The Application of Program Slicing to Regression Testing

David Binkley*

Loyola College in Maryland

ABSTRACT

Software maintainers are faced with the task of regression testing: retesting a program after a modification. The goal of regression testing is to ensure that bug fixes and new functionality do not adversely affect the correct functionality inherited from the original program. Regression testing often involves running a large program on a large number of test cases; thus, it can be expensive in terms of both human and machine time. Many approaches for reducing the cost of regression testing have been proposed. Those that make use of program slicing are surveyed.

Keywords: program slicing, incremental regression testing, test case selection, test-data adequacy criteria, dependence graphs

1 INTRODUCTION

This paper reviews research on the application of program slicing to the problem of regression testing. Regression testing is part of the larger problem of program testing (the application of program slicing to this more general problem is not considered. White [23] provides a survey of software testing methods). Testing is an important part of software engineering as it consumes at least half of the labor expended to produce a working program [5].

Simply put, the primary goal of program testing is to "catch bugs." Testing techniques can be divided into functional testing techniques (*e.g.*, transaction flow testing, syntax testing, domain testing, logic-based

*Computer Science Department, Loyola College in Maryland, 4501 North Charles Street, Baltimore, Maryland 21210, 410-617-2881. binkley@cs.loyola.edu. Supported in part by National Science Foundation grant CCR-9411861.

testing, and state testing) and structural testing techniques (*e.g.*, control flow graph path testing, and data-flow testing) [5]. Functional (black-box) testing techniques do not inspect the code; they generate tests based on requirements and design information. In contrast, structural (white-box) testing techniques are based on the internals of the code.

Program slicing, which is based on the internals of the code, can be most obviously applied to structural testing techniques. For example, one common error made by maintenance engineers is to modify a definition because some use is incorrect, but not check the other uses of that definition [17]. A forward slice from the modified definition identifies all the uses that need to be tested. Perhaps with less obvious impact, program slicing can also be used in functional testing. For example, given a functional test, an executable slice that captures the functionality to be tested should be easier and less expensive to test than the complete program.

The remainder of this section discusses incremental regression testing and its cost reduction, and describes the use of test-data adequacy criteria in testing software. The next section surveys a collection of published techniques that use slicing to reduce the cost of regression testing. Finally, Section 3 contains a brief summary. This paper assumes that the reader is familiar with program slicing (including static and dynamic, and forward and backward slicing) and the computation of slices using data-flow equations and dependence graphs as in the two-pass HRB algorithm presented by Horwitz et al. [13].

1.1 Incremental Regression Testing

"Today software development and testing are dominated not by the design of new software, but by the rework and maintenance of existing software" [5]. Such software maintenance activities may account for as much as two-thirds of the overall cost of software production [19]. In one study, the probability of introducing an error while making a change was between fifty and eighty percent [15]. Increased testing after a change should reduce this percentage. In particular, automation of the testing process should be beneficial because the typical testing process is a human-intensive activity and as such, it is generally costly, time-consuming, error prone, and often inadequately done [17].

Testing during the maintenance phase of a program's life is dominated by *regression testing*, which can be defined as "any repetition of tests (usually after a software or data change) intended to show that the software's behavior is unchanged except insofar as required by the change to the software or data" [5]. Regression testing attempts to validate modified parts of the software, while ensuring that no new errors are

introduced in previously tested code. It is performed to provide confidence that modified code behaves as intended, and that modifications have not inadvertently disturbed the behavior of unmodified code.

The following terminology is used throughout this paper: The previously tested version of a program is referred to as certified. The program obtained by modifying certified is referred to as modified. A *test suite* is a set of test cases used to test a program. Test suite T is the test suite used to test certified and test suite T' is the test suite used to test modified.

Regression testing involves creating test set T', to validate modified, and then running modified on T'. In practice, test suite T' typically contains all of T plus new tests designed to test modified's new functionality. Running modified on all these tests is usually too expensive [15].

Regression testing differs from initial testing in that test cases and the results from their execution on certified are available to the tester. *Incremental regression testing* attempts to exploit this information in two ways. First, if a new component is tested by an existing test case, then the costly construction of a new test case can be avoided. For this reason many techniques attempt to maximize the reuse of tests from T. Second, if it may be asserted that, for a subset of T, modified will produce the same output as certified, then the potentially costly execution of modified on this subset can be avoided. This occurs, for example, if no changed program components are executed by a test.

Incremental regression testing also attempts to exploit the following coverage observations [3]: In a 14,500 line C program with 6500 basic blocks, 183 functions in 27 files and 413 tests, 71% of the basic blocks were executed by at least one test in T, but the average test executed only 11% of the basic blocks. Furthermore, the tests of T executed 88% of the functions, but the average test executed only 26% of the functions. Thus, for a small change, many tests should not need to be rerun.

Two relationships between program semantics and syntax are useful in incremental regression testing. The first begins with the observation that two statements with the same text have the same semantics (given the same input value, they produce the same result). Thus, if two programs have a single syntactic difference, before this difference is executed they must have the same behavior. Second, two intraprocedural slices that are syntactically identical have the same semantics; thus, two programs that share a slice share a computation [16]. This relationship extends to interprocedural slices [6].

Incremental regression testing has the following steps:

(1) Identify a set of affected program components of modified.

(2) Identify a subset of T that tests the affected components. (Many of the slicing-based techniques focus on this step.)

(3) Identify untested components of modified that must be covered by new tests.

(4) Create new tests for the uncovered components.

(5) Run modified on T', the union of the tests produced in Steps (2) and (4).

Some comments on this process are in order.

(1) Step (3) is often performed by running modified on the tests from Step (2) and marking the components that are executed. The unmarked components are those identified by Step (3).

(2) Some techniques attempt to minimize the size of T'.

(3) To revalidate the functionality inherited from certified (*i.e.*, to ensure that no errors are introduced into modified, but not test new functionality), T' can be a subset of T.

(4) When using test data adequacy criteria (defined below), if T is adequate for certified then T' should be similarly adequate for modified.

(5) To perform Step (1), many techniques require a mapping between the components of certified and modified. Some compute this as they go; others, require it as input.

For a test-case selection algorithm to be worthwhile, it must cover its costs. That is, the cost of selecting tests and then running them must be less than the cost of simply rerunning all tests. Some of the algorithms discussed in Section 2 divide the cost of selecting tests into two kinds of processing: *Critical* processing and *off-line* (or *preliminary*) processing. When critical processing begins, regression testing is the dominant activity and can become the (product release) limiting activity. Critical processing cannot begin until after the modification is complete and the tester desires to perform regression testing. In contrast, off-line processing can be performed while a modification is being planned, implemented, etc. Since off-line processing is "free" in many respects, one way to reduce the cost of regression testing is to move as much processing as possible to off-line processing.

Finally, there is an important assumption that is made (implicitly or explicitly) by most research involving program slicing and regression testing. The *controlled regression testing assumption* deals with the environment in which programs are tested:

> When modified is tested with T', we hold all factors that might influence the output of modified, except for the code in modified, constant with respect to their states when we tested certified with T [18].

This assumption may be violated, in practice, when a program is ported to a machine with a different processor or a different amount of memory. Furthermore, it may be violated on a single machine if the location at which a program is loaded affects the program's behavior.

Some of the algorithms given in Section 2 relate the formal semantics of certified and modified based on their slices. These relationships extend from formal semantics to runtime behavior under a deterministic computational model where each time a program is executed on the same input, it produces the same output. This model also requires that the controlled regression testing assumption is not violated.

1.2 Test-Data Adequacy Criteria

Many of the techniques discussed in Section 2 use test data adequacy criteria. A test data adequacy criterion is a minimum standard that a test suite for a program must satisfy. An adequacy criterion is specified by defining a set of "program components" and what it means for a component to be exercised. An example is the *all-statements* criterion, which requires that all statements in a program must be executed by at least one test case in the test suite. Here statements are the program components and a statement is exercised by a test if it is executed when the program is run on that test. Satisfying an adequacy criterion provides some confidence that the test suite does a reasonable job of testing the program.

Test-data adequacy criteria can be divided into three groups: control flow based criteria (for example, all-statements), data-flow based criteria (for example, all-c-uses) [22], and program dependence graph (PDG) based criteria (for example, all-flow-edges) [4]. These three examples are defined as follows:

Definition. The *all-statements* criterion is satisfied by a test suite T if for each statement s there is some test case t in T that exercises s. A statement is *exercised* by test case t if it is executed when the program is run with input t.

Definition. The *all-c-uses* criterion is satisfied by a test suite T if for each computational use (a use not in a predicate statement) of variable x and path in the control flow graph from a definition of x that reaches the use, there is some test case t in T that exercises the definition, the path, and the use. A path is *exercised* by test case t if all the statements along the path are exercised when the program is run with input t.

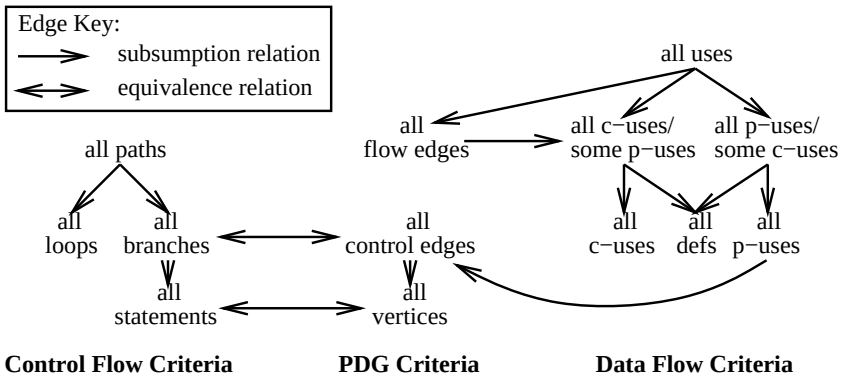

Figure 1: Criteria Relationships [4]

DEFINITION. The *all-flow-edges* criterion is satisfied by a test suite T if for each flow edge e there is some test case t in T that exercises e. Flow edge e from a definition of x at vertex u to a use at vertex v is *exercised* if u is exercised, no definition of x that lies on the control flow graph path taken from u to v is exercised, and finally v is exercised. A vertex is *exercised* if its corresponding statement is exercised.

The program dependence graph, which includes both control and data dependences, can be used to provide a single, unifying framework in which to define adequacy criteria based on control and data flow [4]. Note that a flow dependence edge captures the same information as a definition-use pair (du-pair). Figure 1 shows the relationship for several criteria of the three types.

2 THE TECHNIQUES

This section surveys seven papers found in the literature that apply program slicing to the problem of reducing the cost of regression testing. They are divided into three groups. The first group uses dynamic slicing [14, 2, 1]. The second group represents programs using program dependence graphs [9, 12], and the third group is based on Weiser's data-flow definition of slicing [21]. Some of the techniques make only limited use of program slicing and are included here for completeness.

The following common example is used in each section to illustrate the techniques. It reads the three side lengths of a triangle (assumed to be in descending order), classifies the triangle as one of scalene, isosceles, right, or equilateral, computes its area using a formula based on the class, and outputs the area and the class [3].

```
[ 1]    read(a, b, c)
[ 2]    class := scalene
[ 3]    if a = b or b = c
[ 4]        class := isosceles
[ 5]    if a*a = b*b + c*c
[ 6]        class := right
[ 7]    if a = b and b = c
[ 8]        class := equilateral
[ 9]    case class of
[10]        right:       area := b*c/2
[11]        equilateral: area := a*a*sqrt(3)/4
[12]        otherwise:   s := (a+b+c)/2
[13]                     area := sqrt(s*(s-a)*(s-b)*(s-c))
[14]    end case
[15]    write(area, class)
```

The following test suite was used to test the program.

test	input			output	
t_1	7	7	7	21.22	equilateral
t_2	4	4	3	5.56	isosceles
t_3	5	4	3	6.00	right
t_4	6	5	4	9.92	scalene
t_5	4	3	3	4.47	scalene

2.1 Dynamic Slicing Techniques

Agrawal et al. [3] present three algorithms. The first two are straw men designed to motivate the third. Each algorithm attempts to select a minimal subset of T that will test modified's inherited functionality. They do not consider finding new test cases to cover new functionality. The three algorithms are based on one or more of the following observations:

(1) If a statement is not executed by a test case it cannot affect the program's output for that test case.

(2) Not all statements in the program are executed under all test cases.

(3) Even if a statement is executed under a test case it doesn't necessarily affect the program's output.

(4) Every statement does not necessarily affect every part of the program's output.

The 3 strategies all attempt to relegate the bulk of the processing to off-line processing, which mostly involves computing various slices. The first strategy overestimates the number of tests that must be rerun; the second underestimates it and is thus unsafe. The final strategy provides a better safe approximation.

The first strategy is based on the observation that "if, for a given test case, control never reaches any changed statement during certified's execution then it will never reach one during modified's execution." Thus, modified need only be rerun on those tests cases that execute a changed statement. Statements are treated as changed if they are reached by a new definition (*e.g.*, one caused by the addition of a new assignment statement or a change on the left-hand side of an existing assignment) or their control nesting changes (either by the addition or deletion of a control predicate). This strategy is conservative because every statement that is executed may not affect that output of the program (consider two sequential assignments to the same variable).

The first strategy is conservative but safe (selects all tests that may produce different output). The second strategy errs in the other direction and is unsafe. During off-line processing, the dynamic slice with respect to certified's output for each test case in the test suite is computed. Modified is run on those test cases whose dynamic slice includes a modified statement. This approach is unsafe because a modification may change the statements contained in a dynamic slice.

The final strategy replaces the dynamic slice in the second strategy with a *relevant slice*. Only test cases whose relevant slices contain a modified statement are selected. Informally, a relevant slice extends a dynamic slice with certain elements of a static slice: "It includes the necessary control statements without including the statements that control them" [3]. It includes less of the program than the set of executed statements but more of the program than a dynamic slice. The definition of a relevant slice makes use of the notion of *potentially dependent* [3]:

Definition. The use of a variable v at location l in a given execution history is *potentially dependent* on an earlier occurrence p of a predicate in the execution history if

(1) v is never defined between p and l but there exists another path from p to l along which v is defined, and

(2) changing the evaluation of p may cause this untraversed path to be traversed.

Note that the change in (2) allows only one change; thus, if changing the evaluation of p and some other predicate is necessary, v is not potentially dependent on p.

The *relevant slice* extends the dynamic slice to include predicates on which statements in the slice are potentially dependent and the data and potential dependences of these predicates. It does not include control dependences because this would include unwanted predicates.

EXAMPLE. This example considers two modifications to the example program. For the first modification, the dynamic slicing technique will fail to select test t_2. For the second, the execution trace technique will (unwantedly) select t_2. The relevant slice technique will correctly select t_2 in only the first case.

The first modification erroneously replaces the "`and`" in Line 7 with an "`or`." For test t_2, Lines 1, 2, 3, 4, 5, 7, 9, 12, 13, and 15 are executed. The dynamic slice includes Lines 1, 3, 4, 9, 12, 13, and 15. The execution trace technique selects t_2 because the modified line, Line 7, was executed. However, the dynamic slice technique does not select t_2 because Line 7 is not in the dynamic slice. Test t_2 produces different output after this change, so it should be selected. In this case the relevant slice includes exactly the executed statements, so the relevant slicing technique also selects t_2.

To see the need for relevant slicing, assume Lines 7 and 8 in certified had been

```
[7a]    if a = b
[7b]        if b = c
[8 ]            class := equilateral
```

For test case t_2 the predicate on Line `7a` is true, but the predicate on Line `7b` is false. Because changing `7b` to true causes Line 8 to execute, but changing `7a` to false cannot cause Line 8 to execute, the use of `class` on Line 9 is potentially dependent on `7b`, but not `7a`. Therefore, the relevant slice includes Line `7b`, but not Line `7a`. If in modified Line `7a` is changed to "`if a != b`," the execution trace technique will select t_2 because Line `7a` was executed. However, the relevant slice technique will not, because Line `7a` is not in the relevant slice. ◇

One drawback of the relevant slice strategy is that it requires static data dependence information. This static information can be expensive to compute and is necessarily inaccurate due to the use of pointers, arrays and dynamic memory allocation. Agrawal et al. suggest an approximation that is simpler, but more conservative. This approximation adds all predicates that were executed to the slice being constructed along with their transitive data dependences.

The initial presentation of these three strategies assumes that a single change is made and that this change does not change the control flow graph or any left-hand-side of an assignment statement. These assumptions are relaxed later in the paper; however, several pragmatic issues are ignored. For example, changes in the control flow graph bring up a matching problem where a correspondence between the statements of certified and modified must be found. Relaxing the single change assumption should not adversely affect the strategies; however, this may require further study.

Finally, to reduce overhead (*i.e.*, the cost of instrumenting the original program for dynamic slicing and performing test-case selection) at the expense of running more tests, Agrawal et al. suggest that their techniques be applied at the basic block, function, or even module level rather than the statement level.

2.2 Dependence Graph Techniques

This section includes four techniques that make use of program dependence graphs (PDGs). It assumes some familiarity with the use of PDGs to compute program slices [12, 13]. The first two techniques make heavy use of slicing and introduce several new kinds of slices. The second two use the PDG as an underlying representation, but make limited use of program slicing. They are included here for completeness.

2.2.1 Heavy-Use Techniques

Bates and Horwitz [4] introduced the dependence graph test-data adequacy criterion. They proposed several techniques to identify a safe approximation to (1) the set of components affected by a modification, and (2) tests from T that are guaranteed to exercise these affected components. They were the first to make such guarantees.

Bates and Horwitz introduce three PDG based test-data adequacy criteria: all-vertices, all-control-edges, and all-flow-edges. For a given criterion, their technique identifies affected components, partitions the affected components and those of certified based on their execution behavior, and then selects tests from these partitions. This requires defining the behavior of a component. For the all-vertices criterion, the execution behavior of a component is the sequence of values computed by the component. For example, the predicate of an if statement computes a sequence of boolean values.

This section considers the all-vertices criterion as an example. Thus the components are the vertices of the PDG. These vertices correspond to the statements and control predicates of the program; thus, the all-vertices criterion is equivalent to the all-statements criterion.

Affected vertices are identified by first identifying the set of directly affected points (DAPs). Vertex v is a DAP if it is new in the PDG for modified or has different incoming edges in PDGs for certified and modified [12, 8]. This requires, a priori, a correspondence between the vertices of modified and certified. For the all-vertices criterion, the set affected contains those vertices in the forward slice with respect to the DAPs. This set is a super set of the truly affected vertices. To be safe, affected must include all the truly

affected components. In general, it also includes components that in fact have the same behavior, since precisely computing the affected components is undecidable.

To identify the tests from T that will exercise the components of affected, the vertices of affected and certified are partitioned into equivalence classes based on equivalent execution patterns [4]:

Definition. Vertices v_1 and v_2 have *equivalent execution patterns* iff v_1 and v_2 are exercised the same number of times on any given input. A vertex is exercised when the corresponding statement is executed.

This definition assumes that both programs terminate normally. The extension to handle the three cases involving one or both programs not terminating normally is straightforward [4].

Precise determination of equivalent execution patterns is undecidable. Using a new kind of slice called a *control slice*, Bates and Horwitz provide a safe approximation [4]:

Definition. The *control slice* with respect to vertex v in PDG G is the traditional backward slice of PDG G' taken with respect to v' where

(1) G' contains all the vertices of G plus a new unlabeled vertex v', and

(2) G' contains all the edges of G, plus new control edges from the control-predecessors of v to v'.

Thus, a control slice captures the conditions under which v executes, but not the values used at v. (An alternative definition of a control slice is the backwards slice of G with respect to the set of control predecessors of v.)

The next step partitions the vertices of affected and certified based on control slice isomorphism. Bates and Horwitz prove that if the vertices of certified and modified are partitioned into equivalence classes based on control slice isomorphism, then the vertices in each class have equivalent execution patterns. Thus, all vertices in the same partition are exercised by the same tests. Bates and Horwitz also define control slices for the all-control-edges and all-flow-edges criterion and prove a similar results for each.

The resulting set of tests to be rerun, T', is collected from the partitions that contain affected vertices: For each vertex in affected in an equivalence class with a vertex c of certified, add to T' all tests from T that exercised c.

There may be affected vertices that exist in an equivalence class without any vertices from certified. The technique can't safely determine that any test exercises such vertices. For example, consider a predicate change from "$x > 2$" to "$x > 3$." The algorithm cannot safely determine that a test case that used to

execute the true branch still does so (note that a test with $x = 10$ still does). Like other approaches, at this point Bates and Horwitz run the selected tests and mark the actual vertices exercised. New test cases are then required for all unmarked vertices.

EXAMPLE. If Line 8 of the example program is replaced by Line 8′: "`class := equiangular`" in modified, then the vertices representing Lines 8-15 are all affected. All affected vertices have in their equivalence class at least the corresponding vertex from certified. For example, one equivalence class includes the vertices for Lines 12 and 13 from certified and modified. Furthermore, the vertices for Lines 8 and 8′ are in the same equivalence class; therefore, tests that test Line 8 also test Line 8′. For this change no new tests need be created.

Now consider a separate change that replaces "`and`" with "`or`" in Line 7 giving Line 7′. In this case affected includes Line 8 (and 9-15) from modified. The equivalence class containing Line 8 includes no vertices from certified. A test causing Line 7′ to be true would need to be found. ◇

The work of Bates and Horwitz considers single procedure programs. Binkley [7] presents two complementary algorithms for reducing the cost of regression testing that operate on programs with procedures and procedure calls. The first algorithm can reduce the complexity of the program on which test cases must be rerun: given certified and modified, it produces a smaller program differences that captures the *behavior* of modified that is different from the behavior of certified. The second algorithm is an interprocedural extension of the work of Bates and Horwitz. Both algorithms operate on the system dependence graph (SDG) [13]. Used together, the testing of the "large" program modified with a "large" number of test cases can be replaced by running the smaller program differences on a smaller number of test cases.

The first algorithm produces a program that captures the semantic differences between modified and certified. It is computed by identifying the potentially affected components of modified and then producing a subgraph of modified's SDG, referred to as Δ(modified, certified), which captures the computation of the affected components. This subgraph is then used to produce the program differences.

The meaning of a program component was defined by Bates and Horwitz as the sequence of values produced by each component. This sequence alone is sufficient in the absence of procedures, but in the presence of procedures and procedure calls it provides too coarse a definition. Consider, for example, a program with two independent calls to procedure *increment*. Intuitively, these programs are semantically equivalent since the two calls to *increment* are independent. However, the sequences of values produced by the program components in *increment* depend on the order of the calls to *increment* and are consequently

different. In the presence of procedures, the *meaning* of a component is extended to a set of sequences of values. Each element of this set is produced by a distinct call to the procedure.

The set affected contains a safe approximation to the set of vertices that may have different meaning. Like Bates and Horwitz, it is computed using a forward slice with respect to the directly affected points. To correctly capture the computations represented by these vertices, it is necessary to partition affected into *strongly affected points* and *weakly affected points*. Whereas an affected point potentially exhibits changed behavior during *some* calls to the procedure, a *strongly affected point* potentially exhibits changed behavior during *all* calls to the procedure [7]. Strongly affected points in a procedure P are caused by changes in P and the procedures called by P, but not procedures that call P. This set is safely approximated by taking the first pass of an HRB-forward slice [13] with respect to the directly affected points.

A *weakly affected point* is an affected point that is not strongly affected. Weakly affected points in procedure P are caused by changes in procedures that call P, but not by changes in P or in procedures P calls. This set is safely approximated by removing from the affected points all the strongly affected points.

The operator Δ applied to modified and certified produces a subgraph of modified's SDG that contains all the components necessary to capture the behavior of the affected points. This operator is defined in two parts: an interprocedural backward slice with respect to the strongly affected points and the second pass of the HRB-backward slice with respect to the weakly affected points.

The next step in producing differences makes Δ(modified, certified) *feasible*. (An infeasible SDG is not the SDG of any program.) Once feasible, Δ(modified, certified) is reconstituted into a program differences that has Δ(modified, certified) as its SDG. In general, differences tends to include most of small, single-thought, programs, but smaller percentages of larger programs.

EXAMPLE. Consider changing Line 11, the computation of the area of an equilateral triangle, in the example program. Differences omits the other area computations. Thus, even when it is executed on tests for other kinds of triangles (perhaps to ensure that they are not mistakenly categorized as equilateral), time is not wasted on unwanted computation of `area`. ◇

Binkley's second algorithm performs test-case selection for the all-vertices and all-flow-edges criteria. This section considers the all-vertices criterion as an example. The test-case selection algorithm has two outputs: the subset of T to be rerun and a subset of modified's vertices that the algorithm cannot guarantee are executed by a test from T. This algorithm can be combined with the program differences to reduce both the number of test cases that must be rerun and the size of the program on which they must be run.

The algorithm partitions vertices based on *common execution patterns*. Common execution patterns extend equivalent execution patterns to programs that contain procedures and procedure calls. Equivalent execution patterns, although safe, prove too coarse in the presence of procedures and procedure calls.

DEFINITION. Components c_1 of procedure P_1 and c_2 of procedure P_2 have *common execution patterns* if there exist calling contexts CC_1 for P_1 and CC_2 for P_2 in which c_1 and c_2 have equivalent execution patterns. (A calling-context [7] represents a sequence of active procedure calls.)

Identifying vertices with common execution patterns is undecidable; therefore, in practice a safe approximation is necessary. One such approximation uses calling context slices (a calling-context slice contains less of the program than an interprocedural slice, but more of the program than an intraprocedural slice) [7]:

DEFINITION. A *calling-context slice*, taken with respect to vertex v and calling context CC, includes those statements necessary to capture the computation of v in calling context CC, but no other calling contexts.

Common execution patterns depend on the number of times a vertex is exercised in a given calling context. The number of times vertex v is exercised is determined by the behavior of the vertices on which v is control dependent. Calling-context slices with respect to these vertices are used to identify vertices with common execution patterns [7].

Following Bates and Horwitz, the affected vertices of modified's SDG and the vertices of certified's SDG are partitioned into equivalence classes based on calling-context slice isomorphism. Test-case selection includes in T' those test cases that exercise a vertex from certified that is in the same partition as an affected vertex.

All tests in T' exercise an affected vertex; however, not all test cases in T' need be rerun. Two reductions are possible. The first includes only those tests that execute a DAP. The second further reduces the number of tests selected by maintaining, with each affected vertex v, the set of the DAPs that cause v to be affected. This set is used to weed out test cases that exercise v and only DAPs that do *not* affect v. Such test cases produce the same results in modified and certified.

In the intraprocedural case considered by Bates and Horwitz these reductions are equivalent. In the interprocedural case they are not. In fact the second is actually quite difficult. Both reductions require additional bookkeeping information and thus may not be cost effective.

EXAMPLE. For the example program the test-case selection algorithm performs the same steps as the Bates and Horwitz algorithm because the example program contains no procedures. Consider the effect of moving the computation of `class` into the procedure `compute_class`. If modified contains a new (inde-

pendent) call to `compute_class`, then the vertices associated with procedure `compute_class` have common execution patterns with the corresponding vertices from certified. Tests from T that tested them in certified will continue to test them in modified (through the original call-site). ◇

2.2.2 Limited-Use Techniques

The final two dependence graphs techniques make limited use of program slicing. First, Rothermel and Harrold [20] present algorithms that select tests for affected du-pairs, and that determine those du-pairs that may produce different behavior. Both intraprocedural and interprocedural versions are presented. The algorithms operate over the SDGs for certified and modified.

The algorithms are *safe* in that they select every test that may produce different output in modified. This is in contrast to the test-case selection algorithms presented by Bates and Horwitz and by Binkley, which are not safe in the Rothermel and Harrold sense because of their treatment of deleted components [18]. If a deleted component affects other components in certified that exist in modified, then these other components are affected and their tests are selected. If the deleted component does not affect any other components in certified then its tests are not selected, but are included by Rothermel and Harrold's definition of safe.

The first algorithm, called SelectTests, outputs a subset of T that may produce different results when run on certified and modified. It operates by performing a side-by-side walk of the program's control dependence sub-graphs looking for nodes that have different texts. If the control successors of a predicate or region node n (region nodes summarize the control dependence conditions necessary to reach statements in the region) in certified and modified do not correspond, or correspond but contain predicate, output, or control transfer statements reached by a changed definition, then all tests that exercise n are included in T'. Otherwise, for each affected node control dependent on n that defines a variable v and each use of v reached by this definition, tests that exercise both the definition and the use are selected.

The second algorithm, called CoverageReqs, outputs the set of all affected du-pairs in modified. Note that a du-pair captures the same information as a flow dependence edge. CoverageReqs makes use of forward slicing. It walks the dependence graph for modified, collecting flow dependence edges whose end points differ from those in certified. If it encounters a node n in modified that has no corresponding node in certified, then the forward slice with respect to n is taken. Each flow dependence edge completely (*i.e.*, both ends) contained in this slice is included in T'.

Although only CoverageReqs makes direct use of program slicing, both algorithms contain slice-like control constructs such as "`for each control dependence successor ...`." In other words, the algorithms were written from the perspective of labeling nodes with certain kinds of labels as the algorithm proceeds from node to node. Turning this around, it might be possible to use slicing to gather all nodes that would be given a particular label.

EXAMPLE. For a change on Line 7 of the example program, SelectTests selects all tests because Line 7 is at nesting level 0 (*i.e.*, is executed every time the program is run). Selecting more tests than the dynamic slicing approach of Section 2.1 is indicative of the use of static information and not the Rothermel and Harrold approach. Other static approaches have similar behavior.

For a change on Line 7, CoverageReqs includes du-pairs traversed during the forward slice with respect to the new Line 7. This slice includes all statements after Line 7; thus all du-pairs involving `area` and `s`, and those involving the definition of `class` on Line 8 are selected as needing to be covered.

If Lines 7 and 8 are replaced by

```
[7a]    if a = b
[7b]        if b = c
[8 ]            class := equilateral
```

then a change to Line 8 would cause SelectTests to include tests where `a` and `b` are equal (tests t_1 and t_2) because only they execute Line `7b` (SelectTests appears to be better in the presence of deeper control nesting). For this change, CoverageReqs selects all du-pairs in the forward slice with respect to Line 8, which has the same effect as a change to Line 7 (because Line 8 is control dependent on, and thus in the forward slice with respect to, Line 7). ◇

The final dependence graph based system is TAOS ("Testing with Analysis and Oracle Support") [17]. TAOS was build to support testing, debugging, and maintenance processes based on program dependence analysis. Its goal is to automate certain activities in testing a program because "the typical testing process is a human intensive activity and as such, it is usually unproductive, error prone, and often inadequately done" [17]. It uses oracles to determine the success of a test. In regression testing, existing tests and results of their execution can be used as an oracle against which to compare the results of testing modified.

The TAOS system represents procedures internally as program dependence graphs and can compute forward and backward static and dynamic intraprocedural slices. The focus of the paper is on the development

and the initial testing of a program. Very little detail on the use of the techniques for regression testing is given.

Richardson notes that "One common error made by maintenance engineers is to modify a definition because some use is incorrect, but not check the other uses of that definition. A forward slice from the modified definition identifies all the uses and thus all the definition-use pairs that need to be tested" [17]. These dependences should be retested after the modification to ensure that the change did not have an adverse affect on the rest of the program. The model discussed in the second technique of the next section avoids this situation and thus obviates the need for regression testing.

Future work includes plans to develop a "proactive regression testing process that uses program dependences to determine what must be retested, what tests cases and suites are available for reuse, and automatically initiates regression testing triggered by modification" [17]. It will be interesting to see how this process compares with the other techniques presented in this paper.

2.3 Data Flow Based Techniques

The final pair of techniques are based of Weiser's original data-flow model for computing program slices [21]. First, Gupta et al. describe a "slicing-based" approach to regression testing that explicitly detects du-associations that are affected by a program change [11]. The approach is self described as "slicing-based" because it does not directly use slicing operators, but rather is based on the slicing algorithms introduced by Weiser [21]. It identifies du-pairs that need retesting after a modification. (The paper uses du-triples, which include the variable that induced a du-pair. For consistency, this section uses du-pairs.) The motivating observation behind the approach is that if a test case never executes a modified statement then it never executes any code affected by a modified statement (for this particular execution of the program). Therefore, the technique concentrates on du-pairs that are affected by a change.

The approach is built on two algorithms that are used to determine which tests must be rerun after certain kinds of modifications. Both require as input a mapping between the statements of certified and modified. The first algorithm, BackwardWalk, finds all the definitions of a set of variables that reach a given point in the program. The second algorithm, ForwardWalk, identifies "uses of variables that are directly or indirectly affected by either a change in a value of a variable at a point in the program or a change in a predicate" [11]. It returns du-pair (d, u) if the value used at u is directly or indirectly affected at d by the change. A du-pair is directly affected if it represents a use of a modified definition. A du-pair is indirectly

affected in one of two ways: (1) it is in the transitive closure of a changed definition, or (2) its definition is control dependent on a changed or affected predicate [11]. This set should be the same as the set of du-pairs whose definition is in the forward slice with respect to the change.

BackwardWalk and ForwardWalk are used as building blocks to find all affected du-pairs based on the kind of low-level edit used to produce the change. Low level edits are the following: inserting a use of a variable in an assignment statement, deleting a use of a variable from an assignment statement, inserting a definition of a variable, deleting a definition of a variable, changing the operator in an assignment statement, changing the operator in a conditional, inserting an edge, and deleting an edge. The paper claims that all higher-level edits can be composed of these low-level edits.

For example, deleting a use of variable v from assignment statement s, requires retesting the du-pairs in ForwardWalk(s, v), while inserting a use of variable v in assignment statement s, requires testing the union of two sets. Assume that, after the edit, the statement is "y := ... v ...". The first part of the union is new associations cause by the addition of v. It contains du-pairs (d_i, s) where $d_i \in$ BackwardWalk(s, v). The second part of the union includes affected associations returned by ForwardWalk(s, y).

After a high-level edit, the du-pairs that must be retested are identified. This is done by running the algorithm on the sequence of low-level edits that make up the high-level edit. After all low-level edits have been processed, the set of all possible du-pairs that may require retesting are known. Note that later low-level edits may invalidate previous pairs that must be retested; thus, some cleanup is required. Testing is performed only once per high-level edit.

The algorithm concentrates on identifying the du-pairs that need to be tested. It can make use of existing test suites and select tests to cover these du-pairs, or a test-case generator can be employed. This represents a tradeoff between the expensive space cost of storing a test suite and the expensive time cost of generating test cases.

EXAMPLE. Consider adding a use of a to Line 10 of the example program. The set of du-pairs to be retested is the union of two sets. The first is all definitions of a in BackwardWalk([10], a). There is one such definition at Line 1. Line 10 assigns a value to area; the second part includes affected associations returned by ForwardWalk([10], area). The only use encountered is on Line 15.

As a second example, assume that Line 7 in certified had been "if a = b and b = a." The correction introduces a use of c in Line 7 and consequently BackwardWalk(7, c) identifies a new du-pair from the definition of c in Line 1. The change in the predicate affects the assignment to class and (indirectly) all the

statements of the case statement. ForwardWalk(7, c) identifies the du-pairs 8-9 and 8-15 for `class`, 10-15, 11-15, and 13-15 for `area`, and 12-13 for `s`. ◇

The paper states that the approach extends to interprocedural testing because Weiser's algorithm does. However, experience has shown that interprocedural extension of transitive closure algorithms is not always straightforward and often imprecise [13].

The final data-flow based technique, which is an alternative and novel method for reducing the cost of regression testing, is Gallagher and Lyle's limiting of software modification side effects [10]. Gallagher and Lyle introduce the notion of "decomposition slicing" and its use in "a new software maintenance process model which eliminates the need for regression testing" [10]. This model decomposes the program into two executable parts: a decomposition slice and its complement. Only the decomposition slice may be modified. After modification, it is merged with the complement. The key to the model is that the complement is semantically unchanged by the merge; thus, no regression testing is necessary (empirical verification would help acceptance of this model). By focusing on the decomposition slice, "a maintainer is able to modify existing code cleanly, in the sense that the changes can be assured to be completely contained in the modules under consideration and that no unseen linkages with other modules exist" [10].

The first step of the algorithm is to decompose the program into its *decomposition slices*. A decomposition slice captures all relevant computations involving a given variable. For variable v, a decomposition slice is defined as the union of all slices on v at each output statement where v is output together with the slice on v at the end of the program. Thus, a decomposition slice is the union of other slices, which is itself a slice.

The choice of definition for decomposition slice is not pivotal to the remainder of the approach. In fact if the program contains two independent uses of a variable, including both in a single decomposition slice may be undesirable. An alternative, for example, would use a forward slice from some key program point to identify statements of interest. The decomposition slice would then be the backward slice taken with respect to this set of statements. This example relates decomposition slicing to the Δ operator used by Binkley.

A program to be modified is decomposed into two parts: a maximal decomposition slice, which is to be modified, and its complement, which cannot be modified. The following terms are used to define these two parts of the program: a decomposition slice is *dependent* on another iff the two share a common statement. Decomposition slice s_1 is *strongly dependent* on decomposition slice s_2 if s_1 is a sub-slice of s_2. A *maximal decomposition slice* is not strongly dependent on any other decomposition slice. Statements in exactly one

decomposition slice are called *independent statements*. Finally, given a maximal decomposition slice $S(v)$ of program P, the complement of $S(v)$ is computed by removing from P the independent statements of $S(v)$.

Gallagher and Lyle provide a collection of rules that a maintainer must follow when modifying a decomposition slice. These rules are easily enforced by an editor. For example, "independent statements may be deleted from a decomposition slice" or "new control statements that surround (*i.e.*, control) any dependent statements are not allowed because they may cause the complement to change." The purpose of these rules is to ensure that the complement remains unchanged. Gallagher and Lyle discuss these rules and claim that the complete set of rules is necessary and sufficient.

After all changes to the maximal decomposition slice have been made and tested, it is merged with the complement. The merge is simple; it places statements in the place they would have appeared if the modification had been performed on the original program and not the decomposition slice.

To better understand this model, consider a program in which definition d has uses u_1, u_2 and u_3. Thus, there are du-pairs $(d, u_1), (d, u_2), (d, u_3)$. Assume it has been determined that u_1 is receiving an incorrect value from d. Consequently d is replaced by d'. To test the du-pairs in the new program $(d', u_1), (d', u_2), (d', u_3)$ must be tested. Using Gallagher's approach the existing definition d would be "protected" from change. This forces the introduction of a new variable into the program. For example, say d is the assignment statement "$x := 1$", and u_1 requires x to have the value 2. Rather that replacing "$x := 1$" with "$x := 2$" and retesting all three du-pairs, a new statement "$x' := 2$" is added immediately after "$x := 1$" and all occurrences of x in u_1 are replaced by x'. This avoids the need to retest the du-pairs (d, u_2) and (d, u_3).

EXAMPLE. Assume Line 15 of the example program was two lines, 15a and 15b, that output `area` and `class`, respectively. The decomposition slice on `class` includes Lines 1, 2, 3, 4, 5, 6, 7, 8, and 15a. (The decomposition slice on `area` includes all of the program except Line 15a.)

To add the `class` "`small`" (when `a`, `b`, and `c` are all less than 1), the decomposition slice with respect to `class` is computed. The only independent statement is the final output statement; thus, no changes to variable `class` are allowed. To effect this change, the statement "`output(class)`" would be deleted (independent statements can be deleted), and the following would be added after Line 14.

```
[14a]     if a < 1 and b < 1 and c < 1
[14b]         class' := small
[14c]     else
[14d]         class' := class
[15']     output(class')
```

◇

3 SUMMARY

Program slicing is a useful tool for working on the incremental regression testing problem. Unlike older approaches, which identify only directly affected components, such as du-pairs, slicing can identify indirectly affected "down-stream" components. Slicing can also be used to determine if two components have the same execution behavior.

Incremental regression testing attempts to find good approximate solutions to two unsolvable problems: determining the set of affected components and determining the set of tests that exercise these components. There is, however, no consensus on exactly what the ideal (undecidable) solutions to these problems should be.

As an example of lack of consistency in what the set of affected components should be, consider a change to Line 7 of the example program. The uses of `a`, `b`, and `c` on Line 7 are affected under most definitions given in Section 2. However, the techniques of Bates and Horwitz, Binkley, and Rothermel and Harrold do not select the du-pairs from Line 1 to these uses as needing to be retested, while the technique presented by Gupta et al. does. The data-flow involving these uses has not changed; however the control condition under which they are executed has. Whether or not they should be retested appears to be an open question. This lack of consistency makes finding, improving, and comparing algorithms for determining which components are affected difficult.

As an example of the lack of consistency in answers to the second question, consider whether the largest or smallest "set of tests that exercise these components" should be selected. In the approaches of Bates and Horwitz, and Binkley, for example, by taking one or all of the tests from each equivalence class it is possible to provide either alternative. Selecting the smallest set by taking one test is sufficient to provide adequate coverage. Selecting the largest set of all tests from each equivalence class does not improve coverage, however, it may help uncover faults caused by violations of the controlled regression testing assumption. Future work is necessary to determine if the additional testing produces any added benefit.

When dealing with any approximation to an undecidable problem, it is always possible (and sometimes desirable) to produce a more precise approximation. For example, Rothermel and Harrold provide an example in which the technique of Bates and Horwitz can select tests that cannot possibly traverse new or modified code. Correctly handling this example would make the algorithm more precise; but if, for example, the cost were too high, then the "improvement" may not be desirable.

Most of the techniques presented in Section 2 attempt to produce approximate answers to these two questions. Two techniques present alternative approaches. First, Gallagher and Lyle's technique, which isolates the affects of a change to a decomposition slice, removes the need for regression testing. Second, Binkley's technique, which uses differences in place of modified, reduces the cost of running a selected test case. Further empirical evidence is necessary to determine if these techniques are useful in practice.

Rothermel and Harrold's recent comparison of test case selection techniques [18] compares most of the techniques presented in Section 2. To do so, they define a framework that includes *inclusiveness*, *precision*, *efficiency*, and *generality*. They note that other definitions of inclusiveness and precision have been given. Future work on reducing the cost of regression testing will be simplified by standardized terminology.

Finally, for successful transfer to industry, incremental regression testing techniques need to be tried with "real" programs. Following convincing experimental evidence, their use in production environments is then necessary. They must demonstrate that they are able to detect bugs at least as well as the popular retest-all strategy and do so at a lower cost.

Presently the most promising route to a successful transfer is through the use of dynamic slicing. Algorithms based on dynamic slicing, such as that of Agrawal et al., are easier to scale to full programming languages than those based on static slicing because static slicing requires solving difficult data-flow problems. To accomplish the general transfer will first require agreement and then standardization on the goals of testing, regression testing and incremental regression testing.

References

[1] H. Agrawal and R.A. DeMillo. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Symposium on Testing and Verification*, October 1991.

[2] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, pages 246–256, New York, June

1990.

[3] H. Agrawal, J.R. Horgan, E.W. Krauser, and S.A. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance – 1993*, pages 1–10, 1993.

[4] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*. ACM, 1993.

[5] B. Beizer. *Software Testing Techniques*. van Nostrand Reinhold, second edition, 1990.

[6] D. Binkley. *Multi-procedure program integration*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1991.

[7] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):489–516, August 1997.

[8] D. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[9] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[10] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

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

[12] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, January 1990.

[14] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[15] T.J. Ostrand and E.J. Weyuker. Using data flow analysis for regression testing. In *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*, 1988.

[16] T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin – Madison, June 1988.

[17] D. Richardson. TAOS: Testing with analysis and oracle support. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA '94)*, August 1994.

[18] G. Rothermal and M.J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[19] G. Rothermel and M.J. Harrold. A safe, efficient regression test set selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–219, April 1997.

[20] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–84, August 1994.

[21] M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, May 1981.

[22] J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19:103–109, 1984.

[23] L. White. Software testing and verification. In *Advances in Computers*, volume 26(1), pages 335–390. Academic Press, 1987.