Incremental Program Testing Using Program Dependence Graphs

SAMUEL BATES
samuel@cs.wisc.edu

SUSAN HORWITZ
horwitz@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 West Dayton Street
Madison, Wisconsin 53706

ABSTRACT

Program dependence graphs have been proposed for use in optimizing, vectorizing, and parallelizing compilers, and for program integration. This paper proposes their use as the basis for *incremental program testing* when using *test data adequacy criteria*. Test data adequacy is commonly used to provide some confidence that a particular test suite does a reasonable job of testing a program. Incremental program testing using test data adequacy criteria addresses the problem of testing a modified program given an adequate test suite for the original program. Ideally, one would like to create an adequate test suite for the modified program that reuses as many files from the old test suite as possible. Furthermore, one would like to know, for every file that is in both the old and the new test suites, whether the program components exercised by that file have been affected by the program modification; if no components have been affected, then it is not necessary to rerun the program using that file.

In this paper we define adequacy criteria based on the program dependence graph, and propose techniques based on program slicing to identify components of the modified program that can be tested using files from the old test suite, and components that have been affected by the modification. This information can be used to reduce the time required to create new test files, and to avoid unproductive retesting of unaffected components. Although exact identification of the components listed above is, in general, undecidable, we demonstrate that our techniques provide safe approximations.

This work was supported in part by NSF under grant CCR-8958530, by DARPA, monitored by ONR under contract N00014-88-K-0590, as well as by grants from 3M and Xerox.

ACM-20th PoPL-1/93-S.C., USA

1. INTRODUCTION

Program dependence graphs have previously been proposed for use in optimizing, vectorizing, and parallelizing compilers, [Fer87] and for program integration [Hor89b]. This paper proposes a new use: as the basis for *incremental program testing* when using *test data adequacy criteria*.

Test data adequacy is a widely accepted concept for providing some confidence that a particular test suite (set of input files) does a reasonable job of testing a program [Rap85,How76,Mar90]. For example, a test suite is adequate under the *all-statements* criterion if for every statement in the program there is at least one input file in the suite such that when the program is run using that input file, the statement is executed.

Incremental program testing using test data adequacy criteria addresses the following problem: given a program and a test suite that is adequate under criterion *C*, what should be done after the program is modified? Ideally, one would like to create a test suite for the modified program that is adequate under criterion *C*, and that reuses as many files from the old test suite as possible. Furthermore, one would like to know, for every file that is in both the old and the new test suites, whether the program components exercised by that file have been affected by the program modification; if no components have been affected, then it is not necessary to rerun the program using that file.

Example. Figure 1 shows a program before and after a modification. Version 1 is intended to compute the product of the odd numbers in the range 1..*N* (where *N* is an input that is supposed to be positive); however, the variable *oddP* is incorrectly initialized to zero. This error is corrected in Version 2, which also adds the computation of the product of the even numbers. The statements of Version 2 that are affected by these modifications (*i.e.*, the statements where new values are computed or used) are shown enclosed in boxes.

There are many statement adequate test suites for Version 1. One such suite consists of two files, containing the values 0 and 2, respectively; this test suite is also statement adequate for Version 2. Since the statements of Version 2 that are exercised by the first test file (the statements at lines [1], [2], and [3]) are not affected by the changes to the program, it is not necessary to run Version 2 using the first test file in order to test the modified program. □

In this paper we propose techniques to address the problems of incremental testing. In particular, we discuss how to identify

- which components of the modified program can be tested using files from the old test suite, and
- which components have been affected by the modification.

Given this information, retesting can proceed more quickly: the potentially time consuming task of creating new test files can be reduced or eliminated, and unproductive retesting of unaffected components can be avoided.

Although precise answers to the two questions listed above are, in general, undecidable, we propose the use of *program dependence graphs* and *program slicing* to provide safe approximations. This approach leads to more precise results than have been achieved by previous work in the area of incremental testing. [Ost88,Har,Tah89] In particular, previous work has suffered from two shortcomings:

(1) Previous techniques for matching components of the modified program with test files from the old test suite may not correctly handle multiple program modifications. In the presence of multiple modifications, those techniques may match a component of the modified program with an old test file that fails to test the component.

(2) By using a syntactic rather than a semantic definition of "affected component," previous techniques may fail to identify components that have been transitively affected by modifications elsewhere in the program.

The work presented here avoids these shortcomings.

The remainder of the paper is organized as follows: Section 2 provides background material, including the definitions of program dependence graphs, program slicing, and adequacy criteria based on program dependence graphs. Section 3 addresses the two issues listed above:

- Section 3.1 describes how to determine which components of the modified program can be tested using files from the original test suite (and which components might need new test files);
- Section 3.2 defines what it means for a program component to be affected by a modification and discusses how to identify affected components.

The techniques described in Section 3 assume that *all statements* is the criterion of interest; Section 4 discusses how to apply these techniques to other adequacy criteria, and applies them to the all-flow-edges criterion. Section 5 compares our approach to previous work; Section 6 summarizes our results.

Version 1

```
program main(f:file)
begin
    read(f, N)
    if N<1 then
        write("error: input<1")
    else
        i := 1
        oddP := 0
        while i ≤ N do
            if odd(i) then
                oddP := oddP*i
            fi
            i := i+1
        od
        write(oddP)
    fi
end
```

Version 2

```
program main(f:file)
begin
    read(f, N)
    if N<1 then
        write("error: input<1")
    else
        i := 1
        oddP := 1
        evenP := 1
        while i ≤ N do
            if odd(i) then
                oddP := oddP*i
            else
                evenP := evenP*i
            fi
            i := i+1
        od
        write(oddP)
        write(evenP)
    fi
end
```

Figure 1. Two versions of a program. Version 1 is supposed to compute the product of the odd numbers from 1 to *N*, but incorrectly initializes *oddP* to zero. Version 2 fixes that error and adds the computation of the product of the even numbers. The affected statements of Version 2 with respect to Version 1 are shown in boxes.

2. BACKGROUND

2.1. The Program Dependence Graph

To simplify our presentation, we restrict our attention to programs written in a language with the following characteristics[1]:

- A program consists of a sequence of one or more statements of the following kinds: assignment, read, write, if-then-else, or while-loop.
- Expressions include constants and scalar variables.
- A program reads from a single input file, which is named in the program header.

The program dependence graph (PDG) for a program *P* is a directed graph whose vertices are connected by several kinds of edges. The PDG includes a special *ENTRY* vertex, a vertex for every statement and predicate in *P*, a vertex that represents the initial opening of *P*'s input file, and, for every **read** statement, a vertex for each variable into which a value is read.

The edges of the PDG represent the program's *control* and *data* dependences. The source of a control edge can be a predicate vertex, the *ENTRY* vertex, or a vertex that represents a **read** statement; control edges are labeled either *true* or *false*. The presence of a control edge from vertex *v*

[1] Techniques have been proposed to permit program dependence graphs to be used to represent programs written in more realistic languages [Hor90,Hor89a,Lar88]. We use the simple language described above because it permits us to present our ideas on incremental program testing without having to discuss details that are independent of testing.

to vertex w means that the program component represented by v controls the execution of the component represented by w in the following sense: whenever the component represented by v is evaluated and its value matches the label on the edge, the component represented by w will eventually execute (barring the presence of an infinite loop or an exception such as division by zero); however, if the value does not match the label on the edge, the component represented by w may never execute (the *ENTRY* vertex and the **read** vertices are considered to represent pseudo-predicates that always evaluate to *true*).

Control dependences can be determined for programs with arbitrary control flow [Fer87]; however, for the language under consideration here, these dependences represent the nesting structure of the program:

- There is a control edge labeled *true* from the vertex that represents the predicate of an if-then-else statement to every vertex that represents a statement in the "then" branch of the if, and there is a control edge labeled *false* to every vertex that represents a statement in the "else" branch of the if.
- There is a control edge labeled *true* from the vertex that represents the predicate of a while-loop to every vertex that represents a statement inside the loop.
- There is a control edge labeled *true* from the vertex that represents the predicate of a while-loop to itself.
- There is a control edge labeled *true* from the *ENTRY* vertex to every vertex that represents a statement that is not inside any if-then-else or while-loop.
- There is a control edge labeled *true* from the vertex that represents a statement of the form "**read**($id_1, id_2, ..., id_n$)" to the vertices that represent $id_1, id_2, ..., id_n$.

We denote a control edge from vertex x to vertex y with label L by $x \longrightarrow_c^L y$.

There are three kinds of *data* dependences in a PDG: *flow dependence*, *def-order dependence*, and *read dependence*. Flow dependences are essentially *def-use* chains [Aho86]; that is, there is a flow edge from vertex v to vertex w iff all of the following hold:

(1) v defines a variable x;

(2) w uses x;

(3) there is an x-definition-free path in the program's control flow graph from v to w.

We denote a flow edge from vertex v to vertex w by $v \longrightarrow_f w$.

Def-order dependences are included in PDGs to ensure that semantically different programs cannot have identical PDGs[2]. There is a def-order edge from vertex v to vertex w with "witness" vertex u iff all of the following hold[3]:

(1) v and w define the same variable;

(2) there is a vertex u that is the target of flow edges from both v and w;

(3) there is a path in the program's control flow graph from v to w.

We denote a def-order edge from vertex v to vertex w with witness vertex u by $v \longrightarrow_{do(u)} w$.

Read dependences represent the sequential nature of input read from a file. There is a read edge from vertex v to vertex w iff either of the following holds:

(1) v is the vertex that represents the initial opening of the program's input file, w represents a **read** statement, and there is a read-free path in the program's control flow graph from v to w; or

(2) both v and w represent **read** statements, and there is a read-free path in the program's control flow graph from v to w.

We denote a read edge from vertex v to vertex w by $v \longrightarrow_r w$.

Example. The PDGs for the programs of Figure 1 are given in Figure 2(a).

2.2. Program Slicing

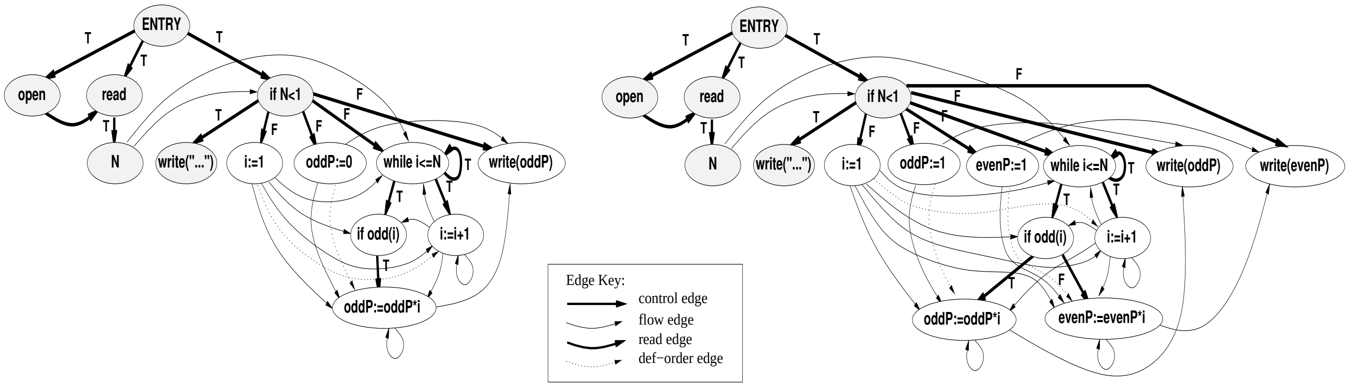
The goal of program slicing (defined by Mark Weiser [Wei84]) is to take a program P, a component c, and a set of variables V, and to produce a projection of P (by removing some of P's statements) such that both P and the projection compute the same values for all variables in V at component c when they are run on the same inputs.

A slicing algorithm that uses PDGs was given by Karl and Linda Ottenstein [Ott84]. This algorithm is slightly more restricted than the algorithm defined by Weiser; programs are sliced with respect to component c and the variables defined or used at c rather than an arbitrary set of variables. The Ottensteins' algorithm has two steps: Step 1 takes as input the program's PDG and the vertex v that represents component c. The output of Step 1 is the set of PDG vertices from which there is a path to v along control and/or flow edges. Step 2 creates the program projection by removing all components that do not correspond to a vertex identified by Step 1.

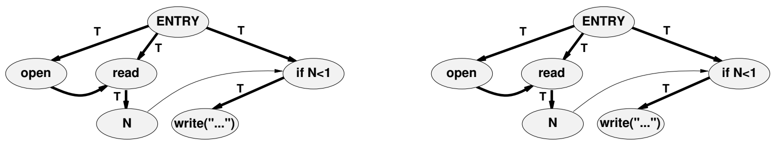
Tom Reps and Wuu Yang defined the PDG slice with respect to a vertex v to be the PDG induced by the vertex set identified in Step 1 of the Ottensteins' slicing algorithm. [Rep89] They showed that isomorphism of PDG slices guarantees equivalent behavior as defined below:

[2] Other definitions of program dependence graphs have included *anti* and *output* dependences in place of def-order dependences. [Fer87]

[3] This definition of def-order edges differs slightly from the one originally given [Hor88]; however, we have shown that the two definitions are equivalent. [Bat92b]



(a) The PDGs of the programs of Figure 1.



(b) The slices of the two PDGs with respect to **write**("...")

Figure 2. The PDGs of the two programs of Figure 1 and their PDG slices with respect to **write**("..."). The components that are in the slices have equivalent behaviors in the two programs.

DEFINITION (statement and predicate behavior). When a program is executed, the *behavior* of a statement or predicate in the program is the sequence of values that arise at that point during the execution. In particular: for an assignment statement, the sequence of values assigned to the left-hand-side variable; for a read statement, the sequence of values read from the input file; for a write statement, the sequence of values written to the output; for a predicate, the sequence of boolean values to which it evaluates.

The sequence of values that arise at statement or predicate s when program P is executed using input file f is denoted by $P(f)(s)$.

DEFINITION (equivalent behavior). Two statements or predicates, s_1 and s_2, in programs P_1 and P_2, respectively, have *equivalent behavior* iff all of the following hold:

- For any input file f on which both P_1 and P_2 terminate normally[4], $P_1(f)(s_1) = P_2(f)(s_2)$.
- For any input file f on which neither P_1 nor P_2 terminates normally, either $P_1(f)(s_1)$ is a prefix of $P_2(f)(s_2)$, or *vice versa*.
- For any input file f on which P_1 terminates normally but P_2 does not, $P_2(f)(s_2)$ is a prefix of $P_1(f)(s_1)$.
- For any input file f on which P_2 terminates normally but P_1 does not, $P_1(f)(s_1)$ is a prefix of $P_2(f)(s_2)$.

THEOREM 2.1 (PDG slice isomorphism guarantees equivalent behavior) [Rep89]. If s_1 and s_2 are statements or predicates (possibly in different programs) such that the PDG slices with respect to the vertices that correspond to s_1 and s_2 are isomorphic (where edge type and vertex and edge labels as well as graph "shape" must be preserved by the isomorphism), then s_1 and s_2 have equivalent behaviors.

Example. Figure 2(a) shows the PDGs of the two programs from Figure 1; shading is used to indicate the vertices that are identified by Step 1 of the Ottensteins' slicing algorithm when slicing with respect to **write**("error: input<1"). Figure 2(b) shows the PDG slices (the graphs induced by the shaded vertices in Figure 2(a)). The two PDG slices *are* isomorphic, and it is clear that the behaviors of the components **write**("error: input<1") are equivalent in the two programs. □

2.3. Defining Adequacy Criteria Using Program Dependence Graphs

As discussed in the Introduction, test data adequacy criteria provide a way to measure how thoroughly a program is tested by a given test suite. An adequacy criterion C is defined by specifying a set of program components, and by defining what it means for a component to be exercised when the program is run. A test suite is deemed adequate under criterion C if for every component c of the program,

[4] A program fails to terminate normally if it executes an infinite loop, or an exception such as division by zero.

there is a test t such that when the program is run on t, component c is exercised.

Some previous work has defined adequacy criteria in terms of a program's control flow graph [Hua75,Kra73]. For example, for the *all-statements* criterion, a component is a control flow graph node, and a node is exercised if the corresponding program statement or predicate is executed. Thus, test suite T meets the all-statements criterion if every statement and predicate in the program can be mapped to a test in T that causes it to execute. For the *all-branches* criterion, a component is a control flow graph edge whose source is a predicate node (*i.e.*, an edge labeled either *true* or *false*), and a component is exercised if the corresponding predicate is evaluated and the value matches the edge label. Thus, test suite T meets the all-branches criterion if every predicate in the program can be mapped to two (not necessarily different) tests in T: one that causes it to evaluate to *true*, and one that causes it to evaluate to *false*.

More recently, adequacy criteria have been defined in terms of the program's data flow (def-use chains), where uses are divided into two classes: *p-uses*, uses that occur in predicates, and *c-uses* (computation uses). [Las83,Nta84,Rap85] For example, for the *all-c-uses* criterion, a component is a def-use chain $d \rightarrow u$ where u is a computation use, and a component is exercised if u uses a value that was defined at d. Thus, test suite T meets the all-c-uses criterion if every def-to-c-use chain $d \rightarrow u$ in the program can be mapped to a test in T that causes u to use a value defined at d.

The program dependence graph, which includes representations of both control and data dependences, can be used to provide a single, unifying framework in which to define adequacy criteria; the vertices and edges of the PDG are the components used in the criteria definitions. [Bat92c] For example, for the *all-vertices* criterion, a component is a PDG vertex, and a component is exercised if the corresponding program statement or predicate is executed. For the *all-flow-edges* criterion, a component is a flow edge $v \longrightarrow_f w$, and a component is exercised if the program statement or predicate that corresponds to w uses a value defined at the program statement that corresponds to v.

Two adequacy criteria can be compared by considering the sets of test suites that are adequate under each criterion. [Rap85] A test suite that is adequate under one criterion C may also be adequate under another criterion D. If this is true for *all* C-adequate test suites, then we say that C subsumes D. Two criteria are equivalent if each subsumes the other. Subsumption can be used to create hierarchies of criteria. Figure 3 shows some examples of these hierarchies and also shows the relationships between criteria defined in terms of control flow graphs, data flow, and PDGs.

As illustrated in Figure 3, some of the PDG adequacy criteria are equivalent to criteria previously defined in terms of control or data flow (for example, the all-vertices and all-statements criteria are equivalent); however, there are some previous criteria (for example, the all-c-uses criterion) that have no obvious analog in the PDG. This means that the techniques described in this paper may not apply when criteria are used that have no PDG analog. However, we believe that the set of PDG criteria is rich enough to be of practical use in program testing, and that the advantages of being able to use the techniques proposed here will outweigh any disadvantages of using PDG criteria.

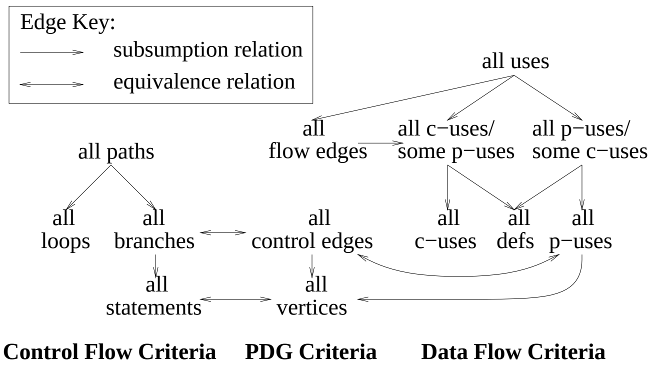


Figure 3. The relationships between PDG criteria and criteria defined in terms of a program's control or data flow.

3. TESTING A MODIFIED PROGRAM

We now turn to the problem of testing a program after it has been changed. In this section, we assume that the adequacy criterion of interest is all-vertices (equivalently, all-statements). Section 4 discusses how to extend our techniques to the PDG edge adequacy criteria.

Ideally, after modifying a program, one would like to create an adequate test suite for the modified program (reusing as many old tests as possible), and to run the program on all of the tests in this suite. However, it may be too expensive to run all the tests. Therefore, a reasonable goal is to run enough tests to guarantee that every *affected* statement of the modified program is exercised (by definition, a statement is affected either if it is a new statement or if its behavior differs from the corresponding statement of the original program[5]). In this case, if a reused test is only known to test *un*affected statements, that test should not be rerun.

Given the goal discussed above, the following process can be used to test a modified program:

[5]This definition of an affected statement requires being able to map a statement of the modified program to the corresponding statement of the original program. In practice, this map could be defined by applying a syntactic matching algorithm to the two programs, such as that of [Yan91], or by using a special editor that keeps track of statement correspondence as the program is modified. Such an editor can be created using systems such as MENTOR [Don84], GANDALF [Not85], and the Synthesizer Generator. [Rep88]

(1) A subset of the original test suite is identified for use in exercising statements of the modified program (this is discussed in Section 3.1).

(2) A subset of the tests selected in Step 1 is identified for use in exercising *affected* statements (this is discussed in Section 3.2).

(3) The programmer runs the modified program on the tests selected in Step 2; when the program is run, a record is kept of the statements that actually were exercised.

(4) The programmer creates new tests for the affected statements of the modified program that were not exercised by the reused tests (for example by using the techniques for test data generation proposed in [Cla76,DeM91,Kor90]).

3.1. Selecting Tests that Exercise Components of New

Let *Old* be a program and *New* be a variant created by modifying *Old*. Assume that T is a statement-adequate test suite for *Old*. For each test t of T, we assume that *Old* has been run on t, and that the statements exercised by t have been gathered in a set s_t; since T is statement-adequate, every statement of *Old* is in at least one of these sets. To identify the tests in T that will exercise the statements of *New*, we group the statements of *Old* and *New* into equivalence classes called *execution classes* such that all statements in the same class have *equivalent execution patterns*, defined as follows:

DEFINITION (equivalent execution patterns). If s_1 and s_2 are statements of P_1 and P_2, respectively, s_1 and s_2 have *equivalent execution patterns* iff all of the following hold:

(i) For any input file on which P_1 and P_2 both terminate normally, s_1 and s_2 are exercised the same number of times;

(ii) For any input file on which P_1 terminates normally but P_2 does not, s_2 is exercised at most as many times as s_1 is exercised;

(iii) For any input file on which P_2 terminates normally but P_1 does not, s_1 is exercised at most as many times as s_2 is exercised.

Consider statements s_1 and s_2 in programs *Old* and *New*, respectively, such that s_1 and s_2 are in the same execution class. There is a test t that exercises s_1, since T is statement adequate; furthermore, since s_1 and s_2 are in the same execution class, s_2 is also exercised by t (provided that *New* terminates normally on t). Any class that contains statements of both *Old* and *New* will thus provide tests for the statements of *New* in the class.

To provide an algorithm for partitioning the statements of *Old* and *New* into execution classes, we define a new kind of slice called a *control slice*:

DEFINITION (control slice). A *control slice* of a PDG G with respect to a vertex v is the PDG slice of a PDG G' with respect to v', where:

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

(ii) G' contains all the edges of G, plus new edges $\{ p \longrightarrow_c^L v' \mid p \longrightarrow_c^L v \}$.[6]

That is, to take a control slice with respect to vertex v, create a new vertex v' with the same control predecessors as v but with no flow predecessors, and take the PDG slice with respect to v'.

The following lemma demonstrates that control slice isomorphism provides a safe criterion for partitioning statements into execution classes.

LEMMA 3.1. If the statements of programs P_1 and P_2 are partitioned into classes based on control slice isomorphism, then all the statements in each class have equivalent execution patterns.

PROOF. Let G_1 and G_2 be the PDGs of P_1 and P_2, respectively, and let v_1 and v_2 be vertices in G_1 and G_2, respectively, whose control slices are isomorphic. There are two cases to consider: v_1 and v_2 are while-loop predicates, and v_1 and v_2 are any other type of vertex (it is impossible for v_1 to be a while-loop predicate if v_2 is not, or *vice versa*, because in that case they would have non-isomorphic control slices). If v_1 is a while-loop predicate, G_1 contains a control edge $v_1 \longrightarrow_c^T v_1$; hence G' contains a control edge $v_1 \longrightarrow_c^T v'$. Therefore the control slice of G_1 with respect to v_1 contains the PDG slice of G_1 with respect to v_1, and similarly for v_2. Furthermore, the PDG slice of G_1 with respect to v_1 is isomorphic to the PDG slice of G_2 with respect to v_2. Hence v_1 and v_2 have equivalent behavior. It is clear from the definitions that vertices with equivalent behavior have equivalent execution patterns, so we are done.

Now suppose v_1 and v_2 are not while-loop predicates. Let w_1 and w_2 be the control predecessors of v_1 and v_2 with labels L_1 and L_2, respectively; then L_1=L_2 and the PDG slice of G_1 with respect to w_1 is isomorphic to the PDG slice of G_2 with respect to w_2. By Theorem 2.1, the

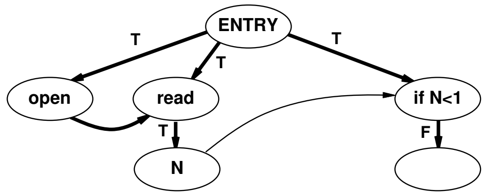


Figure 4. Control slice of Version 1 with respect to "*oddP* := 0" and of Version 2 with respect to "*evenP* := 1".

[6]This set can contain zero, one, or two edges; the *ENTRY* vertex has no control predecessors, a vertex that represents a while-loop predicate has two control predecessors, and all other vertices have exactly one control predecessor.

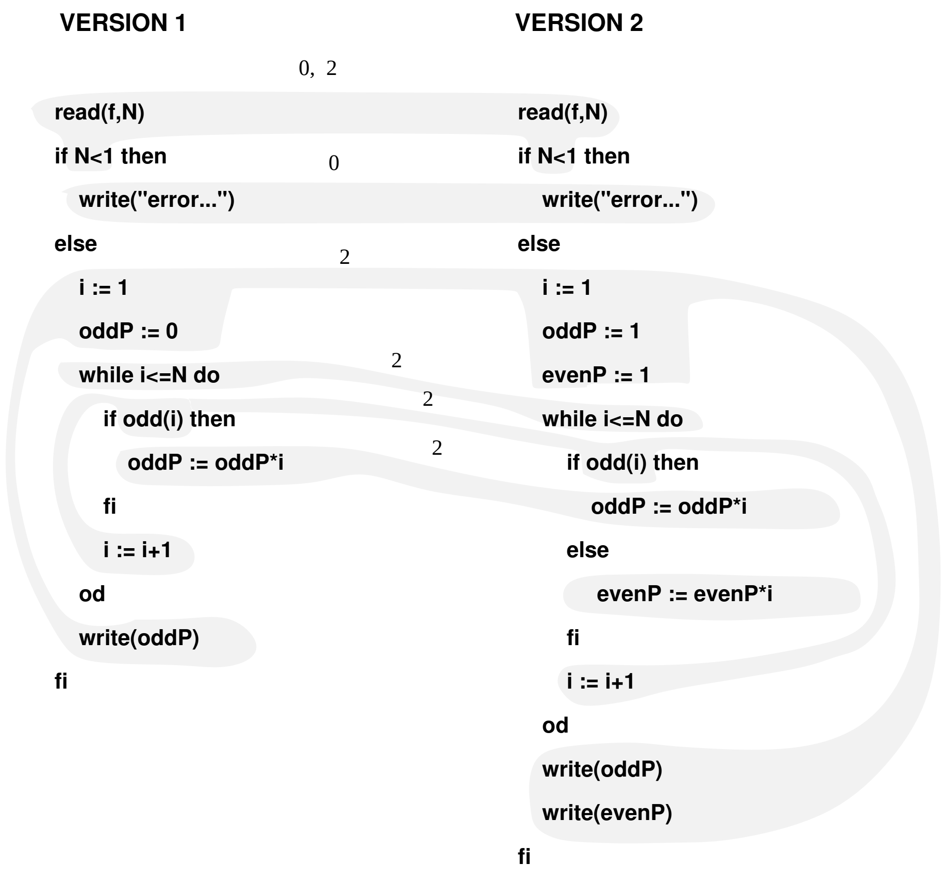


Figure 5. The statements in the programs of Figure 1 partitioned into classes based on control slice isomorphism. Listed above each class is the set of tests that exercise the statements in that class.

predicates w_1 and w_2 have equivalent behavior. We demonstrate that parts (i) and (ii) of the definition of equivalent execution patterns are satisfied; part (iii) follows by symmetry from part (ii).

(i) Let t be a test on which both P_1 and P_2 halt; then $P_1(t)(w_1) = P_2(t)(w_2)$. The vertex v_1 is exercised exactly once for every occurrence of L_1 in the sequence $P_1(t)(w_1)$, since P_1 halts (and similarly for vertex v_2 in P_2). Since $P_1(t)(w_1) = P_2(t)(w_2)$, t exercises v_1 and v_2 the same number of times.

(ii) Let t be a test on which P_1 halts but P_2 does not; then $P_2(t)(w_2)$ is a prefix of $P_1(t)(w_1)$. Let L_2 occur n times in $P_2(t)(w_2)$; v_2 is exercised at most n times. Since $P_2(t)(w_2)$ is a prefix of $P_1(t)(w_1)$, L_1 occurs at least n times in $P_1(t)(w_1)$; P_1 halts, so v_1 is exercised at least n times. Thus v_2 is exercised by t at most as many times as v_1 is exercised by t. □

Example. Figure 4 shows the PDG that is the control slice of Version 1 with respect to the statement "*oddP* :=0;" this PDG is also the control slice of Version 2 with respect to the statement "*evenP* :=1". Because the two statements have the same control slice, they are in the same execution class (and are exercised by the same set of tests). Figure 5 shows all of the statements of Version 1 and Version 2 partitioned into classes based on control slice isomorphism. For each class, the tests that exercise the statements of Version 1 that are in the class are listed; these tests will also exercise the statements of Version 2 that are in the same class. Notice that the class containing "*evenP* :=*evenP*i*" contains no statement of Version 1; hence the technique finds no test in T to exercise "*evenP* :=*evenP*i*". (In fact, the test "2" *will* exercise the statement "*evenP* :=*evenP*i*." This will be discovered if the programmer uses the retesting process given at the beginning of Section 3.) □

An algorithm for determining which tests in T can be used to exercise statements in *New* is given in Figure 6. The first step, partitioning the statements according to control slice isomorphism, can be done in (expected) time proportional to the sum of the sizes of the statements' control slices. [Hor91] One of the inputs to the algorithm is a function f_{Old} mapping each statement of *Old* to the set of tests in T that exercise it; this function can be constructed from the sets s_t mentioned above. The output of the algorithm is a

```
function IdentifyReusableTests(
    Old, New : programs;
    f_Old: map of statements of Old to sets of tests in T)
returns map of statements of New to sets of tests in T
begin
  partition the statements of Old and New into classes based
     on control slice isomorphism
  for each statement s of New do
        f_New(s) := ∅
        for each statement s′ of Old in the same class as s do
             add the tests in f_Old(s′) to f_New(s)
        endfor
  endfor
  return f_New
end
```

Figure 6. Algorithm for determining which tests can be reused.

function f_{New} mapping each statement of *New* to a set of tests in *T* that will exercise it (provided *New* halts on the test). The algorithm also identifies statements that may require new tests to be created; namely, those statements that are mapped to the empty set by f_{New}.

3.2. Deciding Which Tests to Rerun

The techniques of the previous section will match some statements of *New* with each reused test. In this section we describe how to select a subset of the reused tests to be rerun; a test is selected only if it is identified as exercising a potentially affected statement of *New*.

DEFINITION (affected statements). A statement *s* of *New* is *affected* iff one of the following holds:

(1) there is no corresponding statement in *Old*; or

(2) the behavior of *s* in *Old* is not equivalent to the behavior of *s* in *New*.

As discussed in Section 2.2, if the PDG slices with respect to statements s_1 and s_2, in programs *Old* and *New* respectively, are isomorphic, then s_1 and s_2 have equivalent behavior; therefore, we can use isomorphism of PDG slices to identify *un*affected statements of *New*. The reused tests that are selected to be rerun are those that are matched with statements of *New* that have no corresponding statement in *Old* or whose PDG-slice is not isomorphic to the PDG-slice of the corresponding statement of *Old*.

Figure 7 gives an algorithm for selecting tests to rerun. Using the techniques of [Hor91], isomorphism of two PDG-slices can be determined in worst-case time proportional to the size of the smaller of the two slices. Thus, in the worst case, the time required for the algorithm of Figure 7 is proportional to the number of statements in *New*, plus the sum of the sizes of the slices with respect to statements of *New* that have a corresponding statement in *Old*.

Example. Figure 1 shows the statements of Version 2 that are identified as affected with respect to Version 1.

```
function IdentifyTestsToRerun(
    Old, New : programs;
    f_New: map of statements of New to sets of tests in T)
returns set of tests to rerun
declare affected: set of statements of New
begin
 /*Step 1: Identify (a superset of) the affected statements of New*/
    Rerun := ∅
    for each statement s of New do
      if s has no corresponding statement in Old
      then add s to affected
      else let s′ be the statement of Old that corresponds to s in
             if the PDG slice of New with respect to s is not
                isomorphic to the PDG slice of Old
                with respect to s′
             then add s to affected
             endif
           endlet
      endif
    endfor
 /*Step 2: Identify reused tests that are known to test a
 statement of New in affected*/
    Rerun := ∅
    for every statement s of New do
       if s is in affected
       then add the tests in f_New(s) to Rerun
       endif
    endfor
  return Rerun
end
```

Figure 7. Algorithm for determining which reused tests should be rerun.

Statements "*evenP* :=1", "*evenP* :=*evenP*i*", and "**write**(*evenP*)" are identified as affected because they have no corresponding statements in *Old*; statements "*oddP* :=1", "*oddP* :=*oddP*i*", and "**write**(*oddP*)" are identified as affected because their PDG slices are not isomorphic to the PDG slices of the corresponding statements of *Old* (in particular, the PDG slices in *Old* include the vertex "*oddP* :=0", while the PDG slices in *New* include the vertex "*oddP* :=1"). (In this example, PDG-slice isomorphism can be used to identify the affected statements of *New exactly*; in general, this technique may identify as affected some statements whose behaviors are in fact equivalent to the corresponding statements of *Old*.)

Figure 5 shows that affected statements "*oddP* :=1", "*evenP* :=1", "*oddP* :=*oddP*i*", "**write**(*oddP*)" and "**write**(*evenP*)" are exercised by the test "2", so this test should be rerun. The test "0" exercises only unaffected statements, so it need not be rerun. □

4. APPLICATION TO PDG EDGE ADEQUACY CRITERIA

The techniques described in Section 3 can be applied to the PDG edge criteria as well as the all-vertices criterion. As an example, we describe how to define the all-flow-edges criterion and how to apply the techniques to incremental testing with respect to it; the application of the techniques to the all-control-edges criterion is discussed in [Bat92a]. First, we must we define several fundamental concepts about the execution and behavior of a flow edge; in what follows, we explain each concept and give its definition for the all-flow-edges criterion.

The first concept that is required is the definition of what it means to *exercise* a flow edge. A vertex is exercised by a test if the program execution on the test includes an instance of the statement corresponding to the vertex. A flow edge $x \longrightarrow_f y$ corresponds to one or more paths in the control-flow graph beginning with x defining a variable v, ending with y using v, and containing no other definition of v along the path. The edge is exercised by a test if the program execution includes an instance of at least one of the paths corresponding to the flow edge. The program execution may include several instances of v-definition-free paths from x to y; the flow edge is exercised once for each such instance.

In addition, the algorithm IdentifyReusableTests relies on a method for identifying classes of components that have equivalent execution patterns. Section 3.1 showed that comparison of control slices can be used to group vertices into the appropriate classes. Although the definition of equivalent execution patterns was given in Section 3.1 in terms of statements, the definition extends to any program component for which a definition of the component's being exercised can be given.

We now define the control slice of a PDG with respect to a flow edge so that comparison of control slices can also be used to group flow edges that have equivalent execution patterns. The definition is motivated by the following observation: the edge $x \longrightarrow_f y$ is *not* exercised if x or y is not exercised, or if at least one instance of a definition of v is exercised after every instance of x and before every instance of y.

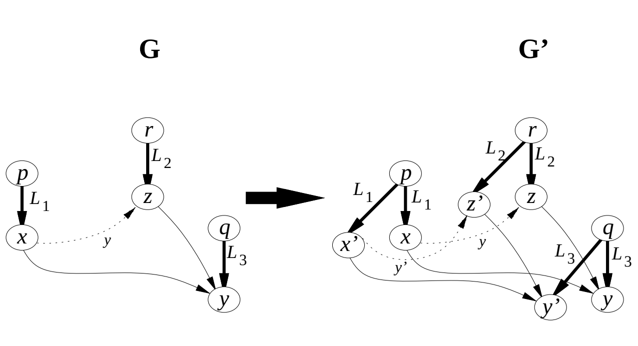


Figure 8. Graph transformation used in defining the control slice with respect to the edge $x \longrightarrow_f y$. Each def-order edge is labeled with its witness vertex.

DEFINITION (control slice with respect to a flow edge). The *control slice* of a PDG G with respect to a flow edge $e = x \longrightarrow_f y$ is the PDG slice of a PDG G' with respect to y', where

(i) G' contains all the vertices of G, plus unlabeled vertices x', y' and, for every vertex z such that $x \longrightarrow_{do(y)} z$, an unlabeled vertex z';[7]

(ii) G' contains all the edges of G, plus the edges $\{ p \longrightarrow_c^L x' \mid p \longrightarrow_c^L x \} \cup \{ q \longrightarrow_c^L y' \mid q \longrightarrow_c^L y \} \cup \{ r \longrightarrow_c^L z' \mid r \longrightarrow_c^L z \} \cup \{ x' \longrightarrow_f y' \mid x \longrightarrow_f y \} \cup \{ z' \longrightarrow_f y' \mid z \longrightarrow_f y \} \cup \{ x' \longrightarrow_{do(y')} z' \}$.

That is, to take the control slice with respect to edge $e = x \longrightarrow_f y$: (1) create new vertices x', y', and z' (where z is the target of a def-order edge from x with witness y) that have the same control predecessors as their counterparts but no flow predecessors; (2) create edges among x', y' and z' that parallel the edges among their counterparts; and (3) take the PDG slice with respect to y'.

Figure 8 shows the construction of G'. Isomorphism of control slices provides the partitioning method required, as the following theorem shows.

THEOREM 4.1. If the flow edges of P_1 and P_2 are partitioned into classes based on control slice isomorphism, then all the flow edges in each class have equivalent execution patterns.

Proof sketch. Let $e_1 = x_1 \longrightarrow_f y_1$ and $e_2 = x_2 \longrightarrow_f y_2$ be flow edges on a variable v with isomorphic control slices. The following conditions are sufficient to conclude that e_1 and e_2 have equivalent execution patterns:

(i) corresponding endpoints of the edges have equivalent execution patterns;

(ii) corresponding definitions of v that occur on control flow graph paths from x_1 to y_1 (from x_2 to y_2) have

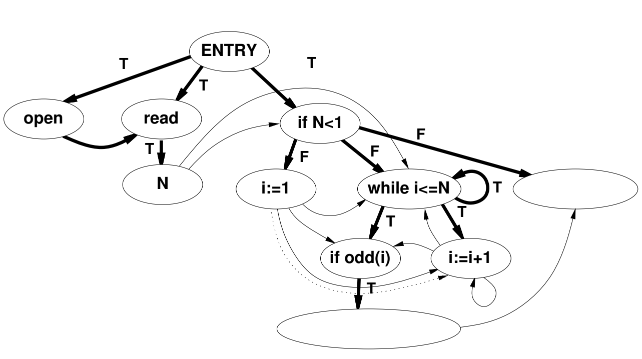


Figure 9. Control slice of Versions 1 and 2 with respect to the flow edge "*oddP* := *oddP* * *i* $\longrightarrow_f$ **write**(*oddP*)."

[7]Some of these vertices may be the same; for instance, if $x = y$, then $x' = y'$.

equivalent execution patterns;

(iii) if both programs halt on a test, then there is an occurrence of the pattern $x_1 ... y_1$ in P_1's execution trace iff there is a corresponding occurrence of the pattern $x_2 ... y_2$ in P_2's execution trace, and corresponding instances of the patterns occur in the same order in the two traces;

(iv) corresponding instances of the patterns $x_1 ... y_1$ and $x_2 ... y_2$ both contain an instance of a definition of v or neither does;

(v) if one of the programs does not terminate on a test, the sequence of instances of the pattern in the nonterminating program's execution trace is a prefix of the sequence of instances of the corresponding pattern in the terminating program's execution trace.

The proof involves showing that if e_1 and e_2 have isomorphic control slices, then conditions (i)-(v) hold. The full proof appears in [Bat92a].

Example. Figure 9 shows the control slice of Version 1 with respect to the flow edge from "*oddP* :=*oddP***i*" to "**write**(*oddP*);" it is also the control slice of Version 2 with respect to the same flow edge. Hence these edges have equivalent execution patterns and are both tested by the test "2". □

The algorithm IdentifyTestsToRerun relies on the identification of affected components in the modified program. The definition of "affected" given in Section 3.2 depends on the concept of equivalent behavior. Thus we need to define the *behavior* of an edge. In Section 2.2, the behavior of a vertex is defined to be the sequence of values computed at the vertex. We define the behavior of a flow edge similarly as the sequence of values that flow along the edge. A value V flows along an edge if the variable defined at the source of the edge gets value V, and a definition-free path is traversed during execution to the target of the edge. The definition of equivalent behavior given in Section 2.2 then applies to flow edges as well as to statements and predicates.

We also need a safe way to identify components with equivalent behavior; this allows us to identify (a superset of) the components that are affected. Section 3.2 showed that comparison of PDG slices can be used to identify vertices whose corresponding statements have equivalent behavior. We now define a PDG slice with respect to a flow edge in order to get a similar result.

DEFINITION (PDG slice with respect to a flow edge). The *PDG slice* of a PDG G with respect to a flow edge $e = x \longrightarrow_f y$ is the PDG slice of a PDG G' with respect to y', where

(i) G' contains all the vertices of G, plus an unlabeled vertex y' and, for every vertex z such that $x \longrightarrow_{do(y)} z$, an unlabeled vertex z';

(ii) G' contains all the edges of G, plus the edges $\{ q \longrightarrow_c^L y' \mid q \longrightarrow_c^L y \} \cup \{ r \longrightarrow_c^L z' \mid r \longrightarrow_c^L z \} \cup \{ x \longrightarrow_f y' \mid x \longrightarrow_f y \} \cup \{ z' \longrightarrow_f y' \mid z \longrightarrow_f y \} \cup \{ x \longrightarrow_{do(y')} z' \}$.

Figure 10 shows the construction of G'. Isomorphism of PDG slices provides a safe way to identify flow edges with equivalent behavior, as the following theorem shows.

THEOREM 4.2. If P_1 and P_2 are programs and e_1 and e_2 are flow edges in the PDGs G_{P_1} and G_{P_2}, respectively, such that the PDG slices with respect to e_1 and e_2 are isomorphic, then e_1 and e_2 have equivalent behavior.

Proof sketch. There are two parts to the proof:

(i) Show that e_1 and e_2 have equivalent execution patterns; and

(ii) Show that identical values flow along the edges in corresponding instances.

Let $e_1 = x_1 \longrightarrow_f y_1$ and $e_2 = x_2 \longrightarrow_f y_2$. The PDG slice with respect to a flow edge contains the control slice with respect to the edge; hence the control slices with respect to

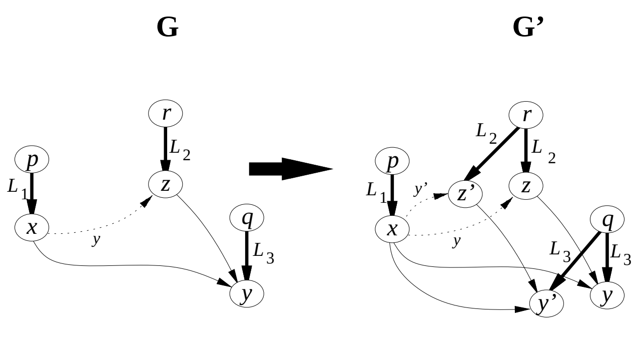


Figure 10. Graph transformation used in defining the PDG slice with respect to $x \longrightarrow_f y$. Each def-order edge is labeled with its witness vertex.

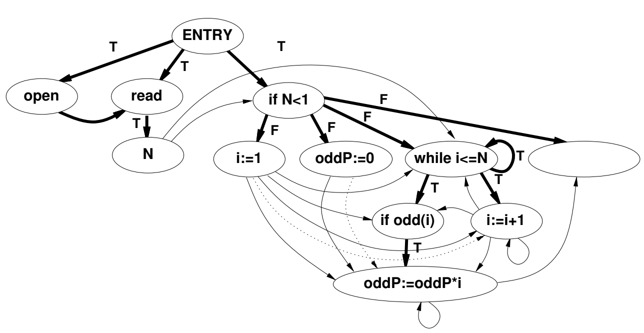


Figure 11. PDG slice of Version 2 with respect to the flow edge "*oddP* :=*oddP***i* $\longrightarrow_f$ **write**(*oddP*)."

e_1 and e_2 are isomorphic. By Theorem 4.1, e_1 and e_2 have equivalent execution patterns. Furthermore, properties (iii)-(v) in the proof of Theorem 4.1 hold. Hence corresponding instances of the patterns $x_1...y_1$ and $x_2...y_2$ contain corresponding instances of x_1 and x_2.

The PDG slice with respect to a flow edge also contains the PDG slice with respect to the source of the edge; the isomorphism of the flow-edge PDG slices implies that x_1 and x_2 have isomorphic PDG slices. By Theorem 2.1, x_1 and x_2 have equivalent behavior. Hence corresponding instances of x_1 and x_2 compute the same value. Since the value that flows along e_1 (respectively, e_2) is defined by x_1 (respectively, x_2), identical values flow along the edges in corresponding instances. □

Example. Figure 11 shows the PDG slice of Version 1 with respect to the flow edge from "*oddP*:=*oddP***i*" to "**write**(*oddP*);" it differs from the PDG slice of Version 2 with respect to the same flow edge in the initial definition of *oddP*. Hence the edge is identified as affected. It does in fact have different behavior from the corresponding edge in Version 1; on the test "2", the value 0 flows along the edge in Version 1, but the value 1 flows along the edge in Version 2. □

To summarize: Recall that an adequacy criterion is defined by specifying a set of program components and defining what it means for a component to be exercised. To permit the retesting techniques presented in Section 3 to apply to a new adequacy criterion, one must also define

(1) a method for grouping components with equivalent execution patterns;

(2) the behavior of a component; and

(3) a method for identifying components with equivalent behavior.

We have given these definitions for the all-flow-edges criterion, and have demonstrated that the definitions satisfy the required properties.

5. RELATED WORK

The work presented here makes contributions in two areas by addressing the problems of incremental program testing with respect to an adequacy criterion, and by defining three new kinds of PDG slice (the control slice with respect to a vertex and a flow edge, and the PDG slice with respect to a flow edge). Accordingly, there are two kinds of related work. In the area of incremental testing, previous work has used the idea of identifying affected def-use pairs as an aid to selecting tests for regression testing. Taha, Thebaut, and Liu [Tah89] describe a method using incremental data flow analysis to partition a test suite for a program into three sets that are relevant for a modification of the program: the *retestable test cases*, which exercise changed paths in the program; the *reusable test cases*, which exercise unchanged paths in the program; and the *obsolete test cases*, which are not valid for the modified program. Ostrand and Weyuker [Ost88] present a similar method of determining the def-use associations that are affected by a program change, and use the results to decide which tests of an adequate test suite need to be rerun and which def-use associations require new tests. Harrold and Soffa [Har] use incremental data flow analysis to identify affected def-use pairs and to select tests for retesting these pairs.

None of the three techniques precisely defines what it means for a flow dependence to be *affected*. All three techniques identify as affected only those flow dependences for which either the definition or the use has undergone an editing change; they fail to take into account changes that transitively affect a flow dependence. Furthermore, multiple editing changes to the program can cause the techniques to select tests for retesting that do not exercise the flow dependences that they are intended to test. In contrast, our techniques for incremental testing, when applied to the all-flow-edges criterion, take into account the transitive effects of a modification, and identify a superset of the affected flow dependences identified by the three techniques. Also, our techniques guarantee that a test selected to exercise a component does in fact exercise it, provided that the program terminates.

A by-product of our work on incremental program testing is a technique that groups program components for coverage by a single test; there has been other work on developing such techniques. Gupta and Soffa [Gup91] order program statements and def-use pairs using the post-dominance relation on the control-flow graph; the ordering is used to guide the generation of tests for a test suite adequate under the all-statements or all-def-use-pairs criterion. Chusho [Chu83Chu87] defines an ordering of control flow graph edges based on execution behavior; a *primitive edge* is one for which no other edge's execution guarantees its execution. He then presents an algorithm for reducing the control flow graph by combining each non-primitive edge with the primitive edge that guarantees its execution. A test that exercises a primitive edge *e* is guaranteed to exercise the non-primitive edges associated with *e*. In both Chusho's and Gupta and Soffa's work, program components are ordered rather than partitioned into equivalence classes; a component that precedes another in the ordering will be exercised if the latter component is exercised but not necessarily *vice versa*. Furthermore, these techniques will only permit the ordering of components within a *single* program; our technique, testing isomorphism of control slices, applies across programs.

In the area of PDG slicing, Schatz and Ryder [Sch92] define a PDG slice called the *condition-preserving slice* that is similar to the control slice defined here (condition-preserving slices are used to identify race conditions in a parallel program). The control slice defined in this paper yields a subgraph of the condition-preserving slice; the latter graph can include flow edges that are not in the control slice. As a consequence, testing condition-preserving slice isomorphism is a less precise method of partitioning statements into execution classes than testing control slice isomorphism.

6. SUMMARY

We have presented techniques that use the program dependence graph and PDG slicing to address the problems of incremental testing. We have also defined three new PDG slices, and shown a semantic property for each one:

- the *control slice* with respect to a vertex or a flow edge has the property that two components with isomorphic control slices have equivalent execution patterns;
- the *PDG slice* with respect to a flow edge has the property that two flow edges with isomorphic PDG slices have equivalent behaviors.

The algorithms of Figures 6 and 7 can be used to automate the selection of tests for a modified program from an adequate test suite for the original program. The first algorithm identifies tests that can be reused to test statements of the modified program; the second algorithm identifies a safe approximation to the set of affected statements of the modified program, and identifies the tests that are guaranteed to exercise those statements.

Our approach represents an important improvement over previous approaches to incremental testing. The algorithms used in previous work to match components of the modified program with existing test files can match a component with a test file that fails to exercise it. Furthermore, because previous algorithms for identifying affected components are based on a *syntactic* rather than a *semantic* definition of "affected," they may fail to take into account the transitive effects of a modification.

ACKNOWLEDGEMENTS

We would like to thank Thomas Ball and Paul Adams for their many helpful comments.

REFERENCES

Aho86.
Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).

Bat92a.
Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs," technical report in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI (1992).

Bat92b.
Bates, S. and Horwitz, Susan, "On integrating programs with input," in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI (1992).

Bat92c.
Bates, S. and Horwitz, Susan, "Test Data Adequacy Criteria Based on Program Dependences," in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI (1992).

Chu83.
Chusho, Takeshi, "Coverage measure for path testing based on the concept of essential branches," *Journal of Information Processing* **6**(4) pp. 199-205 (1983).

Chu87.
Chusho, Takeshi, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing," *IEEE Transactions on Software Engineering* **SE-13**(5) pp. 509-517 (May 1987).

Cla76.
Clarke, Lori A., "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering* **SE-2**(3) pp. 215-222 (September 1976).

DeM91.
DeMillo, Richard A. and Offutt, A. Jefferson, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering* **17**(9) pp. 900-910 (September 1991).

Don84.
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 (1984).

Fer87.
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).

Gup91.
Gupta, Rajiv and Soffa, Mary Lou, "Automatic generation of a compact test suite," Technical Report, University of Pittsburgh, Pittsburgh, PA (1991).

Har.
Harrold, Mary Jean and Soffa, Mary Lou, "An incremental approach to unit testing during maintenance," *Proceedings of the Conference on Softwrae Maintenance* (Phoenix, Arizona, October 24-27, 1988), pp. 362-367 ().

Hor88.
Horwitz, S., Prins, J., and Reps, T., "On the adequacy of program dependence graphs for representing programs," pp. 146-157 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), (1988).

Hor89a.
Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).

Hor89b.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems* **11**(3) pp. 345-387 (July 1989).

Hor90.
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).

Hor91.
Horwitz, S. and Reps, T., "Efficient comparison of program slices," *Acta Informatica* **28** pp. 713-732 (1991).

How76.
Howden, William E., "Reliability of the path analysis testing strategy," *IEEE Transactions on Software Engineering* **SE-**

2(3) pp. 208-215 (September 1976).

Hua75.
Huang, J. C., “An approach to program testing,” *ACM Computing Surveys* **7**(3) pp. 113-127 (September 1975).

Kor90.
Korel, Bogdan, “Automated software test data generation,” *IEEE Transactions on Software Engineering* **16**(8) pp. 870-879 (August 1990).

Kra73.
Krause, K. W., Smith, R. W., and Goodwin, M. A., “Optimal software test planning through automated network analysis,” *Record of the 1973 IEEE Symposium on Computer Software Reliability* (New York, New York), pp. 18-22 (April 30-May 2, 1973).

Lar88.
Larus, J.R. and Hilfinger, P.N., “Detecting conflicts between structure accesses,” *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), pp. 21-34 (June 1988).

Las83.
Laski, Janusz W. and Korel, Bogdan, “A data flow oriented program testing strategy,” *IEEE Transactions on Software Engineering* **SE-9**(3) pp. 347-354 (May 1983).

Mar90.
Marick, Brian, “A survey of test effectiveness and cost studies,” Report No. UIUCDCS-R-90-1652, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL (December 1990).

Not85.
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).

Nta84.
Ntafos, Simeon C., “On required element testing,” *IEEE Transactions on Software Engineering* **SE-10**(6) pp. 795-803 (November 1984).

Ost88.
Ostrand, Thomas J. and Weyuker, Elaine J., “Using data flow analysis for regression testing,” *Proceedings of the Sixth Annual Pacific Northwest Software Quality Conference*(Portland, Oregon), (September 19-20, 1988).

Ott84.
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, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

Rap85.
Rapps, Sandra and Weyuker, Elaine J., “Selecting software test data using data flow information,” *IEEE Transactions on Software Engineering* **SE-11**(4) pp. 367-375 (April 1985).

Rep88.
Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A system for constructing language-based editors*, Springer-Verlag, New York, NY (1988).

Rep89.
Reps, T. and Yang, W., “The semantics of program slicing and program integration,” pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages*, (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science* Vol. *352*, Springer-Verlag, New York, NY (March 1989).

Sch92.
Schatz, E. and Ryder, B. G., “Directed tracing to detect race conditions,” LCSR-TR-176, Laborator for Computer Science Research, Rutgers University, New Brunswick, NJ (February 1992).

Tah89.
Taha, Abu-Bakr, Thebaut, Stephen M., and Liu, Sying-Syang, “An approach to software fault localization and revalidation based on incremental data flow analysis,” *Proceedings of the Thirteenth Annual International Computer Software & Applications Conference* (Orlando, Florida), pp. 552-558 (September 20-22, 1989).

Wei84.
Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

Yan91.
Yang, W., “Identifying syntactic differences between two programs,” *Software – Practice & Experience* **21**(7) pp. 739-755 (July 1991).