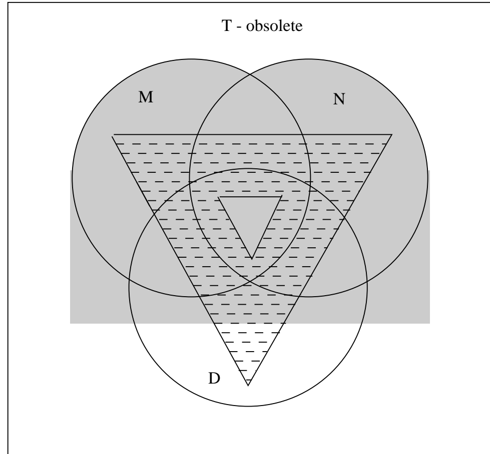

Figure 14: Inclusiveness and precision of PDG techniques.

---

case time $O(|T| * (max(|P|, |P'|))^3)$, and all-PDG-flow edges has worst-case time $O(|T| * (max(|P|, |P'|))^4)$. Furthermore, both slice computation on P' and slice comparisons must be performed in the critical phase of regression testing, after modifications to P are complete. PDG techniques handle multiple modifications in a single application of the algorithm. Finally, both PDG techniques require computation of a complete correspondence between statements in P and their modified versions in P', provided by either a mapping algorithm or an incremental editor.

Generality. PDG techniques are presented only for a restricted set of language constructs; however, they should apply to procedural languages generally. The techniques address all types of code modifications except for code deletion. The techniques do not support interprocedural regression testing beyond the approach of analyzing all procedures in a program. The techniques assume the use of PDG-based test adequacy criteria, and require tools for constructing PDGs (which in turn require tools for performing control dependence and dataflow analysis), tools for performing program slicing, and tools for collecting test traces at the statement level.

5.6 System Dependence Graph Techniques

Binkley[5] presents a technique for interprocedural regression test selection that operates on the system dependence graph (SDG). Given program P and modified version P', SDG techniques use *calling context slicing* — a slicing technique for calculating precise interprocedural slices — on SDGs for P and P', to identify components (vertices or flow edges) in P and P' that have *common execution patterns*. The techniques identify new, preserved, deleted, and affected components in P', where affected components are components in P' that differ from their corresponding components in P, or for which the calling context slice contains

components that are not in P. The techniques select tests that exercise components in P that have common execution patterns with respect to new or affected components in P'.

Inclusiveness. Although we do not prove this, we believe that SDG techniques, like PDG techniques, identify all tests that execute new or modified code. However, like PDG techniques, SDG techniques can omit tests that exercised components deleted from P. For both cases presented in Figure 3, the techniques select no tests. Thus, SDG techniques are not safe.

Precision. The SDG all-vertices technique is more precise than the PDG all-vertices technique, because it avoids selecting tests that execute only affected components – tests that are non-modification-traversing. For example, when applied to the code fragments in Figure 13, the all-SDG-nodes technique selects test $t1$ and omits test $t2$, because $t2$ does not execute any modified statements. Like the PDG techniques, however, SDG techniques admit non-modification-traversing tests for cases such as that of Figure 4.

Figure 15 depicts the inclusiveness and precision of SDG techniques. Because the techniques may admit non-modification-traversing tests, areas outside the circles are shaded. Like PDG techniques, however, both techniques miss some modification-revealing tests through deleted statements, so an area corresponding to such tests is unshaded.

Efficiency. SDG techniques can require $O(|P| + |P'|)$ or $O(|P|^2 + |P'|^2)$ slices, for the all-PDG-nodes and all-PDG-flow-edges versions, respectively. Each slice can require time linear in the size of the SDG. SDG size is polynomial in a number of factors relating to program size, including number of parameters, procedure size, and number of call sites[23]: this polynomial is at least of degree two. Adding in the cost of performing set operations on test suites of size $|T|$, SDG techniques have a worst-case time of at least $O(|T| * (max(|P|, |P'|))^3)$ or $O(|T| * (max(|P|, |P'|))^4)$, depending on the criteria in use. SDG slice computation and slice comparisons are performed with respect to statements in P', so the costs of these operations on P' are incurred after modifications are complete, when testing has entered the critical phase. SDG techniques handle multiple modifications with a single application of their algorithm. The techniques require provision of a complete correspondence between statements in P and their modified versions in P', provided by either a mapping algorithm or an incremental editor. This correspondence must be computed after testing has entered the critical phase.

Generality. SDG techniques apply to procedural languages generally. The techniques address all types of program modifications except for code deletions. The techniques specifically address the problem of interprocedural test selection, but should also function for intraprocedural test selection. The techniques require tools for constructing SDGs (which in turn require tools for performing control dependence and dataflow analysis), and for collecting test traces at the statement level. The techniques assume the use of PDG-based test adequacy criteria.

5.7 The Modification Based Technique

Sherlund and Korel[37] present a selective retest technique that uses static dependence analysis to determine program components that are data or control dependent on modified code, and thus may be affected by a modification. For each of several types of program modifications, the technique specifies a set of program

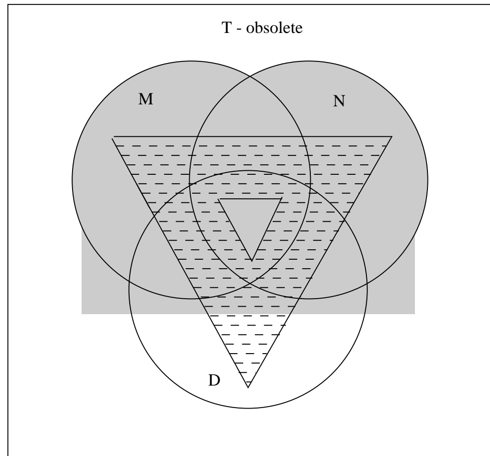

Figure 15: Inclusiveness and precision of SDG techniques.

---

components that can be *influenced* by that modification. The technique instruments the modified program, and requires a tester to run tests from T on the instrumented source. As each test is executed, the technique performs dynamic dependence analysis on its test execution trace, to determine whether the test executed the modified code, and if so, which influenced components it then reached. Testing is complete for the modification when each influenced component has been reached by some test that exercised the modification. In Reference [38], the authors extend the work to handle *logical modifications*, which consist of groups of logically related modifications.

The modification based technique differs from the other techniques we analyze, in that it does not automate the process of selecting T' from T. Instead, the technique identifies coverage requirements; the process of selecting a T' that helps satisfy these coverage requirements is left to the tester. The authors indicate that future work will address the problem of guiding the tester in that selection.

Despite the difference between the modification based technique and techniques that automate the selection of T', it is useful to evaluate the technique alongside other regression test selection techniques. Even though the technique does not automatically select T', the goal of the technique is to let the tester cease testing after having run some subset T' of the tests in T.

Inclusiveness. The modification based technique is a minimization technique. As such, the technique omits modification-revealing tests, because it attempts to select, for each modified code component, only one test that reaches each component affected by that modification. The technique may omit other tests that execute the same pair of modified and affected code components, and thereby omit tests that expose faults. For example, for the code fragments and test cases shown in Figure 8, the technique will allow the tester to cease testing after selecting only one of tests $t3$ and $t4$. A tester who selects $t4$ loses an opportunity to

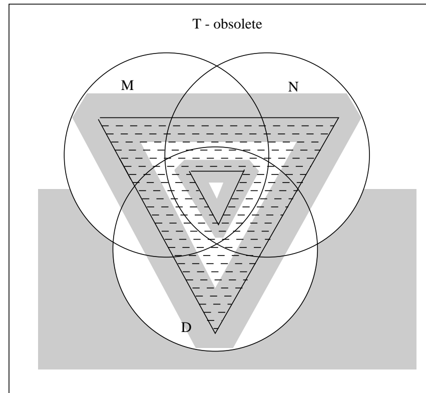

Figure 16: Inclusiveness and precision of the modification based technique.

---

expose the fault that $t3$ exposes.

The modification based technique is not defined for all types of modifications; this too affects the inclusiveness of the technique. For example, it is not clear from the references how the technique handles deletions of procedure calls when those calls do not influence variables, as depicted on the left side in Figure 3.

Sherlund and Korel do not propose a non-minimization version of their technique. If they chose to require selection of multiple tests (rather than just one test) that reach an affected component from a modified component, their technique could still omit modification-revealing tests. For example, because of its restriction that selected tests must reach code that is dependent on modified code, the technique can omit tests that reach $S2$ from $S1$ in the code fragments on the left in Figure 3. Thus, even as a non-minimization technique, the technique is not safe.

Precision. The modification based technique does not require testers to select non-modification-traversing tests, because the techniques count only tests that actually execute modified code toward coverage. Moreover, by using dependence analysis, the technique avoids requiring some tests that, although modification-traversing, are non-modification-revealing. Nevertheless, because the technique depends on a person to locate tests that cover testing requirements, it is likely that in practice, the set of tests T' which that person selects to try to meet the criteria will include non-modification-traversing tests.

Figure 16 depicts the inclusiveness and precision of the modification based technique. The technique may leave unexecuted tests in all categories, and thus areas in all categories remain unshaded. Using the technique, non-modification-traversing tests may inadvertently be run; thus, areas outside the circles are shaded.

Efficiency. As a minimization technique, the modification based technique can be satisfied by selection of small test sets; this can reduce testing time. However, the technique does not at present automate the test

selection process; a person must select and execute tests until the testing requirements for a modification have been met. We cannot quantify the time this process may require. Moreover, the technique assumes knowlege of all code modifications. After the technique analyzes a particular modification or logical modification, and tests have been found that cover that modification, the technique must redo or incrementally update its static analysis before it can consider the next modification. The required static analysis, which includes static data and control dependence analysis, requires $O(|P'|^2)$ time. After running each test, the technique requires dynamic analysis of the trace for that test; this too can require time $O(|P'|^2)$. Thus, for each logical modification the technique requires $O(|T| * |P'|^2)$ time.

Generality. The modification based technique applies to procedural languages generally. The technique is defined for many, but not all, types of program modifications. The technique can apply interprocedurally or intraprocedurally. The technique does not depend on any particular testing criteria, but requires a tool for collecting test traces at the statement level, and tools for performing static and dynamic control and data dependence analysis.

5.8 The Firewall Technique

Leung and White[28] present a selective retest technique directed specifically at interprocedural regression testing that handles both code and specification changes. Their technique determines where to place a *firewall* around modified code modules. Where test selection from T is concerned, the technique selects unit tests for modified modules that lie within the firewall, and integration tests for groups of interacting modules that lie within the firewall. Leung and White[27, 41] extend their technique to handle interactions involving global variables. White et al. [42] discuss experiences implementing the firewall technique.

Inclusiveness. When the unit and integration tests initially used to test system components are *reliable*, such that correctness of modules exercised by those tests for the tested inputs implies correctness of those modules for all inputs, the firewall technique selects all modification-revealing tests, and is safe. As Leung and White note, however, in practice, test suites are typically not reliable. When test suites are not reliable, the firewall technique may omit modification-revealing tests. To see how this may happen, suppose $\mathcal{P}$ is a modified procedure in program P, let T be the set of tests for P, and let $T_{\mathcal{P}}$ be the set of unit and integration tests that apply to modules within the firewall drawn around $\mathcal{P}$. If $T_{\mathcal{P}}$ is not reliable for $\mathcal{P}$, then there may exist some input $i \in D(\mathcal{P})$, such that no test in $T_{\mathcal{P}}$ exercises $\mathcal{P}$ with input i, and such that input i exposes a fault in $\mathcal{P}$. If some system test $t \in T$ exists, such that $t \notin T_{\mathcal{P}}$, and such that t causes $\mathcal{P}$ to be invoked with input i, then t is fault-revealing for P, but the firewall technique does not select t. It follows that in practice, the firewall technique is not safe. The authors state that despite this fact, their technique "provides a sensible utilization of testing resources"[28].

Precision. The firewall technique selects all unit and integration tests of modules that lie within the firewall. Because not all of these tests necessarily execute modified code, the firewall technique selects non-modification-traversing tests. For the programs and modified versions of Figures 4 and 5, for example, the technique selects all tests, even though some are non-modification-traversing.

Figure 17 depicts the inclusiveness and precision of the firewall technique. Because the technique omits

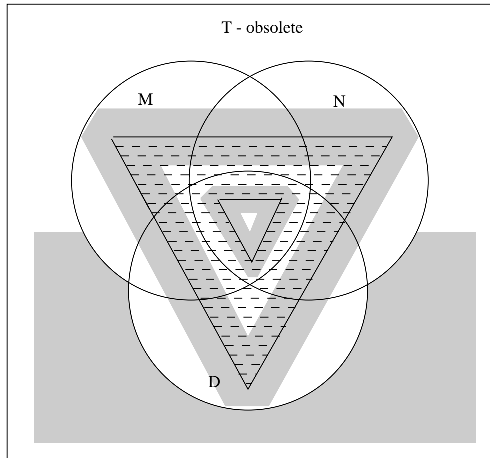

Figure 17: Inclusiveness and precision of the firewall technique.

---

modification-revealing tests in all categories, the diagram contains unshaded areas in all categories. Because the technique admits some non-modification-traversing tests, some areas outside the circles are shaded.

Efficiency. The firewall technique is not described in sufficient formal detail to support a precise analysis of its worst-case running time. We believe, however, that a firewall can be constructed in time proportional to the size of a program's call graph, and that tests can be selected for the firewall in time proportional to the product of the size of the test suite and the size of the call graph. In the worst case, call graph size is proportional to program size; thus, the firewall technique requires time $O(max(|P|, |P'|) * |T|)$ to perform test selection. The technique handles multiple modifications in a single application of its algorithm. The technique also requires computation of the set of modified procedures during the critical phase of testing; as discussed in Section 4, such computation can be performed in time $O(max(|P|, |P'|) * log(max(|P|, |P'|)))$.

The firewall technique has been implemented, and initial measurements of its expense have been reported[42]. The implementation requires a database that may be expensive to set up; however, this setup can be performed during the initial phase of regression testing. Preliminary empirical results suggest that once setup is complete, the analysis and test selection phases of the technique are efficient for large amounts of data.

Generality. The firewall technique is applicable to programs in procedural languages generally. The technique handles all types of code modifications. The technique specifically handles interprocedural test selection, although it does not handle intraprocedural test selection. The technique does not require the use of any underlying testing technique or any particular coverage criteria. The technique does require tools for collecting test traces at the function level. Finally, although in this work we focus on code-based testing needs, it is worth noting that the firewall technique also addresses testing needs with respect to specification changes.

```
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	input
t1	empty file
t2	-1
t3	1 2 3

Figure 18: Procedure illustrating imprecision in the cluster identification technique.

5.9 The Cluster Identification Technique

Laski and Szermer[24] present a technique for identifying single-entry, single-exit subgraphs of a control flow graph (CFG), called *clusters*, that have been modified from one version of a program to the next. The cluster identification technique computes control dependence information for a procedure and its changed version, and then computes the control scope of each decision statement in the procedure by taking the transitive closure of the control dependence relation. The technique uses this information to identify clusters and establish a correspondence between the CFGs of P and P'. While establishing this correspondence, the technique selects tests that execute new, deleted, and modified clusters.

Inclusiveness. The cluster identification algorithm is not presented in sufficient formal detail to support a proof that it is safe; however, we believe the technique is safe for controlled regression testing. The technique handles structural and nonstructural changes, and new and deleted code, by identifying clusters in which structures or code have been added, modified, or deleted, and selecting all tests that exercise the corresponding clusters in P. We believe that this procedure identifies a superset of the modification-traversing tests. For all example programs and code fragments presented in this and the previous sections of this paper, the technique identifies and selects such a superset.

Precision. The cluster identification technique can identify clusters in a manner that allows selection of non-modification-traversing tests. For example, given procedure `avg` of Figure 18, and the test suite for `avg` shown in the figure, the cluster identification technique identifies a cluster consisting of statements $S3$ through $S10$, but does not identify smaller clusters within that cluster. If we add statement "`S5'. print('Improper input.')`" to `avg` just prior to statement $S5$, the cluster identification technique selects tests $t1$, $t2$ and $t3$, because all three tests exercise the modified cluster that encloses the new line. However, only test $t2$

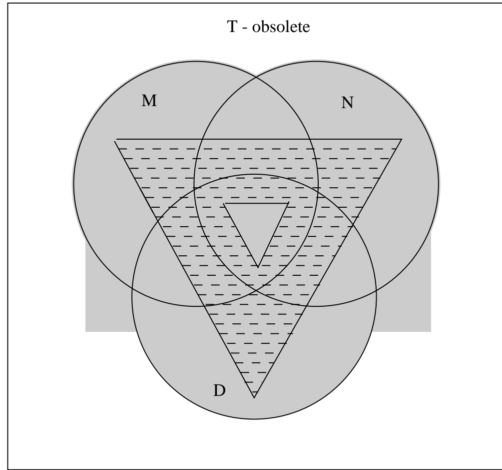

Figure 19: Inclusiveness and precision of the cluster identification technique.

---

actually executes the new statement; tests $t1$ and $t3$ are non-modification-traversing. As a second example, for the procedures of Figure 4, the cluster identification technique selects non-modification-traversing tests that enter the cluster that contains statements $S6'$ and $S7'$. Note, however, that the technique selects exactly the modification-traversing tests for procedures `pathological` and `pathological`$'$ of Figure 5.

The cluster identification technique looks inside modified procedures, and may omit non-modification-traversing tests that go through those procedures on recognizing that those tests do not execute modified clusters. Thus, it is more precise than interprocedural linear equation techniques, which select all tests through modified procedures.

Figure 19 depicts the inclusiveness and precision of the cluster identification technique, given our assertion that the technique selects all modification-traversing tests. Because the technique is safe, the circles in the diagram are fully shaded. Because the technique can admit non-modification-traversing tests, some areas outside the circles are shaded. However, because the technique is more precise than interprocedural linear equation techniques, that shaded area is smaller in this diagram than in diagram (B) of Figure 9.

Efficiency. The running time of the cluster identification technique is bounded by the time required to compute the control scope of decision statements, which is $O(n^3)$ for procedures of n statements. The algorithm for establishing a correspondence between clusters is quadratic in the size of the larger of P and P'. When dealing with test suites of $|T|$ tests, the technique requires $O(|T| * (max(|P|, |P'|))^3)$ time to select tests. The technique handles multiple modifications in a single application of the algorithm. However, the technique computes a correspondence for the entire procedure and modified version, and it performs this computation after modifications are complete, when testing is in the critical phase.

Generality. Because the cluster identification technique works on CFGs, it applies to procedural pro-

grams generally. Moreover, the technique handles all forms of program modifications. The technique does not support interprocedural regression testing beyond the approach of analyzing all procedures in a program. The technique makes no assumptions about development environments or initial design of test suites. The technique requires tools for calculating control dependence and for collecting test trace information at the statement level.

5.10 Slicing Techniques

Agrawal et al. [1] define a family of selective retest techniques that use slicing. For each test $t \in T$, each technique constructs a slice. The authors discuss four different slice types: execution slice, dynamic slice, relevant slice, and approximate relevant slice. An *execution slice* for t contains exactly the statements in P that were executed by t. A *dynamic slice* for t contains all statements in the execution slice for t that have an influence on an output statement in the execution slice. A *relevant slice* for t is like the dynamic slice for t, except that it also contains predicate statements in t that, if changed, may cause P to produce different output, and statements in t on which these predicates are data dependent. Finally, an *approximate relevant slice* for t is like the dynamic slice for t, except that it also contains all predicate statements in the execution slice for t. Given slice sl for test t, constructed by one of the four slicing techniques, if sl contains a modified statement, the techniques select t.

Inclusiveness. As Agrawal et al. show, the dynamic slice technique can omit modification-revealing tests when P contains modified predicate statements, so the technique is not safe. When code modifications do not alter control flow graph edges or definition sets for P, the other slicing techniques are safe. However, additions of predicate or assignment statements to P adversely impact the inclusiveness of slicing techniques. For example, suppose a new assignment statement s is added to P. Because the slices constructed by slicing techniques contain only statements that appeared in P prior to its modification, no slice contains s. Any test that executes the block of code in P' in which s is inserted, however, may be modification-revealing. Because slicing techniques do not select such tests, they are not safe.

Agrawal et al. extend their techniques to address this loss of safety for the relevant and potential relevant slice techniques. When assignment statement s is added to P, the extended techniques include in sl all statements in P that were executed by t, and that use the value computed by s. When predicate statement p is added to P, the extended techniques include in sl all statements in P that are control dependent on p. With these extensions, the relevant and potential relevant slice techniques select some tests that execute new assignment or predicate statements that they would otherwise omit, but they may still omit modification-revealing tests. For example, if statement "`S. print("this code should never execute")`" is added to P, any test that executes S is modification-revealing. However, S does not define any variables, so there are no statements in P that use variables defined in S from which to select tests.

The inclusiveness of slicing techniques is also adversely affected when programs contain multiple modifications. When programs contain multiple modifications, slicing techniques work incrementally, considering changes one by one. Suppose the compound predicate statement "`S1. if c then S2. a := 1`" is added to P, and the statement "`S3. print(a)`" is also added to P at a location reachable from $S2$. This modification

inserts three statements into P'. Suppose that a slicing technique for test selection considers these insertions in order $S1$, $S2$, and $S3$. On considering the new predicate statement $S1$, no statements control dependent on $S1$ are found, so no tests are selected. On considering statement $S2$, because there are no tests known to execute $S2$, and no uses of `a` in P, no tests are selected. Finally, on considering statement $S3$, no tests are known to execute it, so none are selected for it. Given this compound change, tests that reach $S2$ and then $S3$ may be modification-revealing. Because slicing techniques do not select such tests they are not safe.

Precision. In cases where modifications are nonstructural, and do not involve addition of new code, slicing techniques select only modification-traversing tests. By restricting selected tests to those that influence output, the potentially relevant, relevant, and dynamic slices exclude, to different degrees, tests that are modification-traversing but not modification-revealing. When programs contain structural changes, however, the extensions made to increase the inclusiveness of the techniques can cause selection of non-modification-traversing tests. For example, consider the procedures depicted in Figure 4. Suppose statements $S8$, $S6'$, and $S8'$ in the procedures are each replaced by the pair of statements "`x=1`" and "`goto L`". Suppose further that statement $S7'$ is not present in `structchange`', and that a new statement, "`L: print(x)`" is inserted immediately before $S9$ and $S9'$. The extended slicing techniques select all tests that reach the new print statement (the statement with label `L:`), because that statement contains a use of a variable computed in new statement $S6'$. None of these tests are modification-traversing.

Figure 20 depicts the inclusiveness and precision of slicing techniques. The diagrams illustrate the lack of safety of all techniques, leaving portions of the areas corresponding to modification-revealing tests unshaded. Moreover, for all techniques, the diagrams shade some areas corresponding to modification-traversing tests that are non-modification-revealing, signifying the inclusion of such tests.

Efficiency. When program modifications do not add assignment statements or affect flow of control, slicing techniques can perform most of their work in the preliminary phase of regression testing. In this case, the execution slice requires only $O(|T|)$ time per modification to select tests. In the presence of arbitrary modifications, however, slicing techniques are less efficient. To handle additions of assignment statements and predicates, the techniques must compute dynamic data and control dependence information relevant to P', for each test in T. Such computations can require $O(|P'|^2)$ time, and must be performed after modifications are complete; thus, for arbitrary modifications the techniques require $O(|T| * |P'|^2)$ time per modification.

Furthermore, given multiple modifications, to avoid loss of precision and safety, code analyses may need to be repeated or incrementally updated, and test traces collected again, after each modification is considered. Without such recalculation, the techniques cannot account for the cumulative effects of modifications on test paths and slices. At worst, recalculation of traces may require running all tests in T, defeating the purpose of selective retest.

Generality. Slicing techniques work for all procedural-language programs because they depend only on the ability to trace program execution and calculate dependence information. However, the techniques' effectiveness and efficiency decrease in cases where programs contain multiple modifications or modifications to control structure. The techniques do not support interprocedural regression testing beyond the approach of analyzing all procedures in a program. The techniques do not require a particular test suite design or

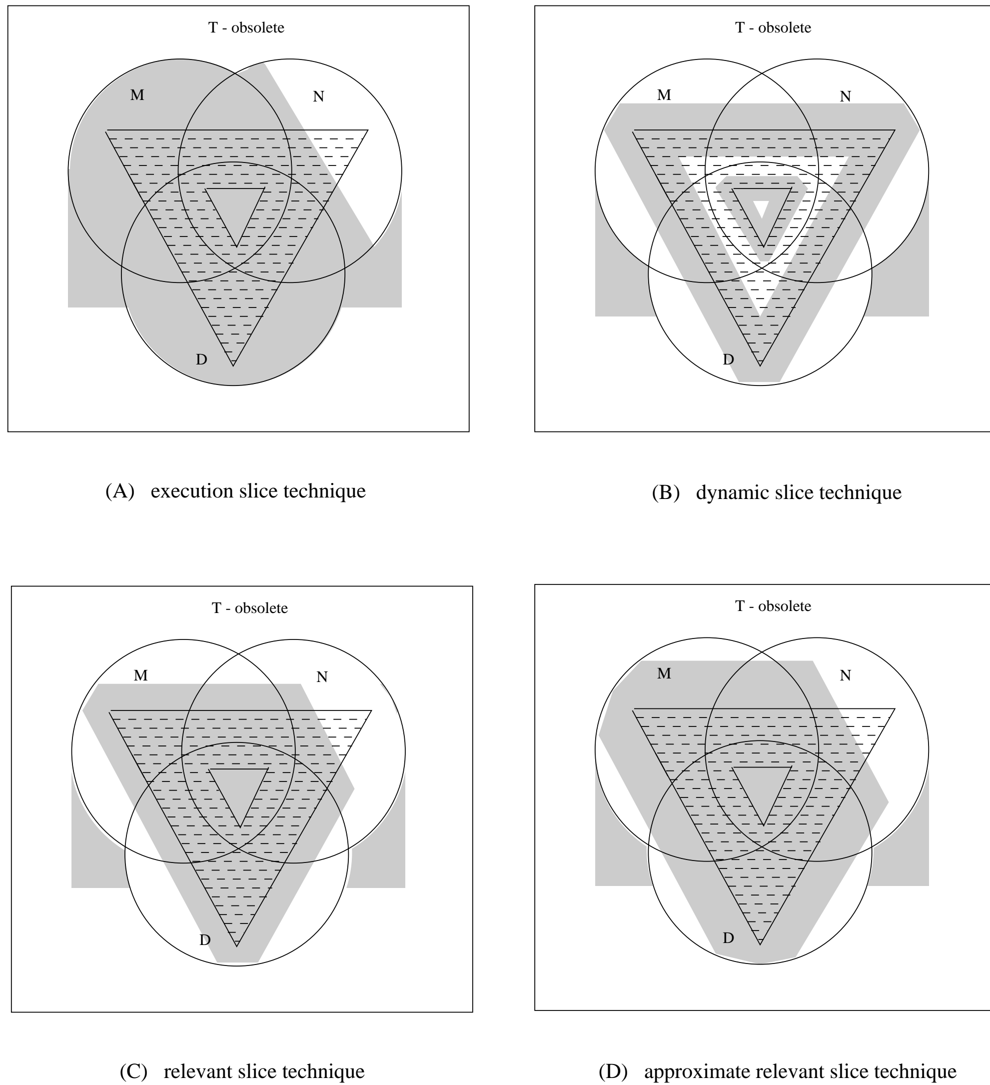

Figure 20: Inclusiveness and precision of slicing techniques.

---

the use of coverage requirements. The techniques require collection of test trace information at either the statement or procedure level; some of the techniques also require tools for static and dynamic dependence analysis.

5.11 Graph Walk Techniques

Rothermel and Harrold[32, 33, 35] present an intraprocedural regression test selection technique that builds control flow graphs (CFGs) for P and P', collects traces for tests in T that associate tests with CFG edges,

and performs synchronous depth-first traversals of the two graphs, comparing nodes (or actually, the program statements associated with those nodes) that are reached along prefixes of execution traces.[15] When a pair of nodes N and N' in the graphs for P and P', respectively, are discovered, such that the statements associated with N and N' are not lexically identical, the technique selects all tests from T that, in P, reached N. This approach identifies tests that reach code that is new in or modified for P', and tests that formerly reached code that has been deleted from P. The technique selects all of the tests in T that are modification-traversing for P and P'[32]. The authors offer an interprocedural version of the technique, also based on CFGs, that can be applied to entire programs or subsystems. Rothermel and Harrold also present versions of their techniques that add data dependence information to CFGs to facilitate more precise test selection.

Inclusiveness. The graph walk techniques select all modification-traversing tests[32]. Thus, for controlled regression testing they are safe.

Precision. Graph walk techniques are not 100% precise for arbitrary programs. Rothermel[32] defines a property of CFGs called the *multiply-visited-node property*.[16] Rothermel proves that when P and P' do not exhibit the multiply-visited-node property, graph walk techniques select *exactly* the tests in T that are modification-traversing for P and P'. When G and G' exhibit the multiply-visited-node property, however, graph walk techniques may select tests that are *not* modification-traversing for P and P'. The procedures shown in Figure 5 exemplify a case where the multiply-visited-node property holds. For the pair of procedures in the figure, graph walk techniques can select non-modification-traversing tests. Nevertheless, in empirical studies conducted using graph walk techniques on nontrivial programs, no cases have been found in which the multiply-visited-node property holds; the only known programs for which the property holds have been contrived for the purpose of demonstrating the property. These empirical results suggest that in practice, graph walk techniques do not select non-modification-traversing tests.

Graph walk techniques that make use of data dependence information further increase the precision of test selection, omitting some modification-traversing tests that are non-modification-revealing.

Like cluster identification techiques, both basic and improved graph walk techniques select tests through modified procedures at a finer grain than linear equation techniques, and thus, are more precise than those techniques.

Figure 21 depicts the inclusiveness and precision of graph walk techniques. Diagrams (A) and (B) show the safety and precision of graph walk techniques for cases where the multiply-visited-node property holds and does not hold, respectively. Diagram (C) shows the safety and precision of the improved versions of the graph walk techniques, that use data dependence information. The safety of the techniques is depicted by the presence of shading in all areas corresponding to modification-revealing tests. The imprecision of the techniques is illustrated by the shading of areas not corresponding to modification-revealing tests. The fact that the techniques can select non-modification-traversing tests when the multiply-visited-node property

[15] Earlier versions of this technique used control, program, or system dependence graphs[33, 35]. The most recent version of the technique is based on control flow graphs[32]. The versions achieve the same results in terms of inclusiveness and precision; however, the CFG-based version is more efficient than the earlier versions. In this discussion, we evaluate the CFG-based version of the technique.

[16] Let G and G' be the control flow graphs for P and P', respectively. The multiply-visited-node property is a predicate that is true of P and P' if, and only if, the graph walk technique, in walking G and G', visits some node in G more than once.

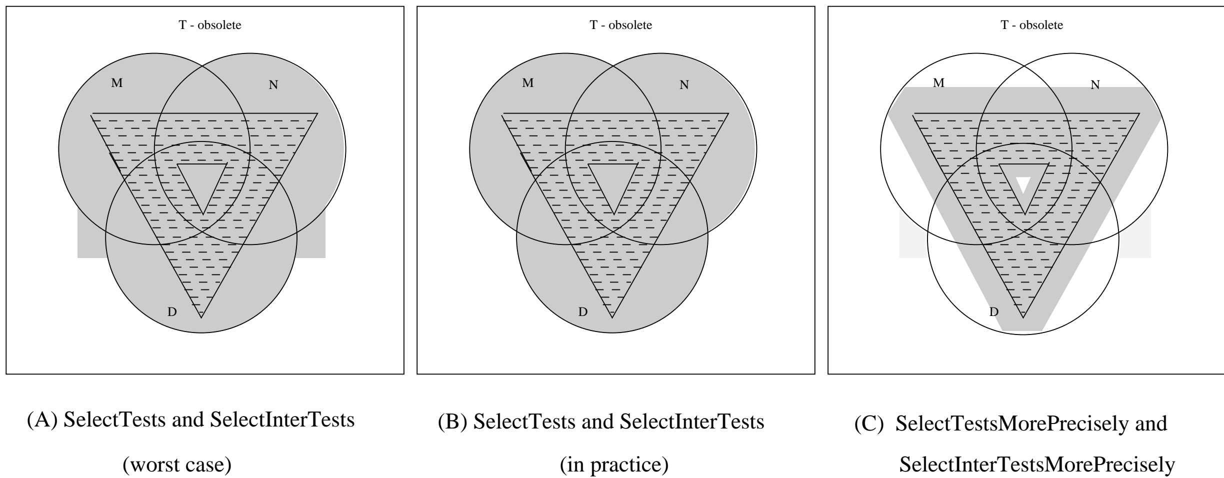

Figure 21: Inclusiveness and precision of graph walk techniques.

holds is illustrated by shading outside the circles in Diagrams (A) and (C); this shading is omitted in Diagram (B), which depicts the conjectured precision of the techniques in practice. However, the shaded area of non-modification-traversing tests area is smaller than the corresponding area in Diagram (B) of Figure 9, reflecting the greater precision of the graph walk techniques. The diagram on the right illustrates the precision gains obtained by the improved versions of the techniques; these techniques omit some tests that are modification-traversing but not modification-revealing.

Efficiency. Graph walk techniques run in time $O(|T| * (max(|P|, |P'|))^2)$. However, when the multiply-visited-node property does not hold, the basic graph walk techniques (that do not require data dependence information) run in time $O(|T| * min(|P|, |P'|))$[32]. The techniques can construct graphs for P and collect test history information during the preliminary phase of regression testing, but must construct graphs for P' and traverse the graphs during the critical phase. The techniques do not require prior computation of a correspondence between P and P'; instead, they locate modifications as they proceed, and in the presence of significant changes avoid unnecessary processing.

Generality. Graph walk techniques apply to procedural languages generally, because control flow graphs and dataflow information can be computed for all such languages. The techniques handle all types of program modifications, and support both intraprocedural and interprocedural test selection. The techniques make no assumptions about initial test suite design or use of coverage criteria. The techniques require test trace information at the basic block level, and tools for constructing control flow graphs; advanced versions of the techniques also require tools for dataflow analysis.

5.12 The Modified Entity Technique

Chen, Rosenblum, and Vo[7] present the modified entity technique, a regression test selection technique that detects modified code entities. Code entities are defined as executable portions of code such as functions, or as nonexecutable components such as storage locations. The technique selects all tests associated with changed entities. The authors have implemented the technique as a software tool, called `TestTube`, that performs regression test selection for C programs. Program entities are kept in a database that, among other things, facilitates comparison of those entities to determine where modifications have occurred. The authors also discuss applications of the technique to the selective retest of nondeterministic systems, where test coverage measures may vary over different test executions, and instrumentation may interfere with test behavior.

Inclusiveness. Although we do not prove this, we believe that by identifying all tests through changed code entities, the modified entity technique identifies all modification-traversing tests. Thus, the modified entity technique is safe for proper regression testing.

Precision. Given a modified function F in program P, the modified entity technique selects all tests that execute F. Because some tests may execute F without executing any modified code in F, the modified entity technique selects non-modification-traversing tests. For example, for the functions and modified versions depicted in Figures 4 and 5, if these functions are called in some program, the technique selects all tests that execute these functions, even though some or all of the tests are non-modification-traversing. Moreover, because the modified entity technique may also select tests that do not execute modified clusters or modified execution traces, the technique is less precise than the cluster identification or CFG-walk techniques. We believe that the set of non-modification-traversing tests selected by the method is equivalent to the set of non-modification-traversing tests selected by the interprocedural linear equation technique.

Figure 22 depicts the inclusiveness and precision of the modified entity technique. Because the technique is safe, all three circles are completely shaded. Because the technique selects non-modification-traversing tests, areas outside the circles are also shaded. That shaded area is comparable in size to the shaded area employed for interprocedural linear equation techniques in Diagram (B) of Figure 9.

Efficiency. The modified entity technique is the most efficient safe test selection technique available. The technique is fully automatable, and runs in worst-case time proportional to the size of the test suite times the number of changed entities in P, which is at worst equivalent to the size of P (though in practice is expected to be much less). Thus the technique runs in time $O(|T| * |P|)$. The technique does require an $O(max(|P|, |P'|) * log(max(|P|, |P'|))$ lexical comparison operation, performed on the code entities in the database during the critical period, to establish which entities have been modified. Experimental performance evaluations show, however, that in practice this comparison operation is one of the least costly operations performed by the technique.[17] The technique handles multiple modifications in a single application of the algorithms.

Generality. Although implemented only for C, the modified entity technique is applicable to procedural languages generally. The technique handles all forms of code modifications. The technique is specifically

[17] David Rosenblum, personal communication.

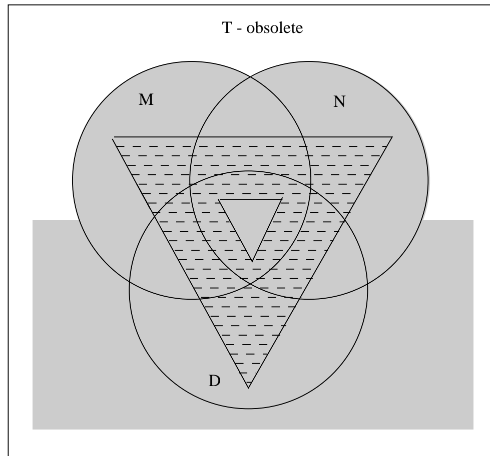

Figure 22: Inclusiveness and precision of the modified entity technique.

designed to handle interprocedural, rather than intraprocedural, regression testing. The technique requires the use of a database containing information about code, but such databases serve other useful purposes. The technique makes no assumption about initial test suite design or use of coverage criteria. Finally, the technique shows promise in application to nondeterministic programs.

5.13 Summary

Table 5.13 summarizes the results of our evaluations of regression test selection techniques. Techniques are listed in the leftmost column; columns two through five summarize our findings with respect to inclusiveness, precision, efficiency, and generality, respectively, for each of the techniques.

We include separate rows for the linear equation technique, to summarize it in its intraprocedural, minimization form and in it interprocedural, nonminimization form. We summarize only the incremental version of the dataflow techniques. We use single rows to summarize the two PDG techniques, the two SDG techniques, the four slicing techniques, and the four graph walk techniques.

The columns summarize the following information for each technique, where applicable:

INCLUSIVENESS. We state whether the technique is safe. If the technique is not safe, we list the category or categories of modifications for which the technique may omit modification-revealing tests.

PRECISION. No techniques are 100% precise. We find it useful to state whether techniques (a) select non-modification-traversing tests, or (b) select all tests through modified procedures instead of selecting tests at a finer granularity. When we know that a technique is more or less precise than another, we include this information.

EFFICIENCY. We list the worst case critical period running time of the technique. Where information on the technique's critical period running time in practice is available we include it. We also list the cost of the correspondence between P and P' that the technique must compute, if such a correspondence is

required. Techniques could instead rely on an incremental editor to provide this information; in that case generality is reduced.

GENERALITY. For each technique, we state whether it is intraprocedural, interprocedural, or both ("level:"), the class of modifications that it handles ("mods:"), the criteria on which it is based, if any ("criteria:"), and its requirements in terms of tool support ("requires:"). It would also be appropriate to list, in this column, the class of languages and programs to which the techniques apply, and whether or not the techniques are fully automatable, However, we find that all techniques are fully automatable except for the modification-based technique (for which the test selection process may be, but is not yet, automated). We also find that all techniques apply to procedural programs generally. Thus, to save space we omit these fields from the table.

To save space, we use the following abbreviations in the table:

inter — interprocedural
intra — intraprocedural
mods — modifications
mt — modification-traversing
non-mt — non-modification-traversing

6 Conclusions and Future Work

We have presented a framework for evaluating regression test selection techniques that classifies techniques in terms of inclusiveness, precision, efficiency, and generality. We have illustrated the application of this framework by using it to evaluate existing regression test selection techniques.

One important contribution of this work is the insight it provides into the state of current research on regression test selection. Where current research is concerned, our evaluations indicate that despite the differences in the goals of various techniques, these techniques may be more clearly compared and understood when our framework is employed. Our framework can be used by researchers to compare their new techniques to existing techniques, and can help them demonstrate the significance of the contributions of their work over existing techniques. Our framework can be used to identify strengths and weaknesses of various techniques, and can help guide the choice of test selection techniques for practical purposes. For example, a testing professional seeking a safe test selection technique can identify, using our evaluations, four possible techniques that will serve the purpose: the linear equation, cluster identification, modified entity, and graph walk techniques. Alternatively, a testing professional who is primarily concerned with using existing tests to achieve some coverage measure may seek a technique that is based on some such measure, rather than a safe technique.

Of equal importance, however, our work suggests several directions for future research on regression test selection. One important direction for future work is experimental study. Few techniques that we evaluated have been implemented, and fewer still have been the subject of empirical studies. With our framework, we have been able to analytically evaluate the fault-detecting abilities and efficiency of existing techniques; however, it is important to pursue empirical evidence as well. We discuss a few important areas for empirical study:

TECHNIQUE	INCLUSIVENESS	PRECISION	EFFICIENCY	GENERALITY
LINEAR EQUATION (minimization) (intra)	unsafe (all categories)	selects non-mt tests	worst case: exponential in $\|P\|$ in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2)$	level: intra mods: not control flow criteria: control/dataflow requires: segment traces, linear equation solver
LINEAR EQUATION (non-minimization) (inter)	safe for controlled regression testing	selects non-mt tests selects all tests through modified procedures	worst case: exponential in $\|P\|$ in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2 * log(max(\|P\|, \|P'\|))$	level: inter mods: handles all criteria: control/dataflow requires: function traces, linear equation solver
SYMBOLIC EXECUTION	not safe (for deletions)	selects non-mt tests	worst case: exponential in $\|P\|$ (may not terminate) in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2 * log(max(\|P\|, \|P'\|))$	level: intra/inter mods: not deletions criteria: partition requires: symbolic execution
PATH ANALYSIS	not safe (for deletions or additions)	selects no non-mt tests	worst case: exponential in $\|P\|$ in practice: unknown correspondence: not required	level: intra mods: not deletions/additions criteria: path requires: statement traces, algebraic design
DATAFLOW (incremental)	not safe (all categories)	selects non-mt tests	worst case: $O(\|T\| * \|P'\|^2)$ per modification in practice: unknown correspondence: not required	level: intra/inter mods: only dataflow affecting criteria: dataflow requires: basic block traces, static and incremental dataflow analysis tools
PROGRAM DEPENDENCE GRAPH	not safe (for deletions)	selects non-mt tests less precise than SDG technique	worst case: $O(\|T\| * (max(\|P\|, \|P'\|)^3)$ or $O(\|T\| * (max(\|P\|, \|P'\|)^4)$ in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2)$	level: intra mods: not deletions criteria: PDG requires: statement traces, control dependence, slicing, dataflow analysis tools
SYSTEM DEPENDENCE GRAPH	not safe (for deletions)	selects non-mt tests more precise than PDG technique	worst case: $O(\|T\| * (max(\|P\|, \|P'\|)^3)$ or $O(\|T\| * (max(\|P\|, \|P'\|)^4)$ in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2)$	level: intra/inter mods: not deletions criteria: PDG requires: statement traces, control dependence, slicing, dataflow analysis tools
MODIFICATION BASED	not safe (all categories) (minimization)	unknown	worst case: $O(\|T\| * \|P'\|^2)$ per logical modification for analysis; test selection is not automated. in practice: unknown correspondence: $O(max(\|P\|, \|P'\|)^2)$	level: intra/inter mods: doesn't handle all criteria: none requires: statement traces, static/dynamic control and data dependence analysis
FIREWALL	safe for controlled regression testing if T is reliable	selects non-mt tests selects all tests through modified procedures	worst case: $O(\|T\| * (max(\|P\|, \|P'\|))$ in practice: "efficient" correspondence: $O(max(\|P\|, \|P'\|)^2 * log(max(\|P\|, \|P'\|))$	level: inter mods: handles all criteria: none requires: function traces
CLUSTER IDENTIFICATION	safe for p.r.t.	selects non-mt tests less precise than graph walk technique	worst case: $O(\|T\| * max(\|P\|, \|P'\|)^3)$ in practice: unknown correspondence: not required	level: intra mods: handles all criteria: none requires: statement traces CFGs, control dependence
SLICING	not safe (some categories)	selects non-mt tests	worst case: $O(\|T\| * \|P'\|^2)$ per modification in practice: unknown correspondence: not required	level: intra mods: doesn't handle all criteria: none requires: statement traces, static/dynamic control and data dependence analysis
GRAPH WALK	safe for controlled regression testing	selects non-mt tests (but not in practice) most precise safe technique	worst case: $O(\|T\| * (max(\|P\|, \|P'\|)^2)$ in practice: $O(\|T\| * min(\|P\|, \|P'\|)$ (without dataflow information) correspondence: not required	level: intra/inter mods: handles all criteria: none requires: statement traces, dataflow analysis (optional)
MODIFIED ENTITY	safe for controlled regression testing	selects non-mt tests selects all tests through modified procedures less precise than graph walk, cluster ident.	worst case: $O(\|T\| * \|P\|)$ in practice: "efficient" correspondence: $O(max(\|P\|, \|P'\|) * log(max(\|P\|, \|P'\|))$	level: inter mods: handles all criteria: none requires: function traces, code database

Table 1: Summary of our framework-based evaluations of regression test selection techniques.

The precision-efficiency tradeoff. Graph walk techniques are more precise than the modified entity technique. However, graph walk techniques gain their precision by increasing the costs of analysis. Similarly, graph walk techniques that use data dependence information are more precise than graph walk techniques that do not use such information, but this precision gain, too, is achieved only at an increase in analysis cost. Empirical studies can help to determine when the increased analysis costs outweigh the gains of increased precision.

Fault-detection abilities. We have used our framework to analytically evaluate test selection techniques in terms of their fault-detecting abilities. We have shown that safe techniques can detect faults that are not detectable by techniques that are not safe; we have also shown that certain techniques are safe, at least, for controlled regression testing. However, we have not determined the impact, in practice, of safety on fault detection. Empirical studies can help to determine whether a safe interprocedural test selection technique, such as the graph walk technique, offers sufficient improvements in fault detection in comparison to an efficient, but nonsafe, interprocedural test selection technique such as the firewall technique.

Interprocedural versus intraprocedural test selection. Many test selection techniques are intraprocedural. Preliminary experimental results suggest that such techniques may not offer savings that justify their costs[32]. More extensive empirical studies can help to determine the level at which test selection should be performed.

Minimization techniques. Minimization techniques take coverage to an extreme, requiring selection of only a single test through some modified or affected component of P'. These techniques significantly reduce the number of tests that must be executed. However, preliminary experimental results suggest that minimization of test suites may have a significant, adverse impact on the ability to detect regression errors[32]. Additional empirical studies in this area would be useful.

Another important direction for future work is the investigation of other selective retest tasks. Our work has focused on the regression test selection problem: the problem of selecting tests from an existing test suite. However, selective retest techniques may be concerned with other tasks. First, simply reusing tests in T may not provide adequate testing of modified programs. Thus, many selective retest techniques also address the coverage identification problem: the problem of locating components of the modified program that should be retested, and judging where additional tests are required. The framework for comparing test selection techniques presented in this paper could be extended to facilitate comparisons of techniques for coverage identification. Second, in this work we have assumed the availability of a technique for identifying obsolete tests; these tests include tests that are obsolete due to changes in specifications. The framework we have presented could be extended to facilitate comparisons of specification-based selective retest techniques.

7 Acknowledgements

S.S. Ravi made many helpful suggestions, especially in regard to the material presented in Sections 3 and 4. This work was partially supported by grants from Microsoft, Incorporated and Data General Corporation, and by National Science Foundation under Grant CCR-9357811 to Clemson University and The Ohio State University.

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 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, 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 352–361, October 1988.

[4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50, November 1992.

[5] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *Proceedings of the Conference on Software Maintenance - 1995*, pages 251–260, October 1995.

[6] P.A. Brown and D. Hoffman. The application of module regression testing at TRIUMF. *Nuclear Instruments and Methods in Physics Research*, Section A, .A293(1-2):377–381, August 1990.

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

[8] H. Crowder, E.L. Johnson, and M. Padberg. Solving large-scale zero-one linear programming problems. *Operations Research*, 31(5):803–834, September 1983.

[9] T. Dogsa and I. Rozman. CAMOTE - computer aided module testing and design environment. In *Proceedings of the Conference on Software Maintenance - 1988*, pages 404–408, October 1988.

[10] K.F. Fischer. A test case selection method for the validation of software maintenance modifications. In *Proceedings of COMPSAC '77*, pages 421–426, November 1977.

[11] 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.

[12] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman, New York, 1979.

[13] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 299–308, November 1992.

[14] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[15] 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.

[16] M.J. Harrold and M.L. Soffa. An incremental data flow testing tool. In *Proceedings of the Sixth International Conference on Testing Computer Software*, May 1989.

[17] M.J. Harrold and M.L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[18] J. Hartmann and D.J. Robson. Revalidation during the software maintenance phase. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 70–79, October 1989.

[19] J. Hartmann and D.J. Robson. RETEST - development of a selective revalidation prototype environment for use in software maintenance. In *Proceedings of the Twenty-Third Hawaii International Conference on System Sciences*, pages 92–101, January 1990.

[20] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31–38, January 1990.

[21] D. Hoffman. A CASE study in module testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 100–105, October 1989.

[22] D. Hoffman and C. Brealey. Module test case generation. In *Proceedings of the Third Workshop on Software Testing, Analysis, and Verification*, pages 97–102, December 1989.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[24] 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 282–290, November 1992.

[25] J.A.N. Lee and X. He. A Methodology for Test Selection. *The Journal of Systems and Software*, 13(1):177–185, September 1990.

[26] H.K.N. Leung and L. White. Insights Into Regression Testing. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 60–69, October 1989.

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

[28] 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.

[29] 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 201–208, October 1991.

[30] R. Lewis, D.W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In *ESEC '89. 2nd European Software Engineering Conference Proceedings*, pages 487–496, September 1989.

[31] T.J. Ostrand and E.J. Weyuker. Using dataflow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, pages 233–247, September 1988.

[32] G. Rothermel. *Efficient, Effective Regression Testing Using Safe Test Selection Techniques*. Ph.D. dissertation, Clemson University, May 1996.

[33] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 358–367, September 1993.

[34] G. Rothermel and M.J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 14–25, September 1994.

[35] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 94)*, pages 169–184, August 1994.

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

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

[38] B. Sherlund and B. Korel. Logical modification oriented software testing. In *Proceedings: Twelfth International Conference on Testing Computer Software*, June 1995.

[39] 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 527–534, September 1989.

[40] A. von Mayrhauser, R.T. Mraz, and J. Walls. Domain based regression testing. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 26–35, September 1994.

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

[42] L.J. White, V. Narayanswamy, T. Friedman, M. Kirschenbaum, P. Piwowarski, and M. Oha. Test Manager: a regression testing tool. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 338–347, September 1993.

[43] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *17th International Conference on Software Engineering*, pages 41–50, April 1995.

[44] W. Yang. Identifying syntactic differences between two programs. *Software—Practice and Experience*, 21(7):739–755, July 1991.

[45] S.S. Yau and Z. Kishimoto. A method for revalidating modified programs in the maintenance phase. In *COMPSAC '87: The Eleventh Annual International Computer Software and Applications Conference*, pages 272–277, October 1987.

[46] J. Ziegler, J.M. Grasso, and L.G. Burgermeister. An Ada based real-time closed-loop integration and regression test tool. In *Proceedings of the Conference on Software Maintenance - 1989*, pages 81–90, October 1989.