Semantics Guided Regression Test Cost Reduction

David Binkley[†]

Loyola College in Maryland

ABSTRACT

Software maintainers are faced with the task of regression testing: retesting a modified program on an often large number of test cases. The cost of regression testing can be reduced if the size of the program that must be retested is reduced and if old test cases and old test results can be reused. Two complimentary algorithms for reducing the cost of regression testing are presented. The first produces a program called `differences` that captures the *semantic* change between `certified`, a previously tested program, and `modified`, a changed version of `certified`. It is more efficient to test `differences`, because it omits unchanged computations. The program `differences` is computed using a combination of program slices.

The second algorithm identifies test cases for which `certified` and `modified` will produce the same output and existing test cases that will test components new in `modified`. Not rerunning test cases that produce the same output avoids unproductive testing; testing new components with existing test cases avoids the costly construction of new test cases. The second algorithm is based on the notion of *common execution patterns*, which is the interprocedural extension of the notion introduced by Bates and Horwitz. Program components with common execution patterns have the same execution pattern during some call to their procedure. They are computed using a new type of interprocedural slice called a *calling context slice*. Whereas an interprocedural slice includes the program components necessary to capture *all* possible executions of a statement, a calling context slice includes only those program components necessary to capture the execution of a statement in *a particular calling context* (*i.e.*, a particular call to the procedure).

Together with `differences`, it is possible to test `modified` by running the smaller program `differences` on a smaller number of test cases. This is more efficient than running `modified` on a large number of test cases. A prototype implementation has been built to examine and illustrate these algorithms.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*programmer workbench*; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs—*control structures, procedures, functions, and subroutines*; E.1 [**Data Structures**] *graphs*

Other Keywords: slicing, regression testing.

Author's address: Computer Science Department, Loyola College in Maryland, 4501 North Charles Street, Baltimore, Maryland 21210, 410-617-2881. email: binkley@cs.loyola.edu.

Preliminary versions of parts of this paper appeared in abridged form in the *Proceedings of the IEEE Conference on Software Maintenance* (Orlando, FL, Nov, 1992) [4] and *Proceedings of the IEEE International Conference on Software Maintenance* (Nice, France, Oct 1995) [8].

[†] supported in part by the David S. Lattanze Center for Executive Studies and National Science Foundation grant CCR-9411861.

1. INTRODUCTION

Software maintainers are faced with the task of regression testing: the process of retesting software after a modification. This process may involve running the modified program on a large number of test cases, even after a small change. Although the effort required to make a small change may be minimal, the effort required to recertify the modified program after such a change may be substantial.

In a recent analysis of regression test-case selection techniques, Rothermel and Harrold define regression testing as a task "performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program [36]." One form of regression testing focuses on the automatic selection of test cases based on the code of the original and modified programs. Such techniques are referred to as *code based* techniques. Rothermel and Harrold identify two kinds of code based techniques: *minimization* techniques attempt to test the changed parts of the program with a minimal number of tests while *safe coverage* techniques attempt to select all tests that may test the changed parts of the program.

This paper presents two code based algorithms for reducing the cost of regression testing. Both algorithms use (safe approximations to) language semantics to identify *run-time behavior* of program components (program behavior is formalized in Section 3). The first algorithm can reduce the complexity of the program on which test cases must be rerun: given a program `certified`, which passes some test suite, and a modified version of this program, `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 a code based test-case selection algorithm with both a minimization and a safe coverage variant. It can reduce the cost of testing by reusing test cases and test results from `certified`'s testing. Test case reuse involves identifying existing test cases that test new components in `modified`. Test results (*i.e.*, the fact the `certified` passed a test) are reused by identifying test cases that have the same behavior in `certified` and `modified`. Reusing test *cases* avoids the costly construction of new test cases. Reusing test *results* avoids the expense of running `modified` on test cases for which it can be *guaranteed* that `modified` and `certified` will produce the same results. Thus, rather than retesting the large program `modified` with a large number of test cases it is possible to certify `modified` by running the smaller program `differences` on a smaller number of test cases.

Recent trends in software engineering (*e.g.*, OOP), which encourage the use of many small procedures, make it essential that algorithms for reducing the cost of regression testing handle the interprocedural impact of changes. To identify reusable test cases and test results in a program with procedures and procedure calls, we introduce the notion of *common execution patterns*, an interprocedural extension of *equivalent execution patterns* [1]. Common execution patterns capture the *semantic* (not syntactic) differences and similarities between program components.

The algorithms presented in this paper make use of program slicing [38, 19]. A slice, taken with respect to a program component p and a variable v, includes all statements of the program that may affect the computation of v at p. To identify statements that have common execution patterns a new kind of interprocedural slice, called a *calling context slice*, is introduced. This slice captures the behavior of a statement on a particular invocation of its procedure. Thus, it contains fewer program components than an *inter*procedural slice, but more than an *intra*procedural slice. One advantage of this is that the algorithm can better handle multiple changes to `certified`. Previous approaches, for example [16, 29], make the assumption that a test case exercises the same components in `certified` and `modified`. However, a change in one computation might affect the path taken by a test case and invalidate this assumption.

`Certified`'s test cases are assumed to provide *adequate* coverage. A set of tests T is judged adequate with respect to some *test data adequacy criterion* if it satisfies some coverage metric for the program [1, 26, 10, 32, 21]. For example, one metric is the all-statement criteria. Test suite T satisfies this criteria for program P, if every reachable statement in P is executed by at least one of the tests in T. The test-case selection algorithm presented in this paper selects a reduced number of test cases by avoiding test cases that will produce the same outcome in `modified` and `certified`; thus, the algorithm assists the tester to achieve coverage.

The remainder of this paper is organized as follows: Section 2 presents some background material. Sections 3 and 4 present the main technical contributions of this paper: Section 3 defines "semantic difference" in the presence of procedures and procedure calls and describes the algorithm for computing `differences`. Section 4 describes the test-case selection algorithm. Illustrations of these algorithms are presented in Section 5 followed by a discussion of related work in Section 6. Finally, Section 7 contains a summary.

2. BACKGROUND

This section first discusses the controlled regression testing assumption [36]. It then discusses the language supported by the algorithms and that used in the examples. Finally, it provides background material on the system dependence graph (the intermediate representation use by both algorithms) and interprocedural slicing [19], and the use of test data adequacy criterion in testing software [39, 1].

2.1. Controlled Regression Testing Assumption

The controlled regression testing assumption deals with the relationship between a program's text, formal semantics, and runtime behavior. The algorithms given in Sections 3 and 4 relate the formal semantics of `certified`, `modified`, and `differences` based on their text. These relationships extend to runtime behavior only under a deterministic computational model where each time a program is executed on the same input, it produces the same output. This model may be violated, in practice, when a program is ported to a different processor or a machine with a different amount of memory. Furthermore, it may be violated on a single machine if the location at which a program is loaded changes the program's behavior or if `differences` requires less memory than `modified` and consequently passes a test on which `modified` would run out of memory.

To facilitate comparisons with other work, we reuse the following assumption (the notation is that used in this paper).

ASSUMPTION (Controlled Regression Testing Assumption [36]). When `modified` is tested with [test suite] 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.

In the sequel, we use the phrase "guarantees" to mean "guarantees given the controlled regressing testing assumption."

2.2. Language

The language supported by the algorithms presented in Sections 3 and 4 is not specified. The algorithms use program slicing as a primitive operation. They can be applied to any language for which a slicing algorithm exists. For example, a prototype of the differencing algorithm from Section 3 has been built for the C language using the slicer Unravel [25].

In contrast, to simplify the presentation, the language used in the examples is a flat imperative ("C"-like) language with recursive procedures where parameters are passed by value-result. It contains only simple control structures such as if statements and while loops. It contains only simple variables, no reference variables or pointers. Furthermore, we assume programs contain no calls to non-existent procedures (*i.e.*, that programs are complete) and have no global variables.

2.3. The System Dependence Graph and Interprocedural Slicing

Programs are represented by System Dependence Graphs (SDGs). An SDG is a collection of procedure dependence graphs (PDGs) connected by interprocedural control- and flow-dependence edges. The vertices of a PDG represent the components (statements and predicates) of the procedure. In addition, each PDG contains a distinguished vertex called the *entry vertex*. The edges of a PDG represent the control and data-flow dependences between components. In addition, at call sites, there are transitive dependence edges that summarize transitive dependences induced by called procedures. The dependence edges in a PDG are a safe approximation to the semantic dependences found in the program [31], which are in general not computable and therefore must be approximated.

In order to correctly handle input and output statements, input and output are modeled as streams: the statement "$print(stream, x)$" is treated as an assignment "$stream = append(stream, x)$." The use of streams connects (transitively) all print statements using the same *stream* with flow dependence edges, but allows different streams to be independent. Symmetrically, reading from an input stream removes a data value from the input stream, which modifies the stream. This stream model correctly handles changes such as the deletion of a print statement.

PDGs are connected at call-sites to form the SDG. Value-result parameter passing, which involves copying actuals to formals, invoking the called procedure, and then copying formals back to actuals, is represented by a set of vertices for each parameter. The copying of values between actual parameter a and formal parameter f is modeled using temporary variables f_{in} and f_{out} (f_{out} is included only if f is modified by the call) as four assignment vertices: an *actual-in* vertex labeled "$f_{in} := a$"; a *formal-in* vertex labeled "$f := f_{in}$"; a *formal-out* vertex labeled "$f_{out} := f$"; and an *actual-out* vertex labeled "$a := f_{out}$". Interprocedural flow dependence edges connect corresponding actual-in and formal-in vertices, and corresponding formal-out and actual-out vertices; an interprocedural control dependence edge connects each call-site vertex to the entry vertex of the called-procedure (see Figure 1).

Backward and forward slices are used by the algorithms presented in Sections 3 and 4. A *backward* slice of SDG G, taken with respect to a set of vertices S, contains those vertices of G whose components potentially affect the components represented in S. A backward slice can be computed using two passes over G. Pass 1, denoted by $b1(G, S)$, starts from all vertices in S and goes backwards (from target to source) along the edges of the SDG without descending into called procedures. Pass 2, denoted by $b2(G, S)$, starts from all vertices reached in Pass 1 and goes backwards along the edges of the SDG but does not ascend to calling procedures. The result of an interprocedural backwards slice is the set of vertices encountered during Pass 1 and Pass 2: $b(G, S) =_{df} b2(G, b1(G, S))$.

Example. Figure 1 shows a program and part of its SDG. The slice of this SDG taken with respect to formal-in vertex labeled "$y_{out} = y$" is shown in Figure 2. (The stream "standard-out" is assumed in all the examples.)

The significance of a backward slice is that it is a semantically meaningful decomposition of a program. This allows portions of a program's behavior to be identified, isolated, and compared using backward slices.

```
procedure Main                procedure A(x, y)           procedure Add(a, b)
  sum := 0                      call Add(x, y)              a := a + b
  i := 1                        call Add(y, 1)            return
  while i ≤ 10 do             return
    call A(sum, i)
  od
  print sum
end
```

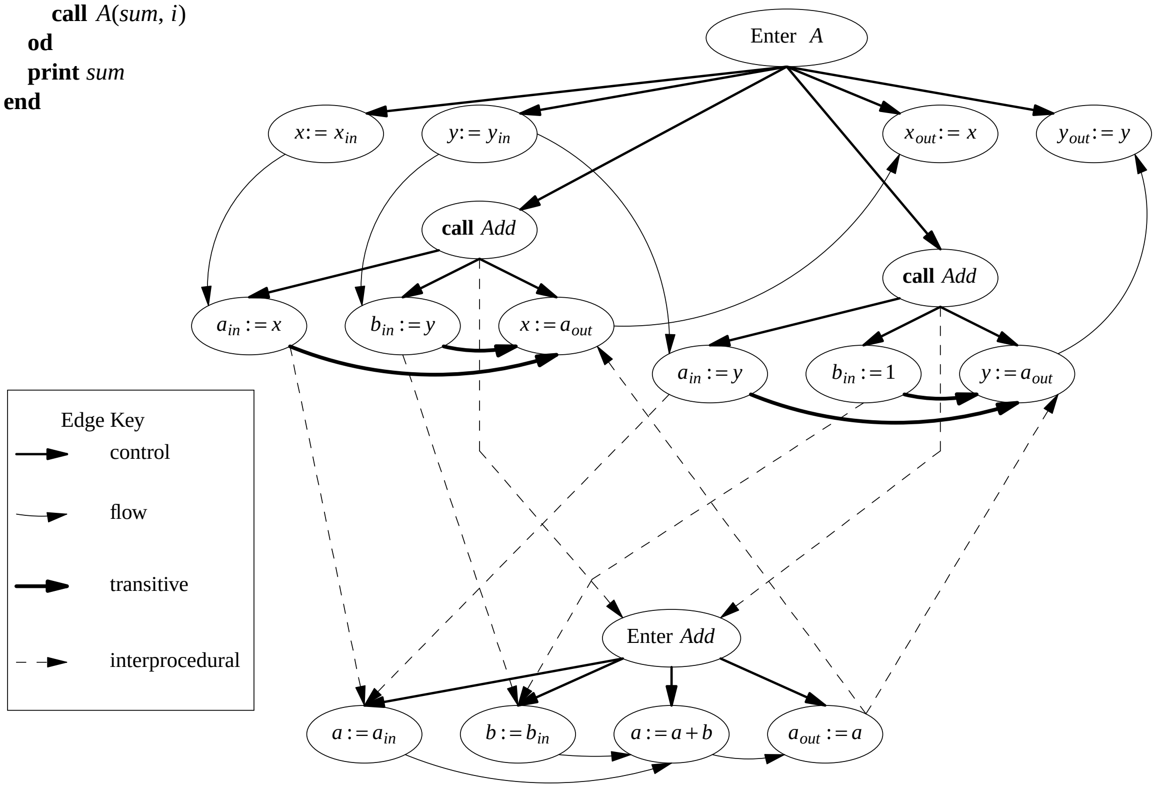


Figure 1. An example system that sums the numbers 1 to 10 and part of its SDG (only the PDGs for *A* and *Add* are shown).

A forward slice is the dual of the backward slice: whereas a backward slice includes those program components that potentially affect a given component, a forward slice includes those components that are potentially affected by a given component. As with backward slicing, an interprocedural forward slice of G taken with respect to S can be computed using two passes. In a forward slice, however, edges are traversed from source to target. Pass 1, denoted by $f1(G, S)$, starts from all vertices in S and goes forwards along the edges of the SDG but does not descend to called procedures. Pass 2, denoted by $f2(G, S)$, starts from all vertices reached in Pass 1 and goes forwards along the edges of the SDG without ascending to calling procedures. The result of an interprocedural forward slice is the set of vertices encountered during Pass 1 and Pass 2: $f(G, S) =_{df} f2(G, f1(G, S))$.

2.4. Test Data Adequacy Criteria

A test data adequacy criterion is a minimum standard that a test suite for a program must satisfy. An example is the *all-statements* criterion, which requires that all statements in a program must be executed by at

```
**procedure** *Main*

   $i := 1$
   **while** $i \le 10$ **do**
      **call** $A(i)$
   **od**

**end**

**procedure** $A(y)$

   **call** $Add(y, 1)$
**return**

**procedure** $Add(a, b)$
   $a := a + b$
**return**
```

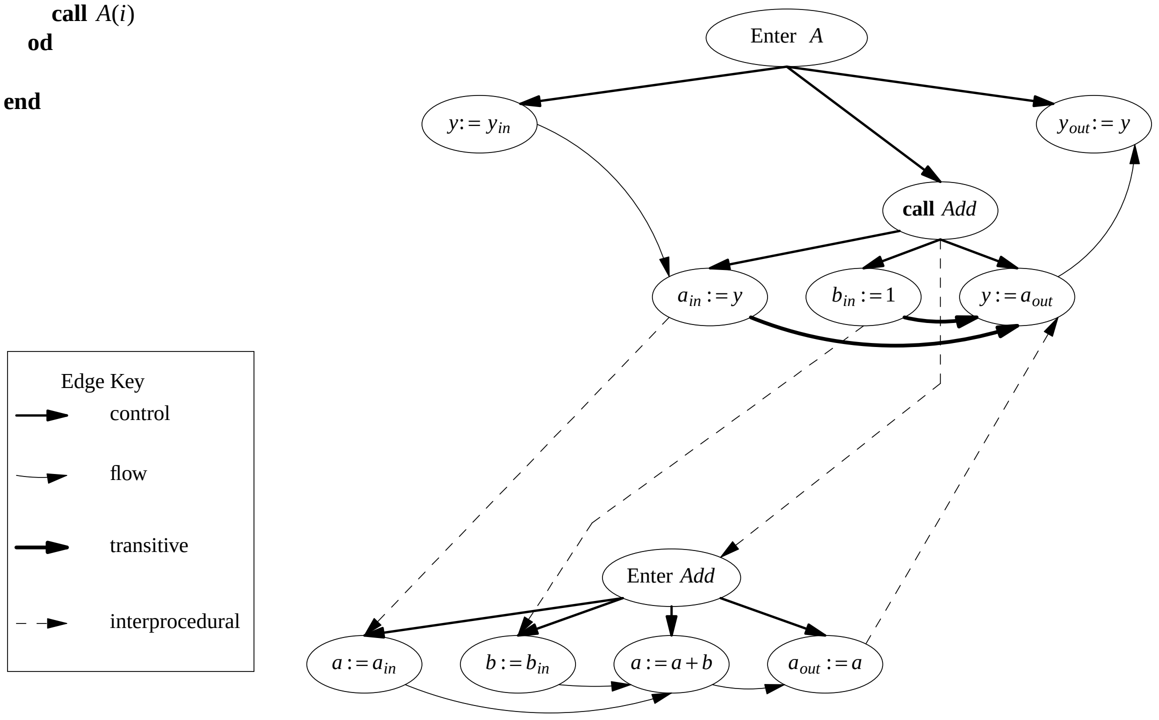


Figure 2. (Part of) the slice of the SDG in Figure 1 taken with respect to the actual-out vertex labeled "$y_{out} := y$" and the program to which is corresponds. This program contains only the looping control from the program in Figure 1.

least one test case in the test suite. Satisfying an adequacy criterion provides some confidence that the test suite does a reasonable job of testing the program.

Test suite adequacy criteria can be divided into at least three groups: control-flow based criteria (*e.g.*, all-statements), data-flow based criteria [39], and program dependence graph based criteria [1]. While the techniques discussed in Section 4 are applicable to any of these, they work more naturally with criteria from the dependence graph group. We therefore consider the *all-vertices* and *all-flow-edges* criteria in Section 4 as representative examples. These two are introduced below, but first, to relate them to the other groups, consider the following relationships [1]:

(1) The dependence graph criterion all-vertices is equivalent to the control-flow criterion all-statements.

(2) The all-flow-edges criterion subsumes the data-flow criterion all-c-uses/some-p-uses (c-uses are computational uses and p-uses are predicate uses).

DEFINITION (All-Vertices Criterion). The *all-vertices* criterion is satisfied by a set of test cases T if for each vertex v there is some test case t in T that exercises v. A vertex is *exercised* if its corresponding statement is exercised. A statement is *exercised* by test case t if it is executed when the program is run with input t.

DEFINITION (All-Flow-Edges Criterion). The *all-flow-edges* criterion is satisfied by a set of test cases T if for each flow edge e there is some test case t in T that exercises e. The flow edge $e = u \longrightarrow_f v$, where u represents an assignment to variable x, is *exercised* if u is exercised; then no definition of x that lies on the control flow graph path taken from u to v is exercised, and finally v is exercised.

3. COMPUTING `differences`

This section contains four technical contributions:

(1) A definition of "semantic difference" in the presence of procedures.

(2) An algorithm for computing *AP*(`modified`, `certified`): the set of *affected points*, those components of `modified` that may exhibit different behavior in `modified` and `certified`.

(3) An algorithm for computing Δ(`modified`, `certified`): the set of `modified`'s components needed to capture the behavior of the components in *AP*(`modified`, `certified`).

(4) An algorithm for constructing the program `differences`: an executable program that can be proven to capture the semantic differences between `modified` and `certified`. This proof is a slight modification of the correctness proof given in [3].

This section concludes with a recap of the algorithm and an example illustrating how `differences` alone can reduce the cost of regression testing.

3.1. Defining Semantic Difference and Affected Points

Before defining "semantic difference" it is necessary to identify a correspondence between the components of `certified` and `modified` and to define an appropriate language semantics. The components of a program are the parts of the program represented by vertices in the program's SDG. A correspondence between them can be obtained using a syntactic matching algorithm such as Yang's [43] or Laski and Szermer's [23]. It can also be maintained using a special editor that maintains statement tags. Such editors can be created by MENTOR [12], GANDALF [28], and the Synthesizer Generator [33]. The effect of the precision of this correspondence on the differencing algorithm is an area of future work. (Where necessary in the figures of this paper, this correspondence is represented by annotating programs with labels. See, for example, Figure 3.)

In addition to final output, we need to reason about the internal state of the computation; thus, the semantics of a program is defined in terms of the "sequence of values" produced by each component. The "sequence of values" produced by a program component means the following: for an assignment statement or parameter vertex, the sequence of values assigned to the target variable; for an input statement, the sequence of values read in; for a predicate, the sequence of boolean values to which the predicate evaluates; and, for an output statement, the sequence of values output.

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, programs 1 and 2 from Figure 3. Intuitively, these programs are semantically equivalent since the two calls to *Add* are independent. However, the sequences of values produced by the program components in *Add* depend on the order of the calls to *Add* and are consequently different.

In the presence of procedures, it is necessary to consider the sequence produced in different *calling contexts*:

Label	Program 1	Label	Program 2	Label	Common Procedure Add
	procedure *Main*		**procedure** *Main*		**procedure** $Add(a, b)$
L1:	**call** $Add(x, 1)$	L2:	**call** $Add(y, 2)$	L3:	$a := a + b$
L2:	**call** $Add(y, 2)$	L1:	**call** $Add(x, 1)$		**return**
	end		**end**		

Figure 3. Programs 1 and 2 are intuitively equivalent; however, the components in procedure *Add* produce different sequences of values.

DEFINITION (Calling Context). The *calling context* for procedure P (or a component in procedure P) is the sequence of call-sites that correspond to activation records currently on the stack when a particular activation of P is begun.

A more refined definition of program semantics, which accounts for calling context, is obtained using the concept of *roll-out*—the exhaustive in-line expansion of call statements to produce a program without procedure calls. Each expansion step replaces a call statement with a new *scope* statement that contains a copy of the body of the called procedure. Scope statements are parameterized by assignment statements that make explicit the transfer of values between actual and formal parameters. To preserve the correspondence between two programs, each statement of a new scope is given a compound label constructed from the label of the replaced call site and the label of the copied statement (see Figure 4). In the presence of recursion, roll-out leads to an *infinite* program. (The meaning of an infinite program is defined by the least upper bound of the meanings of the finite programs that approximate it [37].)

DEFINITION (Program Meaning). The *meaning* of program P maps each component of P to a set of sequences of values. For component c, this set contains the sequences produced by the occurrence of c in *roll-out*(P). Each element of this set is produced in a distinct calling context.

Because rolled-out programs have no procedure calls, the semantic differences between two rolled-out programs can be defined in terms of sequences of values. The semantics of two programs can be related by the semantics of their roll-outs because the *roll-out* operation is semantics-preserving [3]. It should be emphasized that no roll-outs, which may produce infinite programs, are actually performed. The roll-out concept is used only as a conceptual device to help formulate the semantic differences between two programs.

Example. Figure 4 shows the roll-outs of Programs 1 and 2 from Figure 3. For these programs, defining semantic difference using roll-out produces the intuitively correct result that the two are semantically equivalent. This is because corresponding occurrences of "$a := a + b$" (those with the same compound label) compute the same sequence of values.

We can now define the set of *affected points*.

DEFINITION (Affected Points). Component c of `modified` is an *affected point* iff it has an occurrence in *roll-out*(`modified`) with no corresponding occurrence in *roll-out*(`certified`) or a corresponding occurrence in *roll-out*(`certified`) that computes a different sequences of values when both programs are evaluated on the same input.

Label	*roll-out*(Program 1)	Label	*roll-out*(Program 2)
	procedure *Main*		**procedure** *Main*
L1:	**scope** *Add*($a := x$, $b := 1$;	L2:	**scope** *Add*($a := y$, $b := 2$;
	$x := a$)		$y := a$)
L1.L3:	$a := a + b$	L2.L3:	$a := a + b$
	epocs		**epocs**
L2:	**scope** *Add*($a := y$, $b := 2$;	L1:	**scope** *Add*($a := x$, $b := 1$;
	$y := a$)		$x := a$)
L2.L3:	$a := a + b$	L1.L3:	$a := a + b$
	epocs		**epocs**
	end		**end**

Figure 4. The roll-outs of Programs 1 and 2 from Figure 3. (Only the labels for the scope statements and occurrences of the assignment statement "$a := a + b$" are shown. The labels of the statements transferring values in to and out of a scope are not shown.)

To correctly account for the computations represented by affected points, it is necessary to partition this set into *strongly affected points* and *weakly affected points*. Whereas an affected point potentially exhibits changed behavior in *some* calling context, a *strongly affected point* potentially exhibits changed behavior in *all* calling contexts. 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*. 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:

DEFINITION (Strongly Affected Points). Component *c* in procedure *P* of `modified` is a *strongly affected point* iff there is no corresponding component in `certified` or corresponding occurrences of *c* in *roll-out*(`modified`) and *roll-out*(`certified`) compute different sequences of values when corresponding scopes for procedure *P* are invoked with the same initial state (*i.e.*, invoked with the same initial values for *P*'s formal parameters).

DEFINITION (Weakly Affected Points). Component *c* of `modified` is a *weakly affected point* if it is an affected point but not a strongly affected point.

3.2. Computing Affected Points

This section describes how to compute *safe* approximations to the sets of affected points, strongly affected points, and weakly affected points (denoted *AP*(`modified`, `certified`), *SAP*(`modified`, `certified`), and *WAP*(`modified`, `certified`), respectively). The sets *SAP*(`modified`, `certified`) and *WAP*(`modified`, `certified`) are then used in the next section to compute Δ(`modified`, `certified`) the sub-graph of `modified`'s SDG that captures the computation of the affected points.

Since determining any non-trivial property of a program is undecidable, any algorithm for identifying semantic differences must be approximate. The algorithms discussed in the paper are safe: they correctly identify all semantically changed components of the program but might also identify unchanged components of the program as changed. Any component not identified as changed is *guaranteed* to have the same behavior in `certified` and `modified`.

First, a safe approximation of the set of affected points is computed by taking an f (full forward) slice with respect to a special subset of the affected points called the *directly affected points* (DAPs):

DEFINITION.
$$DAP(\texttt{modified}, \texttt{certified}) =_{df} \{ v \in V(G_{\texttt{modified}}) \mid v \notin V(G_{\texttt{certified}}) \vee \text{ or } v \text{ has different intraprocedural edges in } G_{\texttt{modified}} \text{ and } G_{\texttt{certified}} \}.$$

For a discussion of why interprocedural edges are ignored in this definition see [7]. In addition to obvious cases of new components, edge changes result, for example, when a program component is moved from within a control structure to outside the control structure or from one branch of an if statement to the other. The first changes the source of the edge; the second changes the edge's label (from **true** to **false** or **false** to **true**).

The set $AP(\texttt{modified}, \texttt{certified})$ contains all the affected points:

DEFINITION.
$$AP(\texttt{modified}, \texttt{certified}) =_{df} f(G_{\texttt{modified}}, DAP(\texttt{modified}, \texttt{certified})).$$

Recall that a strongly affected point from procedure P is affected by a change in P or a procedure (transitively) called by P. Consistent with this observation, $SAP(\texttt{modified}, \texttt{certified})$ is defined using an *f1* (forward Pass 1) slice. An *f1* slice taken with respect to a vertex in P (or a procedure called by P) does not descend into called procedures. The set $SAP(\texttt{modified}, \texttt{certified})$ contains all the strongly affected points.

DEFINITION.
$$SAP(\texttt{modified}, \texttt{certified}) =_{df} f1(G_{\texttt{modified}}, DAP(\texttt{modified}, \texttt{certified})).$$

Finally, $WAP(\texttt{modified}, \texttt{certified})$ contains all the weakly affected points (the affected points that are not strongly affected):

DEFINITION.
$$WAP(\texttt{modified}, \texttt{certified}) =_{df} AP(\texttt{modified}, \texttt{certified}) - SAP(\texttt{modified}, \texttt{certified}).$$

3.3. Constructing $\Delta(\texttt{modified}, \texttt{certified})$

The operator Δ applied to `modified` and `certified` produces a subgraph of $G_{\texttt{modified}}$ containing all the components necessary to capture the behavior of the components in $AP(\texttt{modified}, \texttt{certified})$. As expressed below, Δ is defined in two parts: one part captures changes associated with strongly affected points; the other captures changes associated with weakly affected points:

(1) Because the execution behavior at each strongly affected point v is potentially modified in *every* calling context in which v is executed, it is necessary to incorporate *all* of v's possible calling contexts in Δ. This is accomplished by taking a b slice with respect to v.

(2) Because the execution behavior at each weakly affected point v is potentially modified only in *some* calling contexts in which v is executed, it is necessary only to incorporate *some* of v's possible calling contexts in Δ. Since the vertices of the call sites that make up the calling contexts in which v has potentially modified execution behavior are *themselves* affected points, it is only necessary to take a *b2* slice with respect to v.

This second point deserves some clarification. Suppose v is a weakly affected point. A *b2* slice with respect to v will only include vertices in P and procedures called by P. It does not include any vertices in procedures that call P. Initially this may seem incorrect because *some* calling context must have changed. However, at least one of the call site, actual-in, or actual-out vertices associated with any changed calling context would itself be an affected point; thus, any changed calling context for P will also be included in Δ as desired.

Putting the two parts of Δ together produces the following definition of $\Delta(\texttt{modified}, \texttt{certified})$.

DEFINITION.

$$\begin{aligned}\Delta(\texttt{modified}, \texttt{certified}) =_{df} \; & b(G_{\texttt{modified}}, SAP(\texttt{modified}, \texttt{certified})) \cup \\ & b2(G_{\texttt{modified}}, WAP(\texttt{modified}, \texttt{certified})).\end{aligned}$$

Operationally, each of the two main terms in the definition of Δ represents three linear-time passes over the SDG of `modified`. During each pass, only certain kinds of edges are traversed.

Example. Figure 5 shows the two parts of $\Delta(\texttt{modified}, \texttt{certified})$ computed from `modified` and `certified` shown in the figure. In this example, $DAP(\texttt{modified}, \texttt{certified})$ contains the new assignment statement "$t := 2$" and the actual-in vertex for t at the second call site on Q in P (this vertex has different incoming flow dependence edges: in $G_{\texttt{certified}}$ it has an edge from the vertex labeled "$t := 1$," while in $G_{\texttt{modified}}$ it has an edge from the vertex labeled "$t := 2$"). These two points are also the only strongly affected points; thus, the first part of $\Delta(\texttt{modified}, \texttt{certified})$,

`certified`	`modified`	$b(G_{\texttt{modified}}, SAP^{\dagger})$	$b2(G_{\texttt{modified}}, WAP^{\ddagger})$	$\Delta(\texttt{modified}, \texttt{certified})$
procedure *Main*	**procedure** *Main*	**procedure** *Main*		**procedure** *Main*
$a := 1$	$a := 1$			
$b := 2$	$b := 2$			
call $P(a)$	**call** $P(a)$	**call** $P()$		**call** $P()$
call $P(b)$	**call** $P(b)$	**call** $P()$		**call** $P()$
end	**end**	**end**		**end**
procedure $P(x)$	**procedure** $P(x)$	**procedure** $P()$		**procedure** $P()$
call $Q(x)$	**call** $Q(x)$			
$t := 1$	$\boxed{t := 2}$	$t := 2$		$t := 2$
call $Q(t)$	**call** $Q(t)$	**call** $Q(t)$		**call** $Q(t)$
$x := 2$	$x := 2$			
return	**return**	**return**		**return**
procedure $Q(z)$	**procedure** $Q(z)$		**procedure** $Q(z)$	**procedure** $Q(z)$
$t2 := z$	$t2 := z$		$t2 := z$	$t2 := z$
return	**return**		**return**	**return**

† $SAP(\texttt{modified}, \texttt{certified})$
‡ $WAP(\texttt{modified}, \texttt{certified})$

Figure 5. The third and fourth columns show the two parts of $\Delta(\texttt{modified}, \texttt{certified})$ computed from programs `certified` and `modified` shown in the first two columns. (The box indicates the modification made in `modified`.) The union of these two programs (really their SDGs) yields a program (SDG) that captures the changed computations of `modified` with respect to `certified`. This union is shown in the rightmost column.

$b(G_{\texttt{modified}}, \mathit{SAP}(\texttt{modified}, \texttt{certified}))$ includes *all* calling contexts for procedure P. The weakly affected points are the formal-in vertex for z in procedure Q, and the assignment statement "$t2 := z$". Therefore, the second part of $\Delta(\texttt{modified}, \texttt{certified})$, $b2(G_{\texttt{modified}}, \mathit{WAP}(\texttt{modified}, \texttt{certified}))$, includes the necessary parts of procedure Q without including any call sites on Q. Together these two parts capture all the affected calling contexts in `modified`: the one modified calling context for Q and all the calling contexts for P.

3.4. Computing `differences` from Δ

To produce the program `differences` from the SDG $\Delta(\texttt{modified}, \texttt{certified})$ requires making $\Delta(\texttt{modified}, \texttt{certified})$ *feasible*. (An infeasible SDG is not the SDG of any program.) This is done in two steps that remove interprocedural infeasibilities [5] and intraprocedural infeasibilities [18, 9]. Once $\Delta(\texttt{modified}, \texttt{certified})$ is feasible, it is reconstituted into a program that has $\Delta(\texttt{modified}, \texttt{certified})$ as its SDG. This is done by projecting the statements of `modified` that are represented in $\Delta(\texttt{modified}, \texttt{certified})$. In other words, the statements of `differences` are the statements of `modified` represented by vertices in $\Delta(\texttt{modified}, \texttt{certified})$ and these statements appear in `differences` in the same order and at the same nesting level as in `modified`.

3.5. Recap

Putting all the pieces together, a complete algorithm for computing the semantic differences between `modified` and `certified` appears in Figure 6. The significance of the program `differences` is that it can be used to reduce the cost of regression testing by reducing the size of the program that test cases must be run on. If $\Delta(\texttt{modified}, \texttt{certified})$ does not contain a parameter mismatch, then the differencing algorithm is a special case of the program integration algorithm developed in [7]. In this case, the proof of correctness for the integration algorithm implies the correctness of the differencing algorithm. Otherwise, if $\Delta(\texttt{modified}, \texttt{certified})$ contains a parameter mismatch, then minor modifications to the correctness proof for the integration algorithm imply the correctness of the differencing algorithm [5]. Illustration 1 in Section 5 demonstrates the use of `differences`.

```
function Difference(modified, certified) returns a program
declare
   A, Base, B, M : programs
   G : an SDG
begin
   G_certified := computeSDG(certified)
   G_modified := computeSDG(modified)
   G := Δ(modified, certified)
   if G is infeasible then
      G := augment(G)
   fi
   differences := ReconstituteProgram(G)
   return (differences)
end
```

Figure 6. The function *Difference* takes as input two programs `certified` and `modified` and produces the program `differences` that captures the semantic differences between `modified` and `certified`.

4. TEST CASE SELECTION

This section describes how test cases from `certified`'s test suite are selected. Before doing so this section first introduces the notion of *common execution patterns*. It then describes how the components of `modified` are partitioned and how these partitions are used to perform test-case selection. It then discusses the computation of these partitions using *calling-context slices*, and finally, presents a recap of the complete test-case selection algorithm. Program `differences` can be run with these tests to recertify `modified`.

4.1. Common Execution Patterns

Common execution patterns extend *equivalent execution patterns* [1] to programs that contain procedures and procedure calls.

DEFINITION (Equivalent Execution Patterns [1]). Components c_1 and c_2 have *equivalent execution patterns* iff c_1 and c_2 are exercised the same number of times on any given input.

This definition and the following definition of *common execution patterns* assume that both programs terminate normally. A program may fail to terminate normally if it contains a non-terminating loop or if a fault occurs, such as division by zero. If one program fails to terminate normally, it may not get (or get back to) the component being tested; thus, that component would execute fewer times in the non-terminating program. The extension of these definitions to handle the additional three cases involving one or both programs not terminating is straitforward. Bates and Horwitz detail this extension for equivalent execution patterns [1].

Equivalence of execution patterns is too strong in the presence of procedures; thus, equivalent execution patterns, while safe, prove too coarse in the presence of procedures and procedure calls. For example, if a call on procedure P is added, none of P's components can be safely determined to have equivalent execution patterns; however, they may have common execution patterns. Common execution patterns require equivalent execution patterns to exist in some (but not all) calling contexts:

DEFINITION (Common Execution Patterns). Components c_1 of procedure P_1 and c_2 of procedure P_2 have *common execution patterns* if there exists calling contexts CC_1 from P_1 and CC_2 from P_2 in which c_1 and c_2 have equivalent execution patterns.

Recall that a vertex is exercised when the corresponding statement is executed. A flow edge is exercised when the source and target of the flow edge are exercised and the definition at the source reaches the target.

4.2. PARTITIONING OF `modified`'s COMPONENTS

To facilitate the test-case selection algorithm developed in Section 4.3, the components of `modified` are divided into four partitions. It should be noted that the meaning of "component" varies depending on the testing criteria. For the two criteria considered below (all-vertices and all-flow edges), a component is a vertex or a flow-edge, respectively.

The four partitions are `affected`, `deleted`, `new`, and `preserved`. `New` and `deleted` components are easily computed given the mapping between the statements of `certified` and `modified`. The bulk of the work in the computation of the set `affected` is done in the computation of the set of affected points AP. Finally, the `preserved` components are those components that are not in `affected`, `deleted`, or `new`. For the all-vertices and all-flow-edges these sets are defined in Table 1.

Partition	All Vertices	All Flow Edges
`affected`	AP – `new`	$\{ u \longrightarrow_f v \mid v \in AP \}$ – `new`[†]
`new`	*VertexSet*(`modified`) – *VertexSet*(`certified`)	*FlowEdges*(`modified`) – *FlowEdges*(`certified`)
`deleted`	*VertexSet*(`certified`) – *VertexSet*(`modified`)	*FlowEdges*(`certified`) – *FlowEdges*(`modified`)
`preserved`	remaining vertices	remaining flow edges

[†]Note that with the all-flow-edges criteria, the computation of AP ensures that the target of any flow edge whose source is in AP is also in AP.

Table 1: partition definitions

Example. Figure 7 illustrates these four sets for the all-flow-edges criterion.

4.3. Test Case Selection

We now describe how test cases are selected for each of the four partitions.

Components of `preserved` and `deleted`

Test cases that only exercise `preserved` or `deleted` components need not be rerun (test cases that exercise only `deleted` components can be removed from the test suite). Assuming that for a small change, most test cases test `preserved` components, not rerunning these test cases should significantly reduce the cost of regression testing.

	`Certified`	`Modified`	Identified Sets
[1]	**read**(a)	**read**(a)	
[2]	**read**(b)	**read**(b)	`affected` $= \{ [1] \longrightarrow_f [4], [1] \longrightarrow_f [5], [2] \longrightarrow_f [4],$
[3]	**if** $(a < 0)$	**if** $(b < 0)$	
[4]	$x = 1 / pow(a, -b)$	$x = 1 / pow(a, -b)$	$[2] \longrightarrow_f [5], [4] \longrightarrow_f [6], [5] \longrightarrow_f [6] \}$
	else	**else**	`new` $= \{ [2] \longrightarrow_f [3] \}$
[5]	$x = pow(a, b)$	$x = pow(a, b)$	
	fi	**fi**	`deleted` $= \{ [1] \longrightarrow_f [3] \}$
[6]	**print** (x)	**print** (x)	`preserved` $= \emptyset$

Figure 7. Program "Certified" has a bug in it: a rather than b appears in the *if* statement (the auxiliary function *pow* takes only positive powers). For illustration purposes, assume the only directly affected point of `modified` is the predicate of the *if* statement (in fact statements [4] and [5] are also DAPs because they have new incoming control dependence edges). The affected points of `modified` are all the vertices subordinate to the *if* statement (which are reachable from the *if* statement via control dependence edges), and statement [6], which can be reached via a flow dependence edge from affected points [4] and [5]. Because their targets are affected points, all the flow dependence edges to the two calls on procedure *pow* are in `affected`.

Components of new

For new components, which by definition have no test case devised for them, it may still be possible to reuse certified's test cases. This is possible when, for example, a new vertex in a while-loop is tested by a test case for another vertex in the loop. More formally, if in some calling context *CC*, a component $new \in$ new and an existing component c have common execution patterns, then *new* is exercised by any test case t which exercises c. If no such component c exists then a new test case must be found (certified's test suite provides a place to start looking for such test cases).

Components of affected

The mistake most often made when considering an affected component a is to assume that a test case t that tests a in certified continues to test a in modified. The problem with this is that one of the changes to certified may have altered the components exercised by t. For example, replacing "≤" with "<" in "*if* $x \le 0$" changes the path through the if statement that a test case with $x = 0$ would take. To avoid this mistake, components are matched by common execution pattern to identify appropriate test cases. This is facilitated by the auxiliary function *exercises*:

DEFINITION (Exercises). *Exercises*(t) maps test case t to the set of components exercised when certified is run on t.

Similar to new components, identifying test cases for an affected component a begins by identifying test cases that exercise a component c of certified such that a and c have common execution patterns. Let T be the set of all test cases t for which $c \in exercises(t)$ and c and a have a common execution pattern.

All tests in T exercise a; however, not all test cases in T must be rerun. Component a is an affected point because its computation is reached by the computation of a directly affected point (DAP). Thus, test cases for modified that execute a, but do not execute a DAP need not be rerun. Only test cases that exercise both a and a DAP are chosen by the test case selection algorithm; test cases that involve a but no DAP will behave the same in certified and modified.

Example. Consider two calls on a procedure. If an affected component c in the called procedure is created by a change at the first call-site, then tests involving (only) calls from the second call-site need not be rerun.

Example. Consider the programs shown in Figure 8. In this example the vertex representing statement [9] is in affected. Although both test cases exercise this vertex, only test t_1 causes a DAP to be executed. Test case t_2 is guaranteed to produce the same result in certified and modified and therefore does not need to be rerun.

Note that if the statement "**read**(a)" is replaced by "$a = 2$" in Figure 8, then "**read**(b)" would be a directly affected point because of the stream model used for input. This not only illustrates the use of the stream model, but also demonstrates how the algorithm handles deletions ("**read**(a)" was deleted) from certified.

A further reduction in the number of test cases to be rerun is possible if, with each affected component, the set of the DAPs that cause the component to be in affected is kept. This set is used to weed out test cases that exercise c and only DAPs that do *not* affect c. Such test cases produce the same results in modified and certified. This further identification may not prove to be cost effective.

	Certified	Modified	Certified's Test Cases
[1]	**read**(a)	**read**(a)	
[2]	**read**(b)	**read**(b)	
[3]	**read**(c)	**read**(c)	$t_1 = 2 \quad -1$
[4]	**if** $(b < 0)$	**if** $(b < 0)$	(vertices tested) [1] – [6], and [9]
[5]	$x = pow(a, -b)$	$x = 1 / pow(a, -b)$	
[6]	$x = c * x$	$x = c * x$	$t_2 = 2 \quad 4$
	else	**else**	(vertices tested) [1] – [4], [7], [8], and [9]
[7]	$x = pow(a, b)$	$x = pow(a, b)$	
[8]	$x = c * x$	$x = c * x$	
	fi	**fi**	
[9]	**print** x	**print** x	

Figure 8. This example assumes all-vertices coverage. The "certified" program, which should print $c * a^b$, has a bug in it: when b is less than zero, the result of $pow(a, -b)$ should be inverted. The fix introduces DAPs [5] and [6] ([6] has different incoming flow dependence edges in `certified` and `modified`), and affected points [5], [6], and [9]. While the execution of test case t_2 includes affected point [9], this case need not be rerun because it tests [1], [2], [3], [4], [7], [8], and [9] which does not include a DAP.

4.4. Identifying Components with Common Execution Patterns

Identifying components with common execution patterns is undecidable; therefore, in practice we must find a safe approximation. This section describes one such approximation for partitioning the components of `certified` and `modified` into equivalence classes based on common execution patterns. The algorithms uses calling context slices:

DEFINITION (Calling-Context Slice). 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.

Thus, a calling-context slice contains less of the program than an interprocedural slice, but more of the program than an intraprocedural slice.

An algorithm that computes calling-context slices is shown in Figure 9. This algorithm uses *b2* (second pass) interprocedural slices, which include the necessary statements in a procedure (and called procedures), but ignore calling procedures. By repeatedly taking *b2* slices back through the call-sites that make up the calling-context, only those parts of the program that contribute to the execution of the statement represented by v in calling context CC are included.

Common execution patterns depend on the number of times a component is exercised in a given calling context. The number of times component c is exercised is determined by the behavior of the components on which c is control dependent. Calling context slices with respect to these components are used to identify components with common execution patterns as follows: for a vertex v, it is sufficient to capture the complete execution behavior of the vertex upon which v is control dependent. This is done by taking the calling-context slice with respect to v's control predecessor. For a flow edge $u \longrightarrow_f v$, it is necessary for u, v, and all intervening definitions of the variable defined at u to have equivalent execution patterns in the same common calling context. This is done by taking the union of the calling-context slices with respect to the control predecessors of these vertices using the same calling-context in each slice. It is shown in [6]

```
function CallingContextSlice($G$, $v$, $CC$) returns a set of vertices
declare
  $G$: an SDG
  $v$: a vertex from $G$
  $CC$: a calling context (a list of call-sites)
  $tmp$, $Answer$: sets of vertices of $G$
begin
  $Answer := \emptyset$
  $tmp := \{ v \}$

  while $CC \neq empty$ do
    $Answer := Answer \cup b2(G, tmp)$
    $tmp := \{ a \mid a$ is an actual-in vertex at call-site $head(CC)$ whose corresponding formal-in vertex is in $b2(G, tmp) \}$
    $CC := tail(CC)$
  od

  $Answer := Answer \cup b2(G, tmp)$
  return($Answer$)
end
```

Figure 9. Function CallingContextSlice returns all vertices of G that affect the behavior of v when v's procedure is called through the sequence of call-sites in calling context CC.

that two vertices (flow edges) whose control successors have isomorphic[1] calling context slices have common execution patterns.

4.5. Complete Algorithm

The complete algorithm for test-case selection is shown in Figure 10. It has two outputs: the subset of `certified`'s test cases to be rerun and the components of `modified` not guaranteed to be covered by any of `certified`'s test cases. Reusing test cases avoids the costly construction of new test cases. Reusing test results avoids the expense of testing `modified` on test cases for which it can be guaranteed that `modified` and `certified` will produce the same results.

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 they must be run on. The semantic properties of slices allow us to prove the algorithm's correctness [6].

[1] Two slices are isomorphic if their *induced subgraphs* are *isomorphic*. The subgraph of graph G induced by vertices V contains V and the edges of G whose endpoints are in V. Two (induced sub)graphs G_1 and G_2 are isomorphic iff the following conditions are satisfied:

(1) There is a 1-to-1 mapping g from the vertex set of G_1 onto the vertex set of G_2 and for every v in G_1, v and $g(v)$ have the same text.

(2) There is a 1-to-1 mapping h from the edge set of G_1 onto the edge set of G_2 and for every edge e in G_1, e and $h(e)$ are of the same type (*e.g.*, both control edges, or both flow edges, etc.) and have the same label.

(3) For every edge $v \longrightarrow u$ in G_1, $h(v \longrightarrow u) = g(v) \longrightarrow g(u)$.

When G_1 and G_2 are isomorphic or when we are trying to prove G_1 and G_2 are isomorphic, for brevity, we will say v and $g(v)$ are the *same* vertex and e and $h(e)$ are the *same* edge.

```
function TestCaseSelection(certified, modified, $T$) returns  the set of tests to be rerun and the set of
                                                              components of modified that need new tests
declare
  certified, modified: the SDG's for programs certified and modified
  $T$: certified's test suite
  $t$, $t'$: test cases
  $c$: a component of certified
  $m$: a component of modified
  TestsToRerun: the set of certified's test cases to be rerun
  ComponentsThatNeedTests:  the set of modified's components not guaranteed to be tested by any
                            of certified's test cases
begin
  TestsToRerun := $\emptyset$
  ComponentsThatNeedTests := $\emptyset$

  for each new or affected component of modified $m$ do
    Let $C = \{ c \in$ certified $\mid c$ and $m$ have common execution patterns $\}$
    if $C = \emptyset$ then
      Insert $m$ into ComponentsThatNeedTests
    else
      if no $c \in C$ exists such that $t' \in$ TestsToRerun and $c \in exercises(t')$ then
        Insert a test case $t \in T$ such that $c \in exercises(t)$ and $t$ exercises a DAP into TestsToRerun
      fi
    fi
  od

  return(TestsToRerun, ComponentsThatNeedTests)
end
```

Figure 10. Function TestCaseSelection returns the set of certified's test cases that must be rerun and the set of components from modified for which the algorithms can find no test cases in certified's test suite. As stated, the algorithms has quadratic worst case running time. This can be improved by computing, in a liner time preprocessing step, all components with common execution patterns. This computation marks certain directly affected points as "tainted" and then propagates taints using forward slices. The resulting algorithm has linear worst case running time.

The algorithm in Figure 10 is an example of a minimization technique [36]. It attempts to select a minimal set of tests that cover all modified components. In addition to minimization techniques, Rothermel and Harrold define safe coverage techniques as "selection algorithms that include 100% of the modification revealing tests (a test is modification revealing iff it causes the outputs of certified and modified to differ) [36]." A simple change to the above algorithm satisfies a similar kind of safety. The change collects *all* test cases that can be shown to test affected or new components of modified by replacing the innermost if statement in Figure 10 with

$$\mathit{TestsToRerun} = \mathit{TestsToRerun} \cup \{ t \mid c \in \mathit{exercises}(t) \text{ and } c \in C \}.$$

The resulting algorithm is not safe in the Rothermel and Harrold sence because of its treatment of deleted components. If a deleted component affect other components in certified that exists 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 by the modified algorithm, but are include by Rothermel and Harrold's definition of safe.

This change does not improve coverage, as selecting one test is sufficient to ensure every affected component that can be shown to be tested by an existing test case is tested. The larger test set may however help uncover faults caused by violations of the controlled regression testing assumption. Future empirical work is necessary to determine if the additional testing produces any added benefit.

5. ILLUSTRATIONS

This section presents three programs that illustrate `differences`, test-case selection, and their combination. Screen dumps from a prototype implementation are shown for each example. This prototype was written in C/C++ with a Tcl/Tk front end. After the illustrations, empirical results, obtained using the prototype, are presented.

5.1. Illustration 1

Figure 11 contains a modified version of the program in Figure 1 used to illustrate the computation of `differences`. `Modified` extends `certified` by adding the lines of procedure *main* involving *prod* and the procedure *Product*. Following the steps of the algorithm, the vertices representing these new statements are the directly affected points. They are also the strongly affected points. The new call on *add* in procedure *Product* causes the vertices representing procedure *add* to be weakly affected points. The full backward slice with respect to the strongly affected points includes the loop and the assignment to *i* from *main* ad the call to increment *i* in *A*. The second pass slice with respect to the weakly affected points includes procedure *add*, but does not ascend out of procedure *add*; thus, `differences` correctly omits the call "$add(x, y)$" in *A* and the computation of *sum* from *main*.

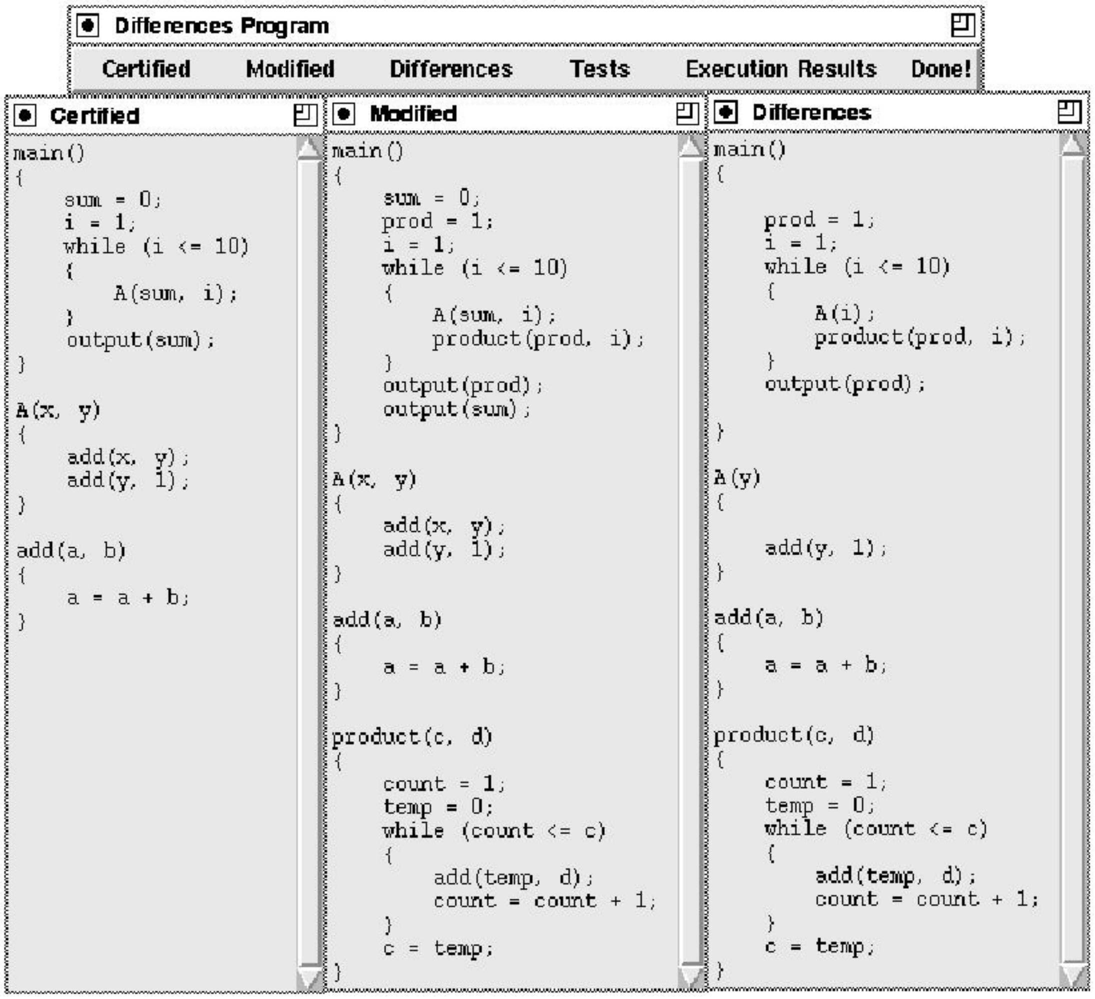
Even without performing test-case selection, the computation of `differences` tells us that the computations of *i* and *sum* are preserved. Thus, if `certified` was tested by two test cases, one testing the computation of *i* and the other testing the computation of *sum*, neither of these test cases need be re-run. `Modified` is *guaranteed* to produce the same output as `certified` for these test cases. All that is required is a test case for the computation of *prod*.

Looking ahead, test-case selection would determine that an existing test case tests all the new statements in *main* and all the statements in *Product* except those inside the while loop. The reason for this is that the call to *Product* in `modified` and the call to *A* in `certified` have common execution patterns. Thus any test case that tests the call to *A* in *main* also tests the call to *Product* and the "top level" statements of *Product*. The test-case selection algorithm cannot *guarantee* that the condition of the while loop will ever be **true**. In this example, it happens to be **true**; thus, the body of the loop is also tested.

5.2. Illustration 2

The second example illustrates the use of test-case selection on an example where `differences` fails to reduce the size of the program. The program, shown in Figure 12, has two computations whose results are joined. The computations compute the shipping cost and tax due on a purchase. A change in either computation affects the join point and thus causes `differences` to include both computations. (Note that for larger programs it should be increasingly unusual for `differences` to contain all of `modified`.)

In `modified` the shipping charge for packages between 5 and 25 dollars has increased from $3.00 to $3.50. Figure 12 includes five test cases (tests 1, 3, and 5 provide adequate all-vertices coverage). The affected and new components are the statements "*shipping_cost = 3.50*," "*total_cost = cost + shipping_cost + tax*," and "*output(total_cost)*." The statement "*shipping_cost = 3*" of `certified` and the statement "*shipping_cost = 3.50*" of `modified` have common execution patterns. The other affected



Certified

```
main()
{
    sum = 0;
    i = 1;
    while (i <= 10)
    {
        A(sum, i);
    }
    output(sum);
}

A(x, y)
{
    add(x, y);
    add(y, 1);
}

add(a, b)
{
    a = a + b;
}
```

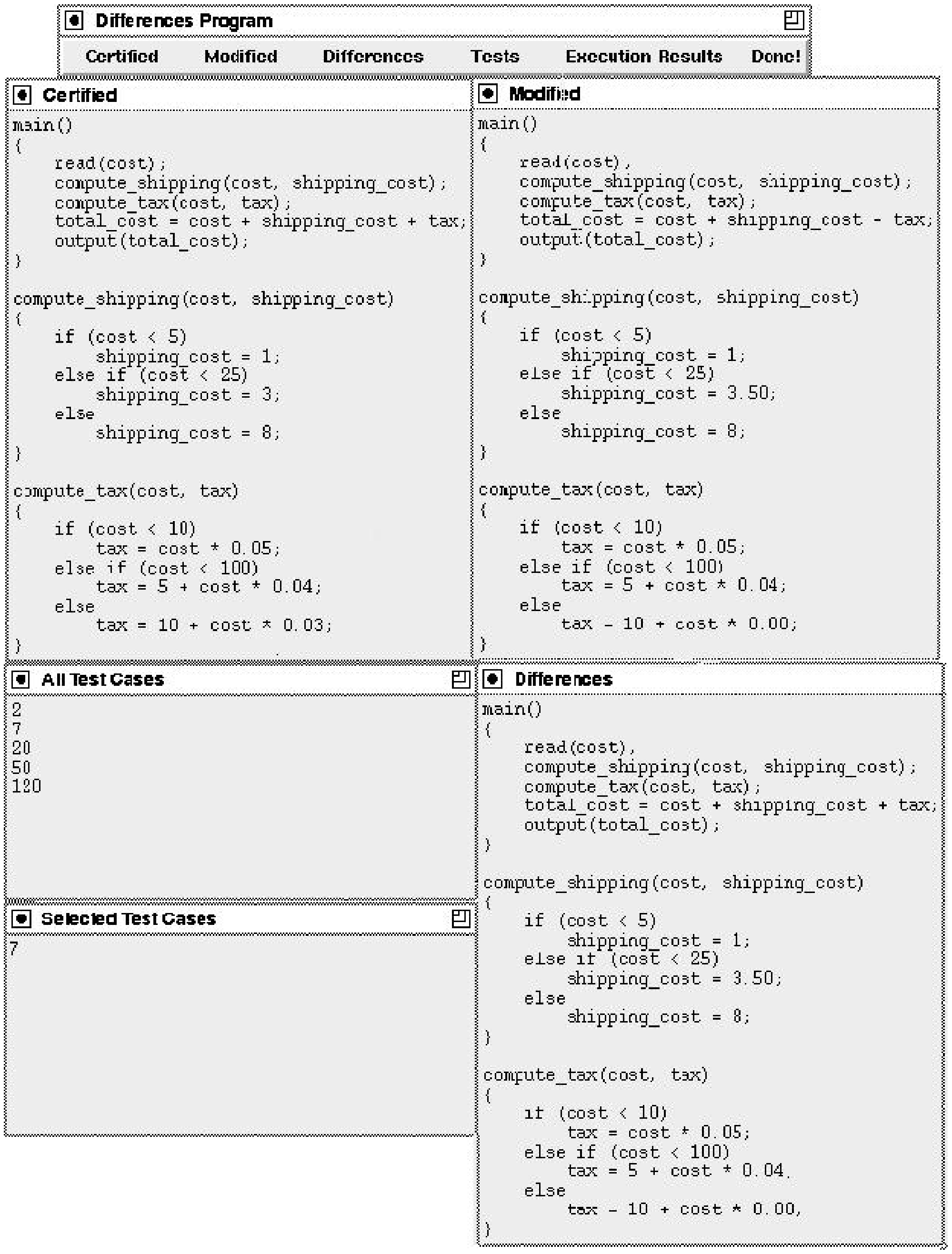
Modified

```
main()
{
    sum = 0;
    prod = 1;
    i = 1;
    while (i <= 10)
    {
        A(sum, i);
        product(prod, i);
    }
    output(prod);
    output(sum);
}

A(x, y)
{
    add(x, y);
    add(y, 1);
}

add(a, b)
{
    a = a + b;
}

product(c, d)
{
    count = 1;
    temp = 0;
    while (count <= c)
    {
        add(temp, d);
        count = count + 1;
    }
    c = temp;
}
```

Differences

```
main()
{

    prod = 1;
    i = 1;
    while (i <= 10)
    {
        A(i);
        product(prod, i);
    }
    output(prod);

}

A(y)
{

    add(y, 1);
}

add(a, b)
{
    a = a + b;
}

product(c, d)
{
    count = 1;
    temp = 0;
    while (count <= c)
    {
        add(temp, d);
        count = count + 1;
    }
    c = temp;
}
```

Figure 11. `Certified`, `modified`, and `differences` for Illustration 1.



Certified

```
main()
{
    read(cost);
    compute_shipping(cost, shipping_cost);
    compute_tax(cost, tax);
    total_cost = cost + shipping_cost + tax;
    output(total_cost);
}

compute_shipping(cost, shipping_cost)
{
    if (cost < 5)
        shipping_cost = 1;
    else if (cost < 25)
        shipping_cost = 3;
    else
        shipping_cost = 8;
}

compute_tax(cost, tax)
{
    if (cost < 10)
        tax = cost * 0.05;
    else if (cost < 100)
        tax = 5 + cost * 0.04;
    else
        tax = 10 + cost * 0.03;
}
```

Modified

```
main()
{
    read(cost);
    compute_shipping(cost, shipping_cost);
    compute_tax(cost, tax);
    total_cost = cost + shipping_cost + tax;
    output(total_cost);
}

compute_shipping(cost, shipping_cost)
{
    if (cost < 5)
        shipping_cost = 1;
    else if (cost < 25)
        shipping_cost = 3.50;
    else
        shipping_cost = 8;
}

compute_tax(cost, tax)
{
    if (cost < 10)
        tax = cost * 0.05;
    else if (cost < 100)
        tax = 5 + cost * 0.04;
    else
        tax = 10 + cost * 0.00;
}
```

All Test Cases

```
2
7
20
50
120
```

Selected Test Cases

```
7
```

Differences

```
main()
{
    read(cost);
    compute_shipping(cost, shipping_cost);
    compute_tax(cost, tax);
    total_cost = cost + shipping_cost + tax;
    output(total_cost);
}

compute_shipping(cost, shipping_cost)
{
    if (cost < 5)
        shipping_cost = 1;
    else if (cost < 25)
        shipping_cost = 3.50;
    else
        shipping_cost = 8;
}

compute_tax(cost, tax)
{
    if (cost < 10)
        tax = cost * 0.05;
    else if (cost < 100)
        tax = 5 + cost * 0.04;
    else
        tax = 10 + cost * 0.00;
}
```

Figure 12. `Certified`, `modified`, and `differences` for Illustration 2.

components of `modified` have common execution patterns with their corresponding component in `certified`. Test case 2 (input cost = 7) tests these three statements in `certified`. Test case selection correctly selects Test 2 as the only test case that needs to be re-run. It ignores the test cases for other shipping costs and the various tax brackets. (If Test 2 is omitted from the test suite, test-case selection correctly selects Test 3.)

5.3. Illustration 3

The third illustration is the word count program shown in Figure 13. This program reads a file and outputs the number of lines, words, and characters in the file. The "certified" version contains an error: the count of the number of lines is off by one. The modification that corrects this is a one line change in the initialization of the program. The computation of `differences` and test-case selection help to reduce the cost of testing `modified`.

Program `differences` captures the change made to *lines* in an executable program. In the computation of `differences`, the directly affected points are the new assignment statement "*lines = 0*" and the statements "*lines = lines + 1*" and "*output (lines)*", which have incoming flow dependence edges from "*lines = 0*." These three vertices are also the affected points (all are strongly affected). The backward slice with respect to these vertices includes the controlling while loop and the statements for reading input. The computation of the number of characters and number of words are not used in the computation of any affected points. Consequently, they are not included in `differences`.

Test-case selection also reduces the cost of testing `modified`. For each affected vertex of `modified` there is at least one vertex of `certified` with common execution behavior. (The new statement in `modified`, "*lines = 0*," has common execution patterns with any "top level" statement in `certified` (*e.g.*, "*inword = 0*" or "*lines = 1*"). Thus, no new test cases are needed. Of the test cases shown in Figure 13 (one input "file" is shown per line ending with a "\0"), tests 1 and 3 are selected. (In fact Test 3 is sufficient as Test 1 tests a subset of the vertices tested by Test 3. At present the implementation does not check for this subset situation.) None of the tests that examine the character counting or the paths through the word counting logic are selected.

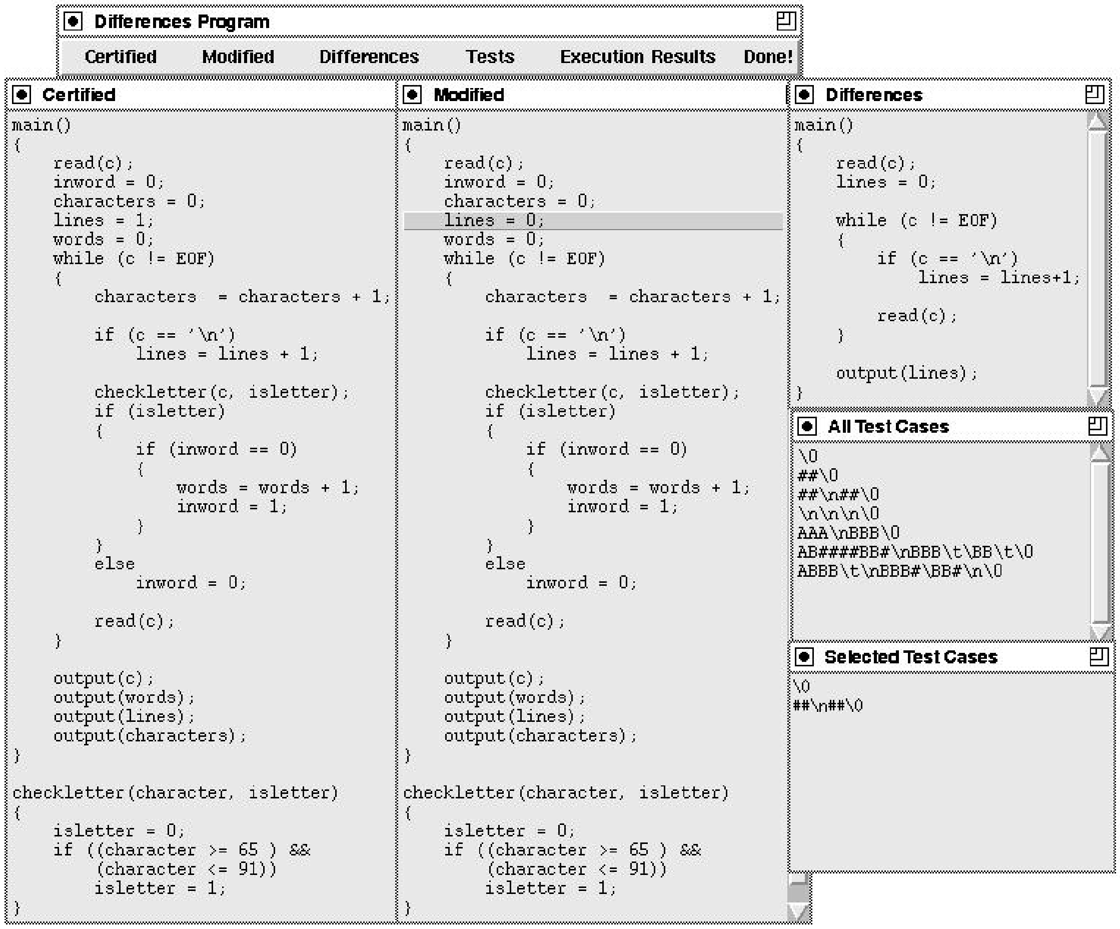
5.4. Empirical Results

Tables 2 and 3 below summarize results obtained when the prototype implementation was applied to five programs. The first three programs are the proceeding illustrations. The last two are representative samples of (1) programs that are computationally intensive, and (2) programs that contain complex control flow. The programs used are a program for predicting the weather, which mis-computes the next days expected wind speed, and an ALU emulator, which requires the correction of a bug in the increment instruction. These two are discussed in more detail at the end of this section.

The first table shows the size of `certified`, `modified`, and `differences` for each of the five programs. The size reduction ranges from 0% to 85%. For small changes, the following trend develops: as the size of `modified` increases, so does the percent reduction. For slicing this trend continues to much larger programs [25]. `Differences` is computed using slicing, so it is expected to follow this trend. For larger changes that can be broken down into a collection of smaller changes, the size of `differences` is proportional to the number of the small changes.

Program	`certified`	`modified`	`differences`	size reduction
Sum / Product	22	36	33	8%
Shipping Costs	28	28	28	0%
Word count	41	41	15	63%
Weather	251	251	132	47%
ALU	623	623	93	85%

Table 2: sizes of the example programs (in statements)



Certified

```
main()
{
    read(c);
    inword = 0;
    characters = 0;
    lines = 1;
    words = 0;
    while (c != EOF)
    {
        characters  = characters + 1;

        if (c == '\n')
            lines = lines + 1;

        checkletter(c, isletter);
        if (isletter)
        {
            if (inword == 0)
            {
                words = words + 1;
                inword = 1;
            }
        }
        else
            inword = 0;

        read(c);
    }

    output(c);
    output(words);
    output(lines);
    output(characters);
}

checkletter(character, isletter)
{
    isletter = 0;
    if ((character >= 65 ) &&
        (character <= 91))
        isletter = 1;
}
```

Modified

```
main()
{
    read(c);
    inword = 0;
    characters = 0;
    lines = 0;
    words = 0;
    while (c != EOF)
    {
        characters  = characters + 1;

        if (c == '\n')
            lines = lines + 1;

        checkletter(c, isletter);
        if (isletter)
        {
            if (inword == 0)
            {
                words = words + 1;
                inword = 1;
            }
        }
        else
            inword = 0;

        read(c);
    }

    output(c);
    output(words);
    output(lines);
    output(characters);
}

checkletter(character, isletter)
{
    isletter = 0;
    if ((character >= 65 ) &&
        (character <= 91))
        isletter = 1;
}
```

Differences

```
main()
{
    read(c);
    lines = 0;

    while (c != EOF)
    {
        if (c == '\n')
            lines = lines+1;

        read(c);
    }

    output(lines);
}
```

All Test Cases

```
\0
##\0
##\n##\0
\n\n\n\0
AAA\nBBB\0
AB####BB#\nBBB\t\BB\t\0
ABBB\t\nBBB#\BB#\n\0
```

Selected Test Cases

```
\0
##\n##\0
```

Figure 13. `Certified`, `modified`, and `differences` for Illustration 3. The change is hi-lighted in `modified`. (In the input files, the # character is used to represent a blank.)

The second table contains timing results, in milliseconds, for the five programs. The times were obtained on a DECstation 5000/133 with 16Mb of memory running Ultrix 4.3a. They are intended to give a rough idea of the algorithm's performance. These numbers do no represent a statistical study. Such a study is one area for future work (see Section 7). Note that differences of less than 10 milliseconds are meaningless due to the resolution of the timer. For example, in the shipping cost example `certified`, `modified`, and `differences` are all identical, but their times vary from 15 to 19 milliseconds. Note also that times do not include the time to build the SDG as we assume that it is stored and incrementally updated with the program as is done in a programming environment [30].

(1) Program	(2) `certified` on all tests	(3) `modified` on all tests	(4) computing `differences`	(5) `differences` on all tests	(6) test-case selection	(7) `differences` on selected tests
Sum / Product	47	109	16	105	23	67
Shipping Costs	15	19	8	16	27	4
Word count	121	102	12	23	20	19
Weather	75,434	107,027	117	29,996	168	29,789
ALU	2024	1797	489	844	949	27

Table 3: analyses times for the example programs (in milliseconds)

Table 3 can be used to compare the time taken to recertify `modified` by directly retesting `modified` and the time taken to recertify `modified` by testing `differences`. Figure 14 shows some of this data graphically. First consider using `differences` without test-case selection. This is done by comparing column 3 (`modified` on all tests) with the sum of columns 4 and 5 (*i.e.*, the cost of computing `differences` and then running it on all tests). The sum is smaller for the last three programs where `differences` contains less than half of `modified`. For the other two programs `differences` contains most of `modified` and thus the pay back for using `differences` does not outweigh the cost of computing it. The difference in time taken is dramatic for the computationally intensive weather program.

The second comparison adds the costs and benefits of test-case selection by comparing column 3 with the sum of columns 4, 6, and 7, which gives the cost of computing `differences`, selecting test cases, and then running `differences` on the selected test cases. (It is not necessary to include column 5 as `differences` is run on selected test cases only.) The sum is smaller for all but the second program. For three of the example programs this sum is greater than the cost of running differences on all tests. The sum of columns 4, 6, and 7 is greater than the sum of columns 4 and 5 for 3 of the 5 example programs. This indicates the time taken to select test cases is not paid back by running `differences` on fewer tests.

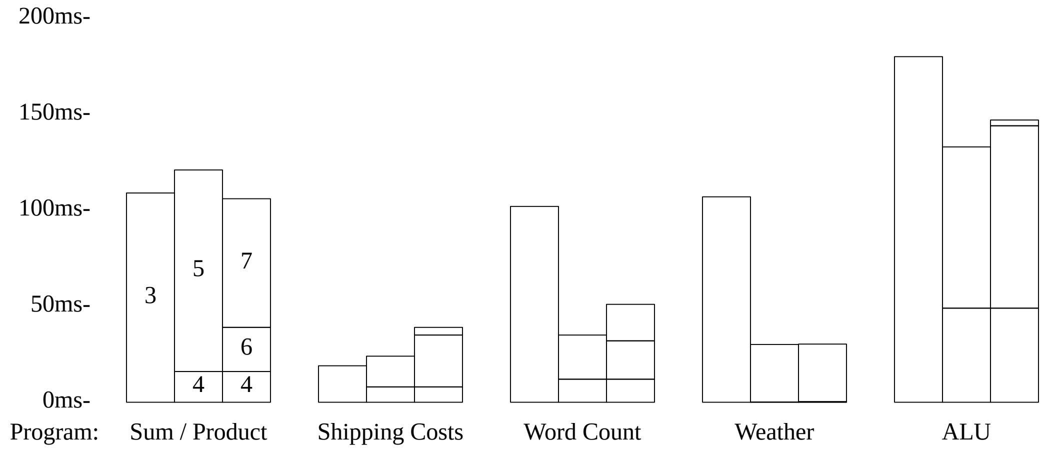


Figure 14. The execution times from Table 3 shown graphically. The Sum/Product graph includes the column numbers from Table 3 used for each bar. The data for Weather and ALU have been scaled by 1000 and 10 respectively.

This should be less true for programs with larger test suites. Further, experimentation on larger programs with larger test suites is needed to determine if this is true in practice.

An additional comparison provides some idea of the benefits of performing test-case selection. Compare column 5, which is the cost of running `differences` on all tests cases, with column 7, which is the cost of running `differences` on only the selected test cases. Test-case selection yields an improvement in all five cases.

Some explanation of the two programs not used as illustrations is in order. The weather program computes tomorrow's temperature, wind, and barometric pressure as the results of fixed point computations. The wind computation in `certified` stops prematurely. `Modified` corrects this problem and completes the computation (thus its increased running time). `Differences` omits the temperature and barometric pressure computations. Thus, running it on all tests cases reduces the testing time by about two thirds. Test case selection omits the test cases for these computations. Consequently `modified` on *selected* test cases (not shown in the table) and `differences` on *all* tests cases takes essentially the same time.

The ALU program emulates an ALU with 16 instructions. The test suite contains 32 test cases as some instructions contain multiple control-flow paths. The emulation of the increment instruction is incorrect in `certified` and corrected in `modified`. In addition to the corrected increment procedure, `differences` contains code shared by all instructions (*e.g.*, instruction de-code, code to perform bit shifts, etc.). When `differences` is run on all test cases, test cases that do not need to be re-run still exercise this shared code. Thus there is room for test-case selection to make an improvement. For this program, test-case selection correctly determines that no new test cases need to be developed and that only one of the 32 original test cases needs to be re-run. Running `differences` on only the selected test cases provides a substantial time reduction over running `differences` on all test cases.

6. RELATED WORK

This section has two parts. The first and smaller part deals with finding semantic differences; the second larger part deals with reducing the cost of coverage based regression testing. In order to concentrate on the aforementioned two kinds of related work, which more closely relate to the topic of this paper, this section does not discuss the more general topic of software testing. White provides a broader and more complete survey of software testing methods [40].

Several algorithms for computing semantic differences have been described [7, 18, 20, 42, 17]. The algorithm described by Binkley et al. [7] generalizes that described by Horwitz el al. [18]. It is the foundation of the semantic differencing algorithms presented in this paper. Both of these algorithms perform semantics based program integration: given a program *Base* and two variants, *A* and *B*, each created by modifying separate copies of *Base*, the goal of program integration is to determine whether the modifications interfere, and if they do not, to create an integrated program that incorporates changed *behavior* of *A* and *B* with respect to *Base* along with the *behavior* common to all three programs. The same Δ operator used in the computation of the program `differences`, is used to capture changed behavior: $\Delta(A, Base)$ captures the changed behavior of *A* with respect to *Base* and $\Delta(B, Base)$ captures the changed behavior of *B* with respect to *Base*. Reps and Horwitz overviewed program integration and the related uses of dependence graphs [20].

The potential usefulness of computing `differences` can be illustrated by comparing it with certain control-flow graph based techniques [2, 13, 29, 16]. The goal of one [2], for example, is to determine which paths in the control-flow graph are affected by a change. Unfortunately, control-flow-graph paths techniques can suffer because multiple computations can share a single control-flow graph path. For example, Procedure *A* in Figure 1 contains a single control-flow graph path but two computations (the new values of *x* and *y*). Since the need to retest the computation of *x* or *y* includes this path, it forces the need to retest both. This is observed in the following quotation:

> if, indeed, the first program block is modified by, for example, the addition of a variable initialization, then all the program paths will be modified · · · and will thus have to be retested. It may, however, be the case that only a small subset of these paths actually use the initialized variable [2].

Because dependence graphs "throw away" unnecessary sequencing information contained in control-flow graphs, the technique presented herein is capable of identifying the two distinct computations in Procedure *A* of Figure 1 and including only the affected one in `differences`.

An improvement to the single procedure program integration algorithm [42] leads to an improved algorithm for intraprocedural differencing [17]. These algorithms use a modified dependence graph called the *Program Representation Graph* (PRG), which combines features of the PDG and *Static Single Assignment* (SSA) form [11]. Even though PRG based algorithms better identify components with equivalent behavior in the absence of procedures and procedure calls, attempts to extend this work to handle procedures and procedure calls have thus far been unsuccessful.

The second kind of related work deals with reducing the cost of regression testing by performing test-case selection. Previous coverage based test-case selection algorithms can be viewed as optimistic solutions to the test-case selection problem because they assume that test cases exercise the same components in `certified` and `modified` (for example see [29] and [16]). This is in contrast to the pessimistic solution presented herein. Optimistic solutions make the assumption that a test case will exercise the same components in `certified` and `modified`. Since another change may affect the flow of control through `modified`, this assumption may be invalid. To handle cases in which this optimism is misguided these techniques run the test cases they believe are required and record the actual components tested. New test cases are then devised to cover untested components. In comparison, the pessimistic approach described in this paper deals with multiple changes to `certified` by partitioning components based on common execution patterns regardless of the number of changes made.

In terms of computational costs, the optimistic approach avoids the cost of determining components with common execution patterns, but incurs the costs of running unnecessary test cases and of creating and running new test cases. The cost of computing common execution patterns is bounded by the size of the program. The cost of running test cases is unbounded in the program's size; it is bounded only by the execution time of the program. Empirical experience would be useful in further comparing these two possibilities.

One advantage of the optimistic approach is when the pessimistic approach cannot safely show that any test tests a component even though such a test exists. For example, consider replacing "**if** $a > 0$" with "**if** $a \geq 0$." This change may affect the direction taken by the if statement; thus, the pessimistic approach cannot safely determine that the same path is taken. But it does not affect the direction for a test with $a = 10$. Here the optimistic approach would discover this when it ran the $a = 10$ test. The pessimistic approach will identify components in the if statement's body as needing new test cases. One place to search for new test cases is in `certified`'s test suite; for example, the test with $a = 10$.

Many previous approaches account for direct changes to the program but may miss indirect ones. Consider the following problem:

> Many debugging or updating fixes involve only simple changes · · ·. One frequently made error is to change a variable assignment because the [right-hand-side expression] is incorrect, but not to check that the new

[right-hand-side] expression is appropriate for all the [left-hand-size] variables' uses. If the results are used in several places, it is easy for the programmer to concentrate on only one or two of those uses, and not to be aware of the uses in other places [29].

Retesting of directly affected definition-use pairs (flow dependence edges) is sufficient if the use is directly affected by the change. However, a change may affect a use "down stream" in the computation. The techniques presented in this paper correctly identify affected down-stream uses.

Rothermel and Harrold recently compared 13 regression test cost reduction techniques using four metrics: inclusiveness, precision, efficiency (complexity) and generality [36]. The following discussion of related work does not attempt to duplicate their work. Rather, we consider first the test-case selection algorithm of Rothermel and Harrold [34, 35], which produces results very similar to the test-case selection algorithm presented in Section 4, and then three other algorithms that use program slicing.

Rothermel and Harrold's first test-case selection method concentrates on the use of control dependence to identify cases that must be rerun [34]. The algorithm does not directly incorporate the effects of changes in data dependence when identifying changes and their effects. To capture semantic changes and thus provide coverage guarantees, it is necessary to consider both control and data dependence changes.

An extension of their work [35], which accounts for both control and data dependence, first considers intraprocedural test-case selection and then interprocedural test-case selection. This new work makes several insightful observations about reducing the cost of retesting software. Both algorithms begin with the enter vertices of the (main) procedures of `certified` and `modified` and perform side-by-side post-order traversals of the two graphs. During the traversals, tests to be rerun are collected. The traversals stop at leaf nodes or predicates that are directly affected points. A second pass is made over `modified`'s graph to discover components that are not tested. This pass uses slicing to identify untested components.

Their extension to interprocedural test-case selection essentially applies the intraprocedural test-case selection algorithm to (a minor modification of) the SDG. This extension is less precise (selects a larger number of tests) than the approach presented herein because of approximations in certain cases involving interprocedural data dependences present in the modified SDG. The interprocedural coverage algorithm represents a more substantial overhaul of the intraprocedural algorithm and is correspondingly more complex. Their approach and the one considered in Section 4 produce very similar results, but contain radically different internals. While a thorough comparison would require stating Rothermel and Harrold's algorithm in detail, the following two differences are representative.

First, the approach in this paper formalizes "exercised flow dependence edge" better and consequently deals better with identifying `new`, `affected`, `preserved`, and `deleted` flow dependence edges. This leads to better identification of which flow dependence edges need to be re-tested. For example, when an intervening definition (nested within a control predicate) is executed, the definition of *exercises* correctly captures only test cases that do not execute this intervening definition.

Second, the statement of the algorithm developed in this paper makes use of higher-level operators (*e.g.*, slicing operators). This allows the hard problem of performing test-case selection to be separated from the hard problem of capturing semantic properties of programs using dependences. In contrast, Rothermel and Harrold's algorithm considers individual data and control dependences directly in the test-case selection algorithm, which is in general more complicated and thus more error prone.

Three techniques that make use of program slicing are presented by Gupta et al. [15], Bates and Horwitz [1], and Kamkar et al. [22]. The first of these makes the observation that existing techniques identify directly affected def-use pairs, but not indirectly affected def-use pairs [15] (A def-use pair contains the same information as a flow dependence edge.) It then goes on to use program slicing to capture both directly and indirectly affected def-use pairs, which are categorized as *new*, *value* (*affected* in the terminology of this paper), and *path* (a kind of transitive dependence). Finally, it presents several algorithms including an algorithm for identifying the definitions that reach a given statement.

Bates and Horwitz [1] proposed the use of dependence graph based test data adequacy criterion. Their algorithms are efficient since slicing a dependence graph is a linear time operation. Bates and Horwitz introduce a number of test data adequacy criterion based on dependence graphs and then relate them to control flow and data flow criterion. Finally, they present algorithms for test-case selection based on their new criterion.

Third, Kamkar et al. apply dynamic slicing to interprocedural data flow testing [22]. Their goal is to increase the reliability of testing and not test case reduction. The use of dynamic slicing in place of static slicing should be considered. Whereas a static slice uses static analysis to determine dependences, a dynamic slice uses a particular execution of a program. This allows a more precise identification of dependences (in particular data dependences) because information about aliases and the values of predicates are known at run time. However, a dynamic slice applies to a particular run of a *particular* program. The ramification to test-case selection, after a modification to a certified program, is that a change in the certified program may change the statements executed by a test case; therefore, changing the dynamic slice. Thus, while dynamic slicing is useful in increasing the reliability of testing, it cannot be applied to the test-case selection problem.

7. SUMMARY AND FUTURE WORK

Knowing the semantic differences between two programs is useful in many program maintenance activities, not the least of which is reducing the cost of regression testing. The techniques described in this paper can reduce costs in three ways: they can reduce the complexity of the program on which tests must be run, they can reduce the number of existing test cases that must rerun, and they can reduce the number of new test cases that must be created. This algorithm is an important improvement over previous algorithms in one or more of the following three ways:

(1) It uses semantic changes rather than syntactic changes to identify affected components of the program and the test cases that must be rerun. To be clear, the semantics of a program are approximated by our techniques. Program semantics form the basis for the definitions of dependence, program slicing, common execution behavior, and `differences`. However, the actual algorithms, which are based on the semantic definitions, must necessarily work with programs. These algorithms compute safe approximations to the semantic definitions. For example, if the algorithm in Figure 9 determines that two components have isomorphic calling-context slices (a syntactic condition) then the components have common execution patterns (a semantic condition). However, if the algorithm cannot determine that two components have isomorphic calling-context slices, then it makes the safe assumption that they do not have common execution patterns. Erring on the safe side is necessary, since exact static analysis of semantic properties is an unsolvable problem.

(2) On test cases that the algorithm determines do not need to be rerun, it *guarantees* that `certified` and `modified` will have the same behavior. A proof appears in [6].

(3) It works in the presence of procedures and procedure calls, which is increasingly important as paradigms such as object-oriented programming produce programs with large numbers of procedures.

We also defined the notion of common execution pattern, which is useful in understanding which test cases for `certified` test components of `modified`. The algorithm for calling-context slices provides a method of determining when two components have common execution patterns.

Most of the algorithms presented in this paper are based on the operation of program slicing. We have constructed a prototype implementation for a subset of C that uses a simple slicer. We are exploring an implementation of these algorithms using Unravel (an ANSI C slicer) [25]. This would allow us to study larger "real-world" programs. Such a study could be carried out using the techniques presented in [24], which describes a model for comparing the cost of the selective regression testing strategies with traditional retest-all strategy.

In particular, the following tradeoffs will be explored. First, the tradeoff between minimal coverage and safe coverage. Minimal techniques are expected to select fewer test cases. We may find that minimal techniques do nearly as well at detecting faults in practice as safe coverage techniques, but save a great deal of time over safe coverage techniques. Or, we my find that minimal approaches detect far fewer faults than safe coverage techniques, but are a cheap way to get some testing done in certain situations. Some evidence that minimization techniques may be "just as good" as safe techniques is given in a study of minimization and safe coverage techniques perform by Wong et al. [41].

Two comparisons are needed. The first will compare the number of faults detected by minimization and by safe coverage techniques. The number of faults detected for minimization techniques should be less than or equal to the number detected by safe techniques. If it is less, then a comparison of the cost of missing faults with the cost of running additional tests is necessary.

The implementation will also be used to access the significance of the controlled regression testing assumption. If a significant number of faults are related to factors such as memory placement and available memory then selective regression testing of `differences` is of limited use. If, on the other hand, all or even most faults are found using selected tests, then selective regression testing becomes more attractive.

In particular, the implementation will be used to access the impact of testing `differences` in place of `modified`. An example of a failure not caught when testing `differences` is a test on which `modified` would run out of memory but `differences` does not. This study will be a comparison between the savings gained by running `differences` in place of `modified`, and the cost of missing a fault that would have been caught by testing `modified` directly. By analogy, early compilers were viewed with skepticism, but ultimately prove to be cost savers and thus are in common use. If techniques such as the selective regression testing are cost savers, they too will come into common use. This may require significant empirical evidence to build the necessary confidence in the new approach.

In conclusion, the algorithms presented in this paper allow a smaller number of test cases to be run on a smaller program. These algorithms are expected to work best on small to moderate changes of large programs where high cost may make maintainers reluctant to perform the regression testing. Consider the following remark by Brian Marick [27]:

> While the long-term benefits of an automated test suite are enormous, the startup cost can be high. For example, testing individual routines in isolation (unit testing) is too expensive except for most critical routines. You spend too much time writing test drivers and the stubs that emulate the subroutines the routine-under-test calls. $\cdots$ Initial test development is bad enough; maintenance becomes a nightmare, and the unit tests are often abandoned.

Having a lower cost method that guarantees the same testing coverage as complete regression testing could remove this reluctance.

Acknowledgments

The limiting of tests to directly affected points was suggested to the author by Gregg Rothermel while the author was visiting Clemson University. Ned Chapin provided early guidance and encouragement. Ed Birrane is responsible for the Tcl/Tk front end. Discussions with Keith Gallagher improved this paper. Roger Eastman, Brad Kuhn, and Ed Birrane's comments also improved the exposition.

REFERENCES

1. Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," in *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, (Charleston, SC, January 10-13, 1993), ACM, New York, NY (1993).
2. Benedusi, P., Cimitile, A., and De Carlini, U., "Post-maintenance testing based on path change analysis," pp. 352-368 in *Proceedings of the IEEE Conference on Software Maintenance,* (Phoenix, Arizona, Oct, 1988), IEEE Computer Society, Washington, DC (1988).
3. Binkley, D., "Multi-procedure program integration.," Ph.D. dissertation and Technical Report TR-1038, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).
4. Binkley, D., "Using semantic differencing to reduce the cost of regression testing," pp. 41-50 in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, FL, Nov, 1992), IEEE Computer Society, Washington, DC (1992).
5. Binkley, D., "Precise executable interprocedural slices," *ACM Letters on Programming Languages and Systems* **2**(1-4)(1993).
6. Binkley, D., "Interprocedural test case selection," Tech Rep. TR95001, Department of Computer Science, Loyola College, Baltimore, MD. (January 1995).
7. Binkley, D., Horwitz, S., and Reps, T., "Program Integration for Languages with Procedure Calls," *ACM Transactions on Software Engineering and Methodology* **4**(1) pp. 3-35 (January 1995).
8. Binkley, D., "Reducing the cost of regression testing by semantics guided test case selection," pp. 251-260 in *IEEE International Conference on Software Maintenance,* (Nice, France, Oct, 1995), IEEE Computer Society, Washington, DC (1995).
9. Choi, J. and Ferrante, J., "Static slicing in the presence of goto statements," *ACM Transacton on Programming Languages and Systems* **16**(4) pp. 1097-1113 (July 1991).
10. Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J., "A formal evaluation of data flow path selection criteria," *IEEE Transactions on Software Engineering* **SE-15**(11) pp. 1318-1332 (November 1989).
11. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
12. Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., "Programming environments based on structured editors: The MENTOR experience," pp. 128-140 in *Interactive Programming Environments*, ed. D. Barstow, E. Sandewall, and H. Shrobe, McGraw-Hill, New York, NY (1984).
13. Fischer, K.F., Raji, F., and Chruscicki, A., "A methodology for re-testing modified software," pp. B6.3.1-6 in *IEEE National Telecommunications Conference Proceedings*, (Nov. 1981).
14. Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," *IEEE Transactions on Software Engineering* **SE-17**(8) pp. 751-761 (1991).
15. Gupta, R., Harrold, M.J., and Soffa, M.L., "An approach to regression testing using slicing," pp. 299-308 in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, Florida, Nov, 1992), IEEE Computer Society, Washington, DC (1992).
16. Harrold, M.J. and Soffa, M.L., "An incremental approach to unit testing during maintenance," pp. 362-367 in *Proceedings of the IEEE Conference on Software Maintenance,* (Phoenix, Arizona, Oct, 1988), IEEE Computer

Society, Washington, DC (1988).

17. Horwitz, S., “Identifying the semantic and textual differences between two versions of a program,” *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 20-22, 1990)*, ACM SIGPLAN Notices* **25**(6) pp. 234-245 (June 1989).

18. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

19. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).

20. Horwitz, S. and Reps, T., “The use of program dependence graphs in software engineering,” in *Proceedings of the 14th IEEE Conference on Software Engineering,* (Melbourne, Australia, June, 1992), IEEE Computer Society, Washington, DC (1992).

21. Howden, W.E., “Reliability of the path analysis testing strategy,” *IEEE Transactions on Software Engineering* **SE-2**(3) pp. 208-215 (September 1976).

22. Kamkar, M., Fritzson, P., and Shahmehri, N., “Interprocedural dynamic slicing applied to interprocedural data flow testing,” pp. 368-395 in *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Quebec, Canada, Sep, 1993), IEEE Computer Society, Washington, DC (1993).

23. Laski, J. and Szermer, W., “Identification of program modifications and its applications in software maintenance,” pp. 282-290 in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, FL, Nov, 1992), IEEE Computer Society, Washington, DC (1992).

24. Leung, H.K.N. and White, L., “A cost model to compare regression test strategies,” pp. 201-208 in *Proceedings of the IEEE Conference on Software Maintenance,* (Sorrento, Italy, Oct, 1991), IEEE Computer Society, Washington, DC (1991).

25. Lyle, J.R., Wallace, D.R., Graham, J.R., Gallagher, K.B., Poole, J.E., and Binkley, D.W., “A CASE tool to evaluate functional diversity in high integrity software,” *U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD,* (1995).

26. Marick, B., “A Survey of test effectiveness and cost studies,” Report No. UIUCDCS-R-90-1652, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL (December 1990).

27. Marick, B., “Three ways to improve your testing,” in *Unpublished Report* (Testing Foundations, Champaign IL), (1993).

28. Notkin, D., Ellison, R.J., Staudt, B.J., Kaiser, G.E., Kant, E., Habermann, A.N., Ambriola, V., and Montangero, C., Special issue on the GANDALF project, *Journal of Systems and Software* **5**(2)(May 1985).

29. Ostrand, T.J. and Weyuker, E.J., “Using data flow analysis for regression testing,” in *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference,* (Portland, Oregon, September 19-20, 1988), ACM, New York, NY (1988).

30. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984)*, ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

31. Podgurski, A. and Clarke, L.A., “A formal model of program dependences and its implications for software testing, debugging, and maintenance,” *IEEE Transactions on Software Engineering* **16**(9)(September 1990).

32. Rapps, S. and Weyuker, E.J., “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering* **SE-11**(4) pp. 367-375 (1985).

33. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A system for constructing language-based editors,* Springer-Verlag, New York, NY (1988).

34. Rothermel, G. and Harrold, M.J., “A safe, efficient algorithm for regression test selection,” *Proceedings of the IEEE Conference on Software Maintenance,* (Montreal, Quebec), pp. 358-367 IEEE Computer Society, (September 1993).

35. Rothermel, G. and Harrold, M.J., “Selecting tests and identifying test coverage requirements for modified software,” *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* Seattle,

Washington), pp. 169-84 (August 1994).

36. Rothermel, G. and Harrold, M.J., “Analyzing regression test selection techniques,” *IEEE Transactions on Software Engineering* **22**(8)IEEE Computer Society, (1996).

37. Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* The M.I.T. Press, Cambridge, MA (1977).

38. Weiser, M., “Program slicing,” pp. 439-449 in *Proceedings of the Fifth International Conference on Software Engineering,* (San Diego, CA, March 1981), (1981).

39. Weyuker, J., “The complexity of data flow criteria for test data selection,” *Information Processing Letters* **19** pp. 103-109 (1984).

40. White, L., “Software Testing and Verification,” pp. 335-390 in *Advances in Computers,*Vol 26., Academic Press (1987).

41. Wong, W., Horgan, J., London, S., and Mathur, A., “Effect of test set minimization of faults detection effectiveness,” pp. 41-50 in *Proceedings of the 17th International Conference on Software Engineering,* (Seattle WA, October 1995), (April 1995).

42. Yang, W., Horwitz, S., and Reps, T., “A program integration algorithm that accommodates semantics-preserving transformations,” *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments,* (Irvine, CA, December 3-5, 1990)*, ACM SIGSOFT Software Engineering Notes,* (1990).

43. Yang, W., “Identifying syntactic differences between two programs,” *Software—Practice and Experience* **21**(7)(July 1991).