Using Semantic Differencing to Reduce the Cost of Regression Testing

David Binkley

Loyola College in Maryland†

Abstract

This paper presents an algorithm that reduces the cost of regression testing by reducing the number of test cases that must be re-run and by reducing the size of the program that they must be run on. The algorithm uses dependence graphs and program slicing to partition the components of the new program into two sets: preserved points—components that have unchanged run-time behavior; and affected points—components that have changed run-time behavior. Only test cases that test the behavior of affected points must be re-run; the behavior of the preserved points is guaranteed to be the same in the old and new versions of the program. Furthermore, the algorithm produces a program `differences`, *which captures the behavior of (only) the affected points. Thus, rather than re-testing the (large) new program on a large number of test cases, it is possible to certify the new program by running the (smaller) program* `differences` *on a (smaller) number of test cases.*

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*

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 the smallest of changes (the top three entries of Weinberg's list of the largest software disasters are one line changes that were not tested [Weinberg83]). Although the effort required to make a (small) change may be minimal, the effort required to recertify the program after such a change may be substantial. This paper presents a method for reducing the cost of recertifying a program by both reducing the number of test cases that must be re-run and by reducing the complexity of the program that they must be run on.

Given a program `certified`, which passes a test suite, and a modified version of this program, `modified`, the algorithm described in this paper produces a third program `differences`, which captures all the changes in `modified` when compared to `certified`. Existing algorithms that compare programs include test-based algorithms, such as the UNIX[1] utility *diff*, which compare programs as strings of text. This approach is very general (*e.g.*, it can be applied to programs, documents, data files, and other text objects). However, it is clearly inadequate for determining which parts of a program must be retested. For example, consider the following changes for which determining textual differences alone is insufficient:

(1) A textual change that does not produce a change in behavior. (In this case *no* re-testing is necessary.)

(2) A textual change in one location of a program that causes a change in behavior "down stream" in either the same procedure, a called procedure, or a calling procedure.

In contrast, the algorithm described in this paper uses language semantics to identify program components that exhibit different run-time *behavior* (the behavior of a program is formalized in Section 4). This algorithm partitions the components of `modified` into *preserved points*—program components that compute the same values in `modified` and `certified`—and *affected points*—program components that compute different values in `modified` and `certified`. One result of this work is that only the affected points must be re-tested;

[1] Unix is a trademark of AT&T Bell Laboratories.

† This work was begun while the author was at the University of Wisconsin—Madison.

Author's current address: Comp. Sci. Dept., Loyola College in Maryland, 4501 N. Charles St., Baltimore, MD 21210 (410) 617-2881.

the behavior of the preserved points is guaranteed to be the same in `modified` and `certified`. In addition, the algorithm produces a program `differences`, which captures the computation of the affected points. Thus, rather than re-testing the (large) program `modified` on 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.

Previous algorithms for determining semantic differences (for example [Horwitz89]) only apply to limited programming languages. The chief difficulty in determining the semantic differences between programs written in "real" programming languages is extending these techniques to handle procedures and procedure calls. The algorithm described in this paper operates on programs that include procedures and procedure calls along with the following language features: simple scalar non-global variables, assignment statements, conditional statements, while-loops, and output statements. To support more completed data structures (such as arrays, records, and pointers) and a richer set of control structures (such as for-loops) requires changes to only the first step of the algorithm. This step produces a dependence graph representation for `modified` and `certified` called a *system dependence graph* (SDG). The algorithm for computing semantic differences is based solely on program slices [Weiser81, Horwitz90] of these SDGs; it is unaffected by additional language features.

The remainder of this paper is organized as follows: Section 2, discusses related work. Section 3, presents background material on SDGs and the interprocedural slicing algorithm discussed in [Horwitz90]. Section 4 presents the main technical contribution of this paper: it defines "semantic difference" in the presence of procedures and procedure calls and describes an algorithm for computing the semantic differences between `certified` and `modified`. Conclusions and future fork are presented in Section 5.

2. Related Work

Recent work on reducing the cost of regression testing has concentrated on reducing the number of test cases that must be re-run [Benedusi88, Fischer81, Harrold88]. These approaches are all based on data-flow analysis of control-flow graphs. The goal of this analysis is to determine which *paths* in the control-flow graph are affected by a change. Unfortunately, the use of control-flow-graph paths limits the effectiveness of these techniques in two ways. First (roughly put) 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 (new values for of *x* and *y*). Since the need to re-test the computation of *x* or *y* includes this path, it forces the need to re-test both.

A second drawback of existing approaches is their computational complexity. (It is unclear if this is an inherent problem with control-flow graph based techniques, or just present in existing algorithms.) For example, the approach in [Benedusi88], identifies unchanged, changed, and new "path expressions". Unfortunately, in its canonical form, the length of a path expression may be exponential in the size of the program.

In contrast the technique described in this paper is more precise because dependence graphs "throw away" unnecessary sequencing information contained in a control-flow graph. For example, it is capable of identifying the two distinct computations in procedure *A* of Figure 1. This is observed in [Benedusi88 where the authors state " 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 $\cdots$ and will thus have to be re-tested. It may, however, be the case that only a small subset of these paths actually use the initialized variable."

Furthermore, the technique presented in this paper, requires space and time linear in the size of the programs being analyzed. (This assumes the program's dependence graphs are stored with the program, as is done in a programming environment [Ottenstein84].) Finally, the approach used in this paper addresses not only the reduction of the number of tests, but also a reduction in the size of the program on which these tests must be run.

The semantic differencing technique described in this paper is based on the work in [Binkley91]. An overview of related semantics based algorithms, which use dependence graphs appears in [Horwitz92]. These algorithms include semantics bases 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 [Binkley91, Horwitz89a]. The changed behavior of *A* with respect to *Base* is captured by the term $\Delta(A, Base)$, which is used to compute `differences`.

In [Gallagher91], Gallagher defines a *decomposition slice*, which is similar to Δ. In his model, a decomposition slice of `certified` is taken with respect to all the output statements for a particular variable *before* any updates are made. Subsequent modifications are restricted to certain statements in this slice. Because statements outside a decomposition slice cannot be affected by modifications inside the slice, only the functionality of the statements in the decomposition slice must be re-tested. A decomposition slice with respect to a set of output statements can be computed using Δ by treating the output statements as strongly affected points (see Section 4).

3. The System Dependence Graph and Interprocedural Slicing

This section briefly summarizes the definition of the system dependence graph (SDG) and the interprocedural slicing algorithm described in [Horwitz90].

3.1. The System Dependence Graph

The SDG extends previous dependence representations to accommodate collections of procedures with procedure calls rather than just monolithic programs. SDGs model a flat ("C" like) language with the following properties:

(1) A complete *system* consists of a single (main) procedure and a collection of auxiliary procedures.

(2) Auxiliary procedures end with **return** statements; the main procedure ends with an **end** statement, which lists the variables that have values in the final state.

(3) Parameters are passed by value-result.[2]

We assume systems contain no call sites of the form $P(x, x)$ or $P(g)$, where g is a global variable. The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, global variables are not discussed explicitly.

Example. Figure 1 shows an example system (and part of its SDG).

An SDG is made up of a collection of procedure dependence graphs (PDGs) connected by interprocedural control- and flow-dependence edges. PDGs are similar to the program dependence graphs used for representing programs in vectorizing and parallelizing compilers [Kuck81, Ferrante87] except that they include vertices and edges representing call statements, parameter passing, and transitive data dependences due to calls.

The vertices of each procedure's PDG represent the predicates, assignment statements, call statements and parameter passing (described below) in the system; in addition, each PDG has a distinguished vertex called the *entry vertex* and the main procedure's PDG contains an *initial-definition vertex* for each variable that may be used before being defined and a *final-use vertex* for each variable named in the program's end statement. (Programs are viewed as state transformers from an initial state, which must define those variables that have initial-definition vertices, to a final state, where only those variables listed in the end statement are defined.)

[2] Other parameter passing mechanisms are also supported by the algorithm discussed in this paper. For example, parameters passed by "value" or by "result" are simply special cases of value-result. The changes necessary to handle reference parameters introduces the need to handle aliasing. See [Horwitz90] for a further discussion of how to handle reference parameters and aliasing.

The vertices representing parameters model value-result parameter passing where initial actual parameter values are copied to formal parameters, the called procedure is invoked, and, finally, result formal parameter values are copied back to actual parameters. This is represented in the SDG by four kinds of *parameter* passing vertices: *actual-in* and *actual-out* vertices at each call site and *formal-in* and *formal-out* vertices in each procedure. Actual-in and formal-in vertices are included for every global variable that may be used or modified as a result of the call and for every parameter; formal-out and actual-out vertices are included only for global variables and parameters that may be modified as a result of the call. (Interprocedural data-flow analysis is required to determine what parameter vertices should be included for each procedure [Banning79, Barth78]).

Within each PDG there are four kinds of edges: *control dependence edges*, which in essence represent the nesting structure of the program, *flow dependence edges*, which represent def-use chains, *summary edges*, which represent transitive dependences due to calls, and *meeting-point edges*, which represent the common use of two actual parameters in a called procedure. All but meeting-point edges are described in [Horwitz90]; meeting-point edges are described in [Binkley91]. PDGs are connected to form an SDG using three kinds of edges: (1) a *call* edge from each call-site vertex to the corresponding procedure-entry vertex; (2) a *parameter-in* edge from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

3.2. Interprocedural Slicing

The semantic differencing algorithm uses *backward* and *forward* interprocedural slices. A backwards slice with respect to vertex v in SDG G is a subgraph of G that contains those program components that potentially affect v.

Interprocedural slicing can be defined as a reachability problem using the SDG. The slices obtained using this approach are the same as those obtained using Weiser's interprocedural-slicing method [Weiser84]. However, this approach produces an imprecise slice because it considers paths in the graph that are not possible execution paths. For example, there is a path in the graph shown in Figure 1 from the vertex of procedure A labeled "$a_{in} := x$" to the vertex of A labeled "$y := a_{out}$." However, this path corresponds to procedure *Add* being called from the first call site in A, but returning to the second call site in A, which is not possible. The value of y after the second call is independent of the value of x before the first call, and so the vertex labeled "$a_{in} := x$" should not be included in the backward slice with respect to the vertex labeled "$y := a_{out}$."

```
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
end(sum)
```

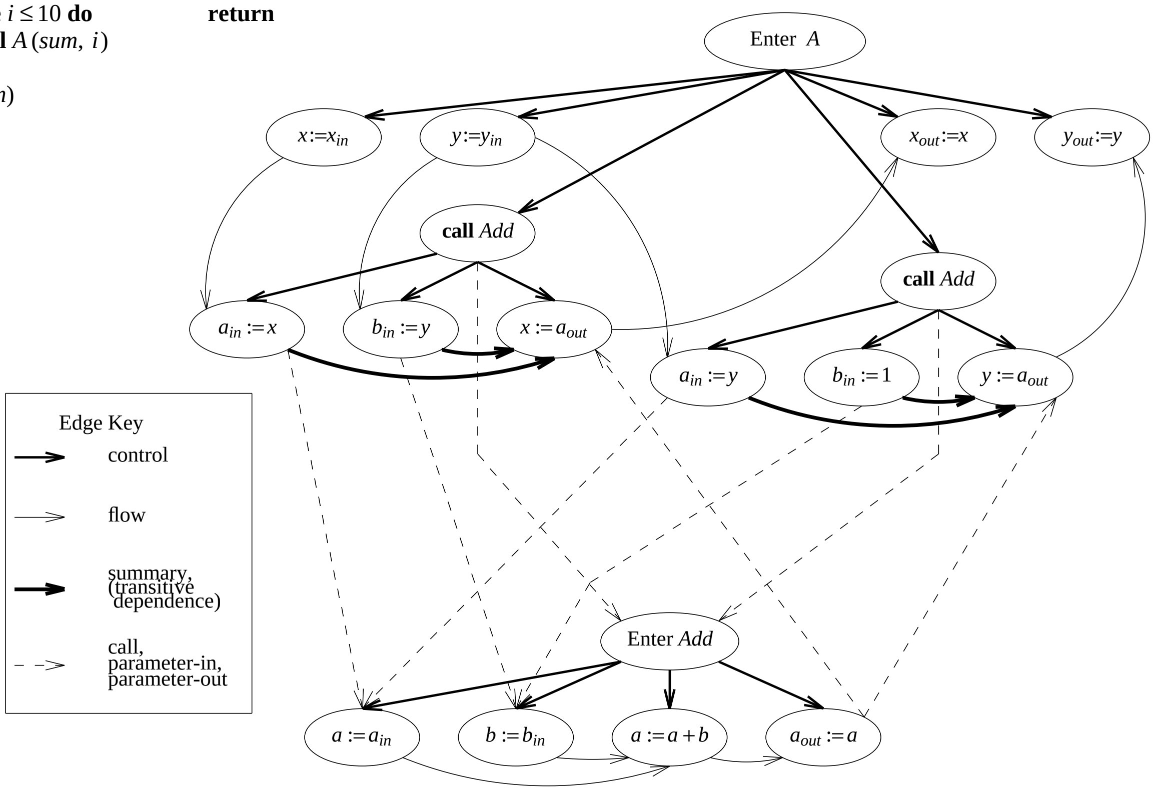

Figure 1. An example system, which sums the numbers from 1 to 10, and part of its SDG (only the PDGs for *A* and *Add* are shown).

To achieve greater precision, the slice of SDG G with respect to a set of vertices S is computed using two passes over G. During each pass summary edges are used to permit "moving across" a call site without having to descend into the called procedure; thus, there is no need to keep track of calling context to ensure that only legal execution paths are traversed. Pass 1 starts from all vertices in S and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, meeting-point edges, and parameter-in edges, but *not* along parameter-out edges. Pass 2 starts from all vertices reached in Pass 1 and goes backwards along flow edges, control edges, summary edges, meeting-point edges, and parameter-out edges, but *not* along call or parameter-in edges. The result of an interprocedural backward slice consists of the sets of vertices encountered during by Pass 1 and Pass 2, and the set of edges induced by this vertex set. An algorithm for finding the vertices in each pass of an interprocedural backward slice is stated in Figure 2.

Example. Figure 3 shows an interprocedural backward slice of the SDG shown in Figure 1.

The remainder of this paper uses the operators *b1* and *b2* to designate the individual passes of a backwards slice. In the terminology of Figure 2, operators *b1* and *b2* are defined as follows:

$$b1(G, S) \triangleq \text{ReachingVertices}(G, S, \{\, parameter-out \,\})$$
$$b2(G, S) \triangleq \text{ReachingVertices}(G, S, \{\, parameter-in, call \,\})$$

The vertices of the (full) backward slice, denoted by $b(G, S)$, are computed by composing *b1* and *b2*:

$$b(G, S) \triangleq b2(G, b1(G, S)).$$

Finally, the backward slice (including edges) of graph G with respect to vertex set S is denoted by $Induce(G, b(G, S))$, where $Induce : SDG \times vertex\text{-}set \rightarrow SDG$ is the function that returns the subgraph of the SDG induced by the vertex-set.

```
function ReachingVertices($G$, $V$, $EdgeKinds$) returns a set of vertices
declare
  $G$: an SDG
  $V$, $WorkList$, $Answer$: sets of vertices of $G$
  $EdgeKinds$: a set of edge-kinds
  $v$, $w$: vertices of $G$
begin
  $WorkList := V$
  $Answer := \emptyset$
  while $WorkList \neq \emptyset$ do
    Select and remove a vertex $v$ from $WorkList$
    Mark $v$
    Insert $v$ into $Answer$
    for each unmarked vertex $w$ such that there is an edge $w \rightarrow v$ whose kind is not in $EdgeKinds$ do
      Insert $w$ into $WorkList$
    od
  od
  return($Answer$)
end
```

Figure 2. Function ReachingVertices returns all vertices in G from which there is a path to a vertex in V along edges whose edge-kind is something other than those in the set *EdgeKinds*.

```
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
```

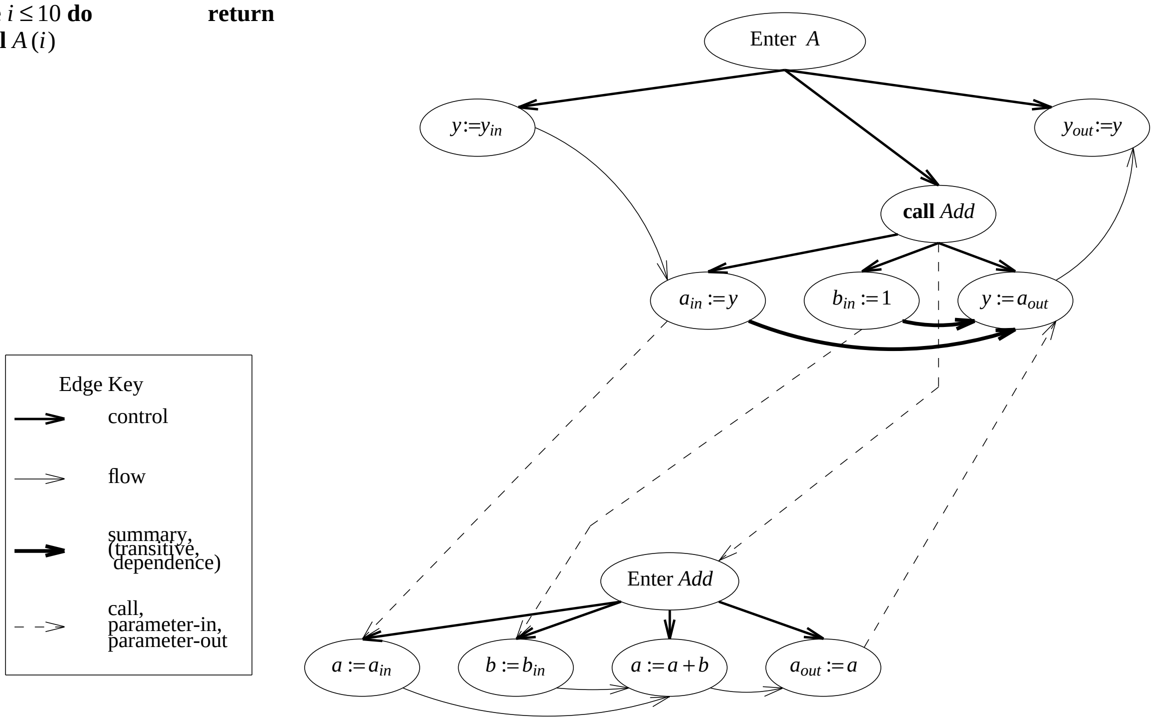

Figure 3. (Part of) the slice of the SDG in Figure 1 taken with respect to the actual-out vertex labeled “$y_{out} := y$” and the program to which is corresponds. This program contains the looping control only from the program in Figure 1.

Forward Slicing

The operation of forward slicing is the dual of the backward slicing operation: whereas a backward slice discovers those program components that potentially affect a given component, a forward slice discovers those components that are potentially affected by a given component. As with backward slicing, an interprocedural forward slice can be computed using two passes. Each pass traverses only certain kinds of edges; however, in a forward slice edges are traversed from source to target. The first pass, an *f1* slice, ignores parameter-in and call edges; the second pass, an *f2* slice, ignores parameter-out edges. Thus, the full forward slice of *G* with respect to *S*, denoted by $Induce(G, f(G, S))$, is defined as $Induce(G, f2(G, f1(G, S)))$. (In the remainder of this paper *Induce* is omitted when there is no ambiguity.)

4. Computing Semantic Differences

This section contains the four technical contributions of this paper:

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

(2) An algorithm for computing $AP(modified, certified)$—(a safe approximation[3] to) the set of *affected points*, those components of `modified` that exhibit different behavior in `modified` and `certified`.

(3) An algorithm for computing $\Delta(modified, certified)$—(a safe approximation to) 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` from $\Delta(modified, certified)$. This program captures the semantic differences between `modified` and `certified`. An important property of `differences` is that it can be proven to capture the semantic differences between `modified` and `certified`. This proof is a slight modification of the correctness proof given in [Binkley91].

This Section concludes with an example illustrating how `differences` is used in reducing the cost of regression testing.

[3] Since determining any non-trivial property of a program is undecidable, any algorithm for identifying semantic differences must be approximate. The algorithm discussed in the paper is safe: it correctly identifies all semantically changed components of the program but may 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`.

4.1. Defining Semantic Difference

Before defining "semantic difference" it is necessary to have a language semantics and to identify a *correspondence* between the components of `certified` and `modified`. (The components of a system are the parts of the system represented by vertices in the system's SDG.) In this paper, we assume the intuitive standard operational semantics. A formal semantics in given in [Binkley91], along with a proof that *roll-out* (introduced below) is a semantics-preserving transformation. This proof is important because it allows the semantics of two systems to be related by the semantics of their roll-outs.

A correspondence between the components of `certified` and `modified` can be obtained using a syntactic matching algorithm, such as that of [Yang92], or a special editor that maintains statement tags.

The semantics of a program could be defined as the "sequence of values" produced by each component of the program. (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 a predicate, the sequence of boolean values to which the predicate evaluates; and for a variable named in the end statement, the singleton sequence containing the variable's final value.) While this is sufficient for programs without procedures, it is insufficient in the presence of procedures because components must produce different sequences of values in different calling contexts.

A more refined definition of program semantics, which correctly 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. Because rolled-out systems have no procedure calls, the semantic differences between two rolled-out systems can be defined in terms of sequences of values. This, in turn, defines the differences between the two systems because *roll-out* is a semantics preserving transformation [Binkley91]. It should be emphasized that no roll-outs, which may produce infinite programs, are actually performed; rather, the roll-out concept is used only as a conceptual device to help properly formulate the semantic differences between two systems.

The set of *affected points*, defined using roll-out, contains those components of `modified` that exhibit a semantic difference when compared to `certified`. To correctly account for calling context when computing $\Delta(modified, certified)$, it is necessary to partition the set of affected points 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 procedures called by *P*. Definitions for these three sets are given below.

DEFINITION (Affected Points). Component *c* of `modified` is an *affected point* if one of its occurrences has no corresponding occurrence in *roll-out*(*certified*), or if it has corresponding occurrences in *roll-out*(*modified*) and *roll-out*(*certified*) that compute different sequences of values when both programs are evaluated on the same initial state.

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

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

4.2. 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). First, a safe approximation to 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*(*modified, certified*)).

$$DAP(modified,\ certified) \triangleq \{\, v \in V(G_{\texttt{modified}}) \mid v \notin V(G_{\texttt{certified}}) \vee v \text{ has different incoming flow, control, or def-order edges in } G_{\texttt{modified}} \text{ and } G_{\texttt{certified}} \,\}.$$

(Def-order edges are additional flow dependence edges, which run from from one definition to another when both definitions define the same variable and reach a common use. For the purposes of defining incoming edges, however, a def-order edge is viewed as a hyper-edge from the two definitions to the common use; thus, it is an incoming edge of the use. Discussions of def-order edges and why interprocedural edges are ignored in the definition of DAP appear in [Binkley91].)

The set *AP*(*modified*, *certified*), which contains all the affected points, is defined using *DAP*(*modified, certified*) as follows:

DEFINITION (*AP*(*modified, certified*)).

$$AP(modified,\ certified) \triangleq f(G_{\texttt{modified}},\ DAP(modified,\ certified)).$$

A safe approximation to the strongly affected points is contained in the set *SAP*(*modified*, *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*(*modified*, *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* considers the parameter-out edges to call sites in *P*, but ignores the parameter-in and call edges to procedures called by *P*; thus, we define *SAP*(*modified*, *certified*) using an *f1* slice:

DEFINITION (*SAP*(*modified*, *certified*)).

$$SAP(modified,\ certified) \triangleq f1(G_{\texttt{modified}},\ DAP(modified,\ certified)).$$

Finally, *WAP*(*modified*, *certified*), which contains all the weakly affected points, is the set of affected points that are not strongly affected:

DEFINITION (*WAP*(*modified*, *certified*)).

$$WAP(modified,\ certified) \triangleq AP(modified,\ certified) - SAP(modified,\ certified).$$

4.3. Constructing Δ(*modified, certified*)

The operator Δ applied to $G_{\texttt{modified}}$ and $G_{\texttt{certified}}$ produces a subgraph of $G_{\texttt{modified}}$ that contains all the components necessary to capture the behavior of the components in *AP*(*modified*, *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, for a weakly affected point, *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 Δ(*modified, certified*):

$$\begin{aligned}\Delta(\textit{modified, certified}) &\triangleq \\ &b(G_{\texttt{modified}}, \textit{SAP}(\textit{modified, certified})) \cup \\ &b2(G_{\texttt{modified}}, \textit{WAP}(\textit{modified, certified})).\end{aligned}$$

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.

4.4. Computing `differences` from Δ

To produce the system `differences` from the SDG Δ(*modified*, *certified*) may require first making Δ(*modified*, *certified*) *feasible*. A feasible SDG is one that corresponds to some system. After Δ(*modified*, *certified*) has been augmented to make it feasible, it is reconstituted into a system.

Δ(*modified*, *certified*) is infeasible if it contains a parameter mismatch.[4] Mismatches involve two call sites on the same procedure that, informally, contain different parameters. Formally, there are two kinds of mismatches: an *actual-in mismatch* and an *actual-out mismatch*. An actual-in (actual-out) mismatch exists if one of the call sites has an actual-in (actual-out) vertex for formal-parameter *x* and the other does not. Δ(*modified*, *certified*) is *augmented* to remove actual-in and then actual-out mismatches before the program `differences` is produced.

If an actual-in vertex *v* must be added to Δ(*modified*, *certified*), simply adding *v* alone is insufficient because an appropriate initial parameter value must be computed for the corresponding actual parameter. In order to include the program components that compute this value, the slice $b2(G_{\texttt{modified}}, v)$ is added to Δ(*modified*, *certified*). Additions are continued until no actual-in mismatches exist. Actual-out mismatches are removed by simply adding missing actual-out vertices; because these vertices represent dead-code no additional slicing is necessary.

Once Δ(*modified*, *certified*) is feasible, it is reconstituted into a program that has Δ(*modified*, *certified*) as its SDG. This is accomplished by projecting the statements of `modified` that are represented in Δ(*modified*, *certified*) into the program `differences`. In other words, the statements of `differences` are the statements of `modified` represented by vertices in Δ(*modified*, *certified*) and these statements appear in `modified` in the same order and at the same nesting level as in `modified`.

[4] It is possible to produce a system from Δ(*modified*, *certified*) in this case. The resulting program may have missing parameters; however, it does capture the differences between `modified` and `certified` provided programs are evaluated using a lazy or demand semantics [Stoy77]. In such an evaluation, the values for missing parameters are never “demanded.”

4.5. Reducing the Cost of Regression Testing

Due to space limitation, this section presents the use of *differences* in reducing the cost of regression testing by way of an example. Consider the program `modified` shown in Figure 4. The set *SAP*(*modified*, *certified*) contains the vertices representing the statements in boxes (these are also the directly affected points); the set *WAP*(*modified*, *certified*) contains the vertices representing procedure *Add*. Figure 5 shows the program `differences` computed from these points using the algorithm developed in Sections 4.2-4.4.

As stated in Section 1, `differences` is smaller and therefore more efficient than `modified`. For example, the program `differences` shown in Figure 5 is smaller and more efficient than the program `modified` shown in Figure 4 because it does not contain the computation of the variable *sum*. Section 1 also states that it may be unnecessary to run `modified` on all the test cases used to test `certified`. For example, assume `certified` from Figure 1 was tested using two test cases: one testing the computation of *i* and the other testing the computation of *sum*. Because the statements that assign to *i* or *sum* are preserved points (*i.e.*, not affected points), these two test cases need not be re-run; `modified` is *guaranteed* to pass them. In addition, even if a test case must be re-run, it can be re-run using `differences` in place of `modified`; this reduces the cost of running the test because `differences` is a smaller, more efficient, program.

5. Conclusions and Future Work

Knowing the semantic differences between two programs is clearly useful in many program maintenance activities. Not the least of which is reducing the cost of regression testing. The technique described in this paper reduces this cost in two ways: it reduces the number of tests that must be re-run, and it reduces the complexity of the program on which they must be run. This is accomplished by using the system dependence graph in place of the control-flow graph, which avoids the unnecessary relation between components on the same path of a control-flow graph.

Future work includes a better formalization of how test cases are selected once `differences` has been computed. The basic idea is to use input statements in Δ(*modified*, *certified*) in an analysis similar to that in [Benedusi88]. After identifying the tests that must be re-run, it may be possible to reduce the complexity of the test cases using techniques analogous to program slicing to “slice” the test inputs. This is especially true if the test cases are generated by another program; in this case, a slice of the generating program taken with respect to the output statements that match up with input statements in Δ(*modified*, *certified*) produces a generating program that produces only test inputs required by the input statements in `differences`.

```
procedure Main
    $sum := 0$
    $\boxed{prod := 1}$
    $i := 1$
    while $i \le 10$ do
        $\boxed{\textbf{call } Product(prod, i)}$
        call $A(sum, i)$
    od
end($sum$, $\boxed{prod}$)

procedure $A(x, y)$
    call $Add(x, y)$
    call $Add(y, 1)$
return

$\boxed{\textbf{procedure } Product(a, b)}$
    $\boxed{count := 1}$
    $\boxed{temp := 0}$
    $\boxed{\textbf{while } count \le a \textbf{ do}}$
        $\boxed{\textbf{call } Add(temp, b)}$
        $\boxed{\textbf{call } Add(count, 1)}$
    $\boxed{\textbf{od}}$
    $\boxed{a := temp}$
$\boxed{\textbf{return}}$

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

Figure 4. The system `modified` was obtained from the system `certified` shown in Figure 1. (Boxes indicate the changes made to `certified`.)

```
procedure Main

    $prod := 1$
    $i := 1$
    while $i \le 10$ do
        call $Product(prod, i)$
        call $A(i)$
    od
end($prod$)

procedure $A(y)$

    call $Add(y, 1)$
return

procedure $Product(a, b)$
    $count := 1$
    $temp := 0$
    while $count \le a$ do
        call $Add(temp, b)$
        call $Add(count, 1)$
    od
    $a := temp$
return

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

Figure 5. The system `differences` obtained using the techniques described in this paper when applied to `modified` from Figure 4 and `certified` from Figure 1.

Future work on computing `differences` deals primarily with computing slices. For example, the addition of *declaration dependences* to the SDG would allow reduced type definitions—declarations that included only selected fields from a record—to be included in `differences`. Information on declarations would also be useful in reducing the size of individual test inputs.

Finally, it would be beneficial to have a formal comparison between the cost of (a) using the method presented in this paper, (b) using previous method for reducing the cost of regression testing, and (c) using the simple re-run all tests strategy. Such a comparison could be carried out using the techniques presented in [Leung91], which produces a model for comparing the cost of the selective regression testing strategies with the traditional retest-all strategy.

Acknowledgments

Discussions with Keith Gallagher improved this paper dramatically. The author also wishes to thank the reviewers (especially reviewer 33) for their helpful suggestions.

REFERENCES

Banning79.
Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).

Barth78.
Barth, J.M., "A practical interprocedural data flow analysis algorithm," *Commun. of the ACM* **21**(9) pp. 724-736 (September 1978).

Benedusi88.
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).

Binkley91.
Binkley, D., "Multi-procedure program integration," Ph.D. dissertation and Tech. Rep. TR-1038, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

Ferrante87.
Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

Fischer81.
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).

Gallagher91.
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).

Harrold88.
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).

Horwitz89.
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).

Horwitz89a.
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).

Horwitz90.
Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

Horwitz92.
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).

Kuck81.
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., “Dependence graphs and compiler optimizations,” pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

Leung91.
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).

Ottenstein84.
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).

Stoy77.
Stoy, J.E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* The M.I.T. Press, Cambridge, MA (1977).

Weinberg83.
Weinberg, G., “Kill that code!,” *Infosystems* **48-49**(August 1983).

Weiser81.
Weiser, M., “Program Slicing,” pp. 439-449 in *Proceedings of the Fifth International Conference on Software Engineering,* (San Diego, CA, March 1981), (1981).

Weiser84.
Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

Yang92.
Yang, W., “Identifying syntactic differences between two programs,” to be published in *Software—Practice and Experience* (1992).