The Use of Program Dependence Graphs in Software Engineering

Susan Horwitz and Thomas Reps
University of Wisconsin

ABSTRACT

This paper describes a language-independent program representation—the *program dependence graph*—and discusses how program dependence graphs, together with operations such as program slicing, can provide the basis for powerful programming tools that address important software-engineering problems, such as understanding what an existing program does and how it works, understanding the differences between several versions of a program, and creating new programs by combining pieces of old programs.

The paper primarily surveys work in this area that has been carried out at the University of Wisconsin during the past five years.

1. INTRODUCTION

A fundamental goal of software engineering is to make program development and maintenance easier, faster, and less error prone. This includes addressing problems like

- understanding what an existing program does and how it works,
- understanding the differences between several versions of a program,
- creating new programs by combining pieces of old programs.

Tools that assist programmers with such problems are most useful if they are *language based*, that is, if they incorporate knowledge about the programming language in use. On the other hand, it is desirable to base these tools on language-independent algorithms and data structures so as to avoid the need to re-design and re-develop a set of tools for every different programming language.

This paper describes a language-independent program representation—the *program dependence graph* [12, 18, 24]—and discusses how program dependence graphs, together with operations such as program slicing, can form the basis for powerful programming tools that address the problems listed above. Our focus is primarily on work done at the University of Wisconsin during the past five years (and some of the material presented in this paper is excerpted from previous publications). Related work by other groups is cited, but is not explored in detail. Uses of program dependence graphs for purposes outside the scope of software engineering (for example, for discovering potential parallelism in sequential programs) is not discussed.

The goal of our work has been to address the problems listed above by using knowledge about the program obtained through static analysis—in particular, knowledge about the execution behavior at the different points in the program. (By a "program point" we mean an assignment statement, an output statement, or a predicate of an if-then-else statement or while loop.) Questions concerning the execution behavior at a given point p in a program are ordinarily addressed by considering the sequence of *states* that would arise at p when the program is run on some initial state. However, this approach is incompatible with the goals of our work, which deals with program "projections"—programs in which some of the statements have been removed—and "program variants"—programs that have been modified by the user (for example, to fix a bug or to extend its functionality). The problem is that the sequence of states that arise at related points in two program variants (or in a program and one of its projections) are almost certain to differ. For example, any modification that introduces a new program variable will change the sequence of states at *every* point in the program; a modification that changes the way even a single existing program variable is used is likely to change the sequences of states produced at most points in the program. Consequently, for our work it is necessary to use an alternative approach to capturing the behavior at a point in a program.

For our work, the execution behavior exhibited at program point p is considered to be the *sequence of values* produced at p when the program is run on a given initial state:

(i) if p is an assignment statement, the behavior at p is the sequence of values assigned to the left-hand-side variable;
(ii) if p is an output statement, the behavior at p is the sequence of values written to the output;
(iii) if p is a predicate, the behavior at p is the sequence of boolean values to which p evaluates.

To build tools that can help programmers to understand the differences between several versions of a program or to create new programs by combining pieces of old programs, it is necessary to be able to determine when different program points have *equivalent behaviors*. Given program points p_1 and p_2 in programs P_1 and P_2, respectively, we say that p_1 and p_2 have equivalent behavior if for every initial state on which both P_1 and P_2 terminate normally, p_1 and p_2 produce the same sequence of values. (A definition of equivalent behavior that takes the possibility of nontermination into account can be found in [19, 44, 46]; for the purposes of this paper, we limit our scope of interest to terminating programs.) Although behavioral equivalence is undecidable in general, it is possible to develop *safe* algorithms for detecting equivalent components (see Sections 3 and 4).

The problems we are concerned with fall into three different classes, depending on whether the problem deals with one program, two programs, or three (or more) programs:

Slicing problems (one program)

S.1. For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at q were different then the behavior at p would be different.

S.2 (the dual of S.1).
For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at p were different then the behavior at q would be different.

S.3. For a given program point p in program P, find a projection Q of P such that when P and Q are run on the same initial state, the behaviors at p in P and Q are identical.

Differencing problems (two programs)

D.1. Given programs *Old* and *New*, and a correspondence between their components, find [a superset of] the components of *New* whose behaviors differ from the corresponding components of *Old*.

D.2. Given programs *Old* and *New*, but with *no* correspondence between their components, find [a superset of] the components of *New* for which there is no component in *Old* with the same behavior.

D.3. Given programs *Old* and *New*, find a program *Changes* that exhibits all changed behaviors of *New* with respect to *Old*.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by NSF under grants DCR-8552602, DCR-8603356, CCR-8958530, and CCR-9100424, by DARPA, monitored by ONR under contract N00014-88-K-0590, as well as by grants from the Cray Research Foundation, DEC, Eastman Kodak, IBM, 3M, and Xerox.

Authors' address: Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

Integration problems (three or more programs)

I.1. Given a program *Base* and two or more variants, together with a correspondence between the programs' components, determine whether the changes made to create the variants from *Base* are compatible. If the changes are compatible, combine the variants to form a single, integrated program; otherwise identify the source of the interference.

I.2. Same problem as I.1, but with *no* correspondence between the components of *Base* and the variants.

For each set of problems, we first address the case of single-procedure programs and then extended our solution to multi-procedure programs.[1] The different problems will be discussed at different levels of detail. For some of them, we give an algorithm; for others (for which the solution is more involved) we explain the key issues, describe the general approach, and give references to sources of more complete descriptions.

The eight problems are organized in tabular form in Figure 1. The table indicates where each problem is discussed in this paper, and also gives references to the literature.

The remainder of the paper is organized as follows: Section 2 defines the program dependence graph (and two related graph representations of programs—*system* dependence graphs and program *representation* graphs). Section 3 discusses program slicing. Section 4 discusses several approaches to identifying the differences between two versions of a program. Section 5 describes algorithms for program integration. Section 6 provides information about a prototype system implemented at the University of Wisconsin that incorporates the ideas described in the paper.

2. DEPENDENCE GRAPHS

Different definitions of program dependence representations have been given, depending on the intended application; however, they are all variations on a theme introduced in [23] and share the common feature of having explicit representations of both control dependences and data dependences. In Section 2.1 we define *program dependence graphs*, which can be used to represent single-procedure programs; that is, programs that consist of a single main procedure, with no procedure or function calls. In Section 2.2 we extend program dependence graphs to *system* dependence graphs, which represent programs with multiple procedures and functions. In Section 2.3 we discuss a variant of the program dependence graph called the program *representation* graph.

2.1. The Program Dependence Graph

In this section we discuss how to build a program dependence graph for a program written in a restricted language including only scalar variables, assignment statements, if-then-else statements, output statements, and while loops. (Input statements are not included; however, programs are assumed to be run on an initial state, so variables can be used before being defined. Such variables take their values from the initial state.) Techniques for handling arbitrary control constructs, arrays, and pointers can be found in [4, 12, 17, 26, 42].

The *program dependence graph* (or PDG) for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges. The vertices in G_P represent the assignment statements and predicates of P. In addition, G_P includes a special *Entry* vertex, and also includes one *Initial definition* vertex for every variable x that may be used before being defined. (This vertex represents an assignment to the variable from the initial state.)

The edges of G_P represent *control* and *data* dependences. The source of a control dependence edge is always either the *Entry* vertex or a predicate vertex; control dependence edges are labeled either **true** or **false**. The intuitive meaning of a control dependence edge from vertex v to vertex w is the following: if the program component represented by vertex v is evaluated during program execution and its value matches the label on the edge, then, assuming that the program terminates normally, the component represented by w will eventually execute; however, if the value does not match the label on the edge, then the component represented by w may never execute. (By definition, the *Entry* vertex always evaluates to **true**.)[2]

For the restricted language under consideration here, control dependence edges reflect the nesting structure of the program (*i.e.*, there is an edge labeled **true** from the vertex that represents a *while* predicate to all vertices that represent statements nested immediately within the loop; there is an edge labeled **true** from the vertex that represents an *if* predicate to all vertices that represent statements nested immediately within the true branch of the *if*, and an edge labeled **false** to all vertices that represent statements nested immediately within the false branch; there is an edge labeled **true** from the *Entry* vertex to all vertices that represent statements that are not nested inside any *while* loop or *if* statement).

Data dependence edges include both *flow* dependence edges and *def-order* dependence edges.[3] Flow dependence edges represent possible flow of values, *i.e.*, there is a flow dependence edge from vertex v to vertex w if vertex v represents a program component that assigns a value to some variable x, vertex w represents a com-

[2]Podgurski and Clarke have explored an alternative concept that they call *weak control dependence*, which accounts for the control effects of potentially nonterminating constructs [31].

[3]Some definitions of program dependence graphs include output dependences and anti dependences in place of def-order dependences. See [16] for a comparison of the two approaches.

	Slicing problems			Differencing problems			Integration problems	
	S.1	S.2	S.3	D.1	D.2	D.3	I.1	I.2
Single-procedure programs	Sect. 3.1. [30], [18], [44]	Sect. 3.1. [20]	Sect. 3.1. [41]	Sect. 4.1.1. [18], [19]	Sect. 4.1.2. [44], [19], [21]	Sect. 4.1.1.	Sect. 5.1.1 and 5.1.2. [18], [44], [46]	Sect. 5.1.3. [38], [21]
Multi-procedure programs	Sect. 3.2. [20], [22], [8]	Sect. 3.2. [20]	Sect. 3.2. [41]	Sect. 4.2.1. [7]	Sect. 4.2.2. [21]		Sect. 5.2. [8]	

Figure 1. A guide to the problems discussed in the paper.

[1]The definition of equivalent behavior given above is not always appropriate in the case of multi-procedure programs. Alternative definitions are proposed in the sections that discuss these problems.

```
program
  P := 3.14
  rad := 3
  if DEBUG then rad := 4 fi
  area := P* (rad*rad)
  circ := 2*P*rad
  output(area)
  output(circ)
end
```

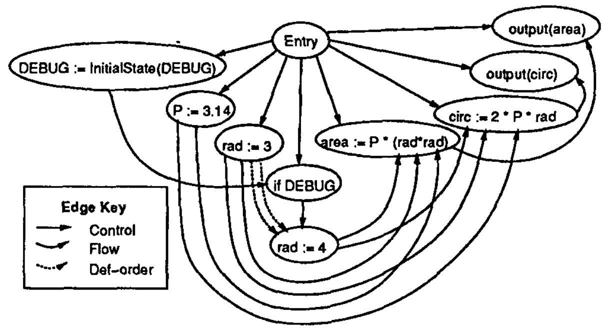


Figure 2. An example program and its program dependence graph.

ponent that uses the value of variable *x*, and there is an *x*-definition clear path from *v* to *w* in the program's control-flow graph.

Flow dependences are further classified as *loop independent* or *loop carried* [1]. A flow dependence from *v* to *w* is carried by the loop with predicate vertex *p* if both *v* and *w* are enclosed in the loop with predicate vertex *p*, and there is an *x*-definition clear path from *v* to *w* in the control-flow graph that includes a backedge to predicate vertex *p*. Loop-carried flow dependence edges are labeled with their carrying-loop-predicate vertex. A flow dependence from *v* to *w* is loop independent if there is an *x*-definition clear path from *v* to *w* in the control-flow graph that includes *no* backedge. It is possible to have several loop-carried flow edges between two vertices (each labeled with a different loop-predicate vertex); it is also possible to have both a loop-independent flow edge and one or more loop-carried flow edges between two vertices.

Def-order dependence edges are included in program dependence graphs to ensure that inequivalent programs cannot have isomorphic program dependence graphs [16]. A program dependence graph contains a def-order dependence edge from vertex *v* to vertex *w* with "witness" vertex *u* iff all of the following hold:

(1) Vertices *v* and *w* are both assignment statements that define the same variable.
(2) Vertices *v* and *w* are in the same branch of any conditional that encloses both of them.
(3) There is a flow dependence edge from *v* to *u*, and there is a flow dependence edge from *w* to *u*.
(4) The program component represented by *v* occurs before the program component represented by *w* in a preorder traversal of the program's abstract syntax tree.

A def-order edge is labeled with its "witness" vertex.

Note that a program dependence graph can be a multigraph (*i.e.*, can have more than one edge $v \rightarrow w$). In this case, the different edges from *v* to *w* are distinguished by their types (control, loop-independent flow, loop-carried flow, or def-order) and/or by their labels (the carrying loop-predicate vertex for loop-carried flow edges, and the "witness" vertex for def-order edges).

Example. Figure 2 shows a program that computes the circumference and the area of a circle, and the program's PDG. All edge labels have been omitted from the PDG. In this example all control dependence edges would be labeled **true**; one of the def-order edges between vertices "*rad* := 3" and "*rad* := 4" would be labeled with the name of the node that represents the assignment "*area* := *P** (*rad***rad*)," and the other would be labeled with the name of the node that represents the assignment "*circ* := 2**P***rad*."

2.2. System Dependence Graphs

The system dependence graph (SDG) extends the definition of program dependence graphs to accommodate collections of procedures (with procedure calls) rather than just single-procedure programs. Our definition of the SDG models a language with the following properties:

(1) A complete *system* consists of a single (main) procedure and a collection of auxiliary procedures.
(2) Parameters are passed by value-result.[4]

We make the further assumption that there are no calls of the form $P(x, x)$ or of the form $P(g)$, 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, we do not discuss global variables explicitly.

An SDG is made up of a collection of procedure dependence graphs, which are essentially the same as the program dependence graphs defined above, except that they may include additional vertices and edges to represent procedure calls. Procedure dependence graphs are connected by interprocedural control- and flow-dependence edges.

A procedure call is represented using a *call* vertex; parameter passing is represented using four kinds of *parameter* vertices: On the calling side, parameter passing is represented by two sets of vertices, called *actual-in* and *actual-out* vertices, which are control dependent on the call vertex; in the called procedure, parameter passing is represented by two sets of vertices, called *formal-in* and *formal-out* vertices, which are control dependent on the procedure's Entry vertex. Actual-in and formal-in vertices are included for every parameter; formal-out and actual-out vertices are included only for parameters that may be modified as a result of the call. (Interprocedural data-flow analysis can be used to determine which parameters may be modified as a result of a procedure call [5].)

In addition to control, flow, and def-order dependence edges, an SDG includes one new kind of intraprocedural edge called a *summary* edge. Summary edges, which connect some of the actual-in vertices at a call site with some of the actual-out vertices at the same call site, represent transitive dependences due to calls. Intuitively, there should be an edge from a vertex that represents a value being passed in to formal parameter *x* to a vertex that represents a value being copied back from formal parameter *y* if the initial value of *x* might be used to compute the final value of *y*. This property is, in general, undecidable; therefore, a safe approximation is computed as follows: a summary edge is added from actual-in vertex *v* to actual-out vertex *w* if there is an "executable" path in the system dependence graph from *v* to *w*; that is, a path that respects calling context by matching calls with returns. A polynomial-time algorithm that computes the set of summary edges is given in [20].

Connecting procedure dependence graphs to form a system dependence graph is straightforward, involving the addition of three kinds of interprocedural edges that represent direct dependences between a call site and the called procedure: (1) a *call* edge is added from each call vertex to the corresponding procedure-entry vertex; (2) a *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

[4]Techniques for handling parameters passed by reference and for dealing with aliasing are discussed in [20].

```
program                              procedure Mult3( op1, op2, op3, result )
  P := 3.14                            result := op1*op2*op3
  rad := 3                           end
  if DEBUG then rad := 4 fi
  call Mult3(P, rad, rad, area)
  call Mult3(2, P, rad, circ)
  output(area)
  output(circ)
end
```

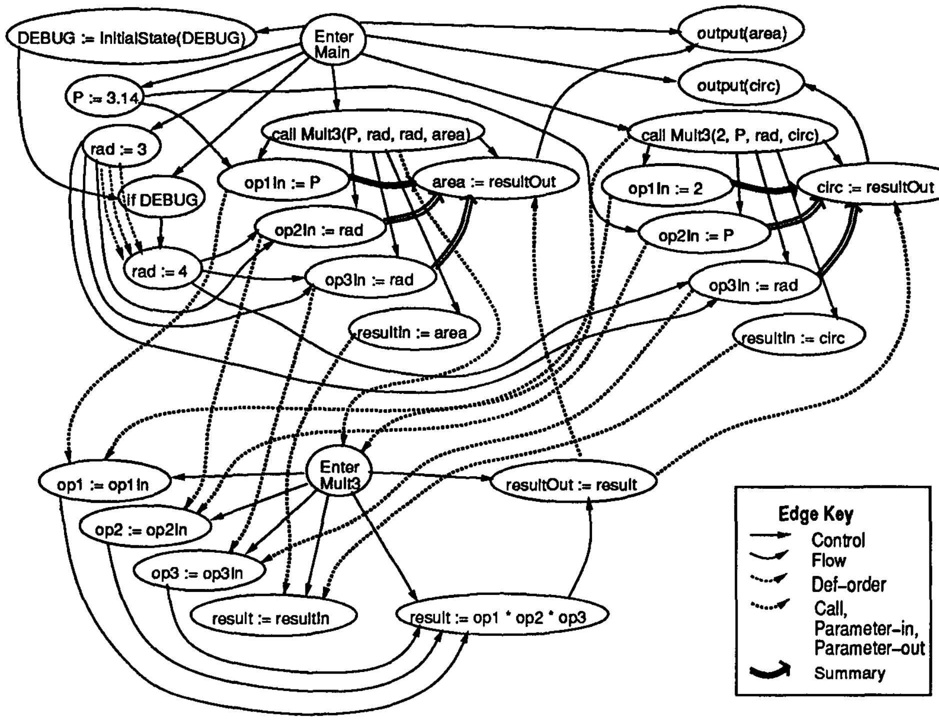


Figure 3. An example program and its system dependence graph.

Example. A multi-procedure version of the program from Figure 2 and the corresponding SDG are shown in Figure 3.

2.3. Program Representation Graphs

Program Representation Graphs (PRGs), like program dependence graphs, represent programs without procedures. PRGs are currently defined only for programs in the limited language used in Section 2.1, including scalar variables and assignment, if-then-else, output, and while statements. PRGs combine features of program dependence graphs and static single assignment forms [2,9,39,40]. The difference between a PDG and a PRG is that the latter includes special "ϕ vertices" as follows: One vertex labeled "ϕ_{if}: $x := x$" is added at the end of each *if* statement for each variable x that is defined within either (or both) branches of the *if* and is live at the end of the *if*; one vertex labeled "ϕ_{enter}: $x := x$" is added inside each *while* loop immediately before the loop predicate for each variable x that is defined within the *while* loop, and is live immediately after the loop predicate (*i.e.*, may be used before being redefined either inside the loop or after the loop); one vertex labeled "ϕ_{exit}: $x := x$" is added immediately after the loop for each variable x that is defined within the loop and is live after the loop.

The reason for adding the ϕ vertices is to ensure that every use of a variable (in an assignment statement, an output statement, or a predicate) is reached by exactly one definition. Because of the presence of ϕ vertices, def-order edges are not needed in PRGs.

Two advantages of using a PRG instead of a PDG are:

(1) A partitioning algorithm has been defined for PRGs to separate program components into equivalence classes based on the components' behaviors. While a slice isomorphism testing algorithm can be used to provide a similar partitioning using PDGs, the algorithm for PRGs is more powerful. (The partitioning algorithm and the slice isomorphism algorithm are discussed and compared in Section 4.1.2.)

(2) Two single-procedure program-integration algorithms have been defined; one uses PRGs while the other uses PDGs. The algorithm that uses PRGs is better able to accommodate semantics-preserving transformations than the algorithm that uses PDGs. (Both algorithms are discussed in Section 5.1.1.)

Example. Figure 4 shows the PRG of the program from Figure 2.

3. PROGRAM SLICING

In this section we discuss algorithms to solve the three slicing problems given in Section 1:

S.1. For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at q were different then the behavior at p would be different.

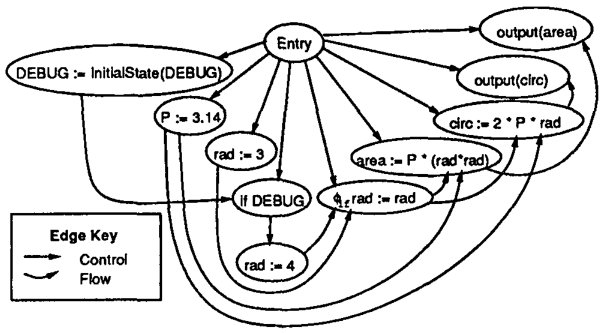


Figure 4. The Program Representation Graph for the program of Figure 2.

S.2. For a given program point p in program P, find [a superset of] the points q of P such that if the behavior at p were different then the behavior at q would be different.[5]

In other words, problem S.1 asks for the set of points that might *affect* the behavior at p, while problem S.2 asks for the set of points that might be *affected by* p.

S.3. For a given program point p in program P, find a projection Q of P such that when P and Q are run on the same initial state, the behaviors at p in P and Q are identical.

Problems S.1 and S.3 are closely related. Often, a solution to one provides a solution to the other, though we will see in Section 3.2 that this is not always true when dealing with multi-procedure programs. Nevertheless, we often blur the distinction between the two problems, referring to both of them as *backward slicing*. Problem S.2 is referred to as *forward slicing*.

Tools that compute backward and forward slices can help a programmer to understand what a program does and how it works by permitting him to focus his attention on meaningful subparts of the program. For example, if a programmer determines via testing that a variable x used at a point p in the program has the wrong value, it may be useful to examine the backward slice of the program with respect to point p. This slice will include all points involved in the computation of x's (erroneous) value, and will *exclude* points that are not relevant to the value of x at point p. Similarly, if a programmer intends to change a program, for example by deleting a statement p, it might be worthwhile first to examine the forward slice with respect to p to make sure that he understands all of the effects of p, both direct and transitive.

Additional uses for a solution to problem S.3 are discussed in Section 4.1.

Example. Figure 5(a) shows the program from Figure 2; components of the program that are in the backward slice with respect to statement "**output**(*circ*)" are shown enclosed in boxes. In Figure 5(b), the components of the program that are in the forward slice with respect to statement "$P := 3.14$" are shown enclosed in boxes.

Notice that the statement "**output**(*area*)" is not included in the backward slice shown in Figure 5(a). Therefore, executing the projection of the program that includes just the boxed components would produce different output than executing the original program. In particular, the projection would write a single value to the output (the value of *circ*) while the original program would

program
  $\boxed{P := 3.14}$
  $\boxed{rad := 3}$
  if $\boxed{DEBUG}$ **then** $\boxed{rad := 4}$ **fi**
  $area := P * (rad*rad)$
  $\boxed{circ := 2*P*rad}$
  output(*area*)
  $\boxed{\textbf{output}(circ)}$
end

(a) Backward slice from "**output**(*circ*)"

program
  $\boxed{P := 3.14}$
  $rad := 3$
  if $DEBUG$ **then** $rad := 4$ **fi**
  $\boxed{area := P * (rad*rad)}$
  $\boxed{circ := 2*P*rad}$
  $\boxed{\textbf{output}(area)}$
  $\boxed{\textbf{output}(circ)}$
end

(b) Forward slice from "$P := 3.14$"

Figure 5. (a) shows the backward slice of the program from Figure 2 with respect to statement "**output**(*circ*)"; (b) shows the forward slice with respect to statement "$P := 3.14$".

write two values (the value of *area* followed by the value of *circ*). This is consistent with our definition of behavior, which considers sequences of values produced at individual program points. However, it is also possible to model the idea that each output value has an effect on all future output. This can be accomplished by treating an output statement of the form "**output**(*exp*)" as if it were an assignment statement of the form "*output* := *output* || *exp*." (In essence, this chains a program's output statements together with flow dependences.)

The next two subsections discuss slicing algorithms for single- and multi-procedure programs, respectively.

3.1. Single-Procedure Slicing

Although Weiser's original slicing algorithm was expressed in terms of solving a sequence of dataflow-analysis problems [41], both backward and forward slices can be computed more efficiently using traversals of program dependence graphs [30].[6] The backward (forward) slice of program P with respect to point p can be computed in time linear in the size of the slice by following control and flow dependence edges backward (forward) in P's PDG, starting from the vertex that represents point p. The subgraph that includes the vertices reached during this traversal, together with the induced edge set, is called the *PDG slice*. A backward slice of P with respect to p is denoted by $b(P, p)$; a for-

[5] In both S.1 and S.2, when we say "if the behavior at a point were different" we mean "if the expression(s) used at that point were changed". For example, an assignment statement "$x := y + z$" can be changed to "$x := a - b$" but the left-hand-side variable x cannot be changed.

[6]Weiser's algorithm actually solves a problem that is a generalization of slicing problem S.3. Weiser viewed the behavior at a program point as the sequence of states that arise at that point, and his slicing problem was the following: For a given program point p and a set of variables V, find a projection Q of P such that when P and Q are run on the same initial state, the sequences of states that arise at point p in the two programs have identical values for all variables in V. Our goals are more limited; because of the definition we use for the execution behavior at a program point, in essence we limit set V to the variables used or defined at p.

Gallagher has developed a prototype software-maintenance tool to prevent editing changes from having undesired effects on the portions of the code that the user does not want to change [13]. Although his work uses slicing to address a problem of interest in software engineering, it is outside the scope of this paper because it relies on Weiser's notion of slices, which cannot be computed using dependence graphs.

ward slice of P with respect to p is denoted by $f(P, p)$.

The set of program points that correspond to the vertices of the backward PDG slice with respect to point p provides a solution to problem S.1; similarly, the set of program points that correspond to the vertices of the forward PDG slice with respect to p provides a solution to problem S.2. The program obtained by removing all statements and predicates that are not included in the backward PDG slice with respect to p provides a solution to problem S.3.

Example. The boxed components shown in Figure 5(a) correspond to the vertices reached by following control and flow dependences backward in the PDG shown in Figure 2, starting at the vertex labeled "**output**(*circ*)." Similarly, the boxed components shown in Figure 5(b) correspond to the vertices reached by following control and flow dependences forward in the PDG shown in Figure 2, starting at the vertex labeled "$P := 3.14$."

The correctness of our solutions to problems S.1, S.2, and S.3 follow from the Slicing Theorem and Corollary proved in [33]:

Theorem. (*Slicing Theorem*). *Let s be a PDG slice of program P and let Q be a program whose program dependence graph is isomorphic to s. If σ is a state on which P halts, then (1) Q halts on σ, and (2) corresponding points in s and Q compute the same sequence of values.*

Corollary. (*Slicing Corollary*). *Let* s_1 *and* s_2 *be PDG slices of programs* P_1 *and* P_2*, respectively, such that* s_1 *is isomorphic to* s_2*. Then, for any initial state σ on which both* P_1 *and* P_2 *halt, corresponding points in* s_1 *and* s_2 *compute the same sequence of values.*

3.2. Multi-Procedure Slicing

For programs *with* procedures, simply following control, flow, and interprocedural edges in the program's system dependence graph fails to take *calling context* into account. That is, a path along such edges may correspond to a procedure call being made at one call site, but returning to a different call site. Because of this failure to account for calling context, this slicing technique can include many extra program points in a slice, and is thus unsatisfactory.

Example. Figure 6 shows the slice that would be obtained by a backward traversal of the SDG of Figure 3 starting at the vertex that represents "**output**(*circ*)." The value of *circ* used in the output statement is computed in the second call to *Mult3*; thus the traversal (correctly) "descends" into the procedure dependence graph for *Mult3*. However, when the traversal reaches the *Entry* vertex for *Mult3*, it *incorrectly* "ascends" to *both* call sites. Thus, using this technique for computing slices, the final slice includes both calls to *Mult3*, although only the second call has an effect on the value of *circ* that is output.

A technique for computing slices using system dependence graphs that correctly handles the calling-context problem was defined in [20].[7] This technique involves *two* passes: Pass 1 of a slice (either backward or forward) with respect to point p, starts from the vertex that represents p; it moves "across" edges within a procedure (including stepping across procedure calls by following summary edges) and "up" from a called procedure to all corresponding call sites, but not "down" from a procedure to the procedures that it calls. This is accomplished in a backward slice by ignoring parameter-out edges, and in a forward slice by ignoring call and parameter-in edges. Pass 2 starts from the set of vertices reached during Pass 1; it follows edges "across" and "down" but not "up". Pass 2 of a backward slice ignores call and parameter-in edges; Pass 2 of a forward slice ignores parameter-out edges.

Example. Figure 7 shows the two passes of the backward slice of the SDG from Figure 3 with respect to "**output**(*circ*)." Figure 7(a) shows the vertices reached during Pass 1 as well as their induced edges; Figure 7(b) adds the vertices and edges included in the slice due to Pass 2. Figure 7(c) shows the components of the program that correspond to the vertices identified by the slice. Note that only the relevant call to *Mult3* is included in the slice.

We use the following notation to denote the separate passes of interprocedural slicing: a backward pass-1 slice of P with respect to p is denoted by $b1(P, p)$; a backward pass-2 slice of P with respect to p is denoted by $b2(P, p)$; a forward pass-1 slice of P with respect to p is denoted by $f1(P, p)$; a forward pass-2 slice of P with respect to p is denoted by $f2(P, p)$.

The vertices included in the two passes of an interprocedural slice provide a solution to problems S.1 and S.2. However, they do not always provide a solution to problem S.3. This is because the projection that includes just the program points that correspond to the vertices in the SDG slice may not be a syntactically correct program due to *parameter mismatch*. In particular, the slice may include two calls to the same procedure, with each call using a different subset of the procedure's parameters. The corresponding program projection is an incorrect program because the numbers of actual parameters at the call sites do not match the number of formal parameters in the procedure.

A solution to problem S.3 can be obtained from an SDG slice S by augmenting S as follows:

```
while S includes an instance of parameter mismatch do
  let v be an actual-in vertex for parameter a in a call to procedure P
      such that v is not in S, but there is another call to procedure P
      in S that does include an actual-in vertex for parameter a
  in augment S by adding the vertices and edges of the pass-2 back-
      ward slice starting at vertex v
od
```

That is, we augment the slice by including "missing" parameters and as much more of the program as is needed to compute their values.

4. PROGRAM DIFFERENCING

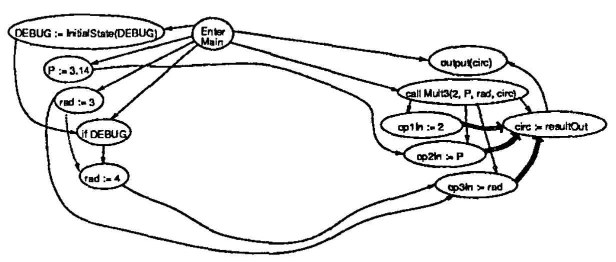
Programmers are often faced with the problem of finding the differences between two versions of a program. Text-based tools, such as the Unix utility *diff*, can be unsatisfactory because they only identify *textual* differences, providing little or no information about the *semantic* differences between the two versions. Although identifying semantic differences exactly is in general an undecidable problem, it is possible to devise algorithms that identify a safe approximation to (*i.e.*, a superset of) the semantic differences between two programs.

In this section, we discuss techniques for solving the three differencing problems introduced in Section 1:

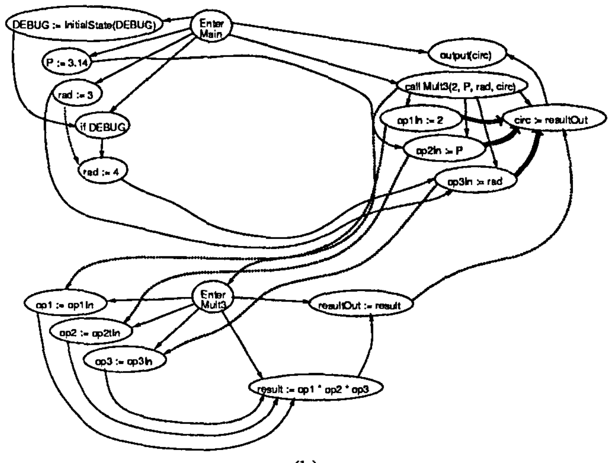
```
program
  \boxed{P := 3.14}
  \boxed{rad := 3}
  if \boxed{DEBUG} then \boxed{rad := 4 fi}
  \boxed{call Mult3}(\boxed{P},\boxed{rad},\boxed{rad},\boxed{area})
  \boxed{call Mult3}(\boxed{2},\boxed{P},\boxed{rad},\boxed{circ})
  output(area)
  \boxed{output(circ)}
end

procedure \boxed{Mult3}(\boxed{op1}, \boxed{op2}, \boxed{op3}, \boxed{result})
  \boxed{result:=op1*op2*op3}
end
```

Figure 6. The slice obtained by following edges backward in the SDG of Figure 3 starting from "**output**(*circ*)". This technique ignores calling context, and so includes *both* calls to *Mult3*.

[7]A different technique for interprocedural slicing has been given by Hwang *et al.* [22]. Their method also correctly handles the calling-context problem, but does not use SDGs. See [20] for a comparison of the two algorithms.



(a)



(b)

```
program
  P := 3.14
  rad := 3
  if DEBUG then rad := 4 fi
  call Mult3(P, rad, rad, area)
  call Mult3(2, P, rad, circ)
  output(area)
  output(circ)
end
procedure Mult3(op1, op2, op3, result)
  result := op1*op2*op3
end
```

(c)

Figure 7. A backward (interprocedural) slice of the SDG of Figure 3 with respect to component "**output**(*circ*)". The vertices reached during Pass 1 as well as their induced edges are shown in (a); the vertices and edges added by Pass 2 are shown in (b); the components of the program that correspond to the vertices identified by the slice are shown in (c).

D.1. Given programs *Old* and *New*, and a correspondence between their components, find [a superset of] the components of *New* whose behaviors differ from the corresponding components of *Old*.

D.2. Given programs *Old* and *New*, but with *no* correspondence between their components, find [a superset of] the components of *New* for which there is no component in *Old* with the same behavior.

D.3. Given programs *Old* and *New*, find a program *Changes* that exhibits all changed behaviors of *New* with respect to *Old*.

A solution to differencing problem D.3 is potentially useful in the context of program maintenance. For example, if program *New* is produced from program *Old* by fixing a bug, then rather than retesting all of *Old*'s functionality, only that portion represented in *Changes* must be retested, because the remainder of *New* is computationally equivalent to *Old*. This would be particularly beneficial if *Changes* were very small compared to *New*.

Solutions to problems D.1 and D.2 are useful for creating a tool that, given programs *Old* and *New*, annotates *New* to indicate its semantic and textual differences with respect to *Old*. A component of *New* is considered to represent a semantic difference either if there is no corresponding component in *Old* or if the corresponding component has a different behavior; a component of *New* is considered to represent a textual difference with respect to *Old* if there is a corresponding component of *Old* with the same behavior, but the text of the component has changed.

Example. Figure 8 shows program *Old* (the program from Figure 2) and two *New* programs. New_1 changes the name of variable *P* to *PI* and moves the assignment "*rad* := 3" inside the conditional; New_2 changes the initialization of variable *P*. The *New* programs are annotated to show their textual and semantic differences with respect to *Old*. Note that while a text-based differencing tool would have identified the same set of differences for program New_1, such a tool would have identified only the assignment "*P* := 3.1416" as a change in New_2.

In differencing problem D.1, we assume that we are furnished a way to identify corresponding components of *New* and *Old*. In practice, this correspondence could be established by using a special editor to create *New* from (a copy of) *Old*, where the editor keeps track of the "migration" of components as *New* is edited.[8] The correspondence could also be established by applying a *syntactic* matching algorithm to the two programs, such as that of [45].

The component-correspondence relation in differencing problem D.1 furnishes the means for establishing how program-dependence-graph vertices from different versions correspond. It is the component-correspondence relation that is used to determine "identical" vertices when operations are performed using vertices from different program dependence graphs. For instance, when we speak below of "identical slices," where the slices are actually taken in different graphs (*i.e.*, $b(G_{Old}, v) = b(G_{New}, v)$), we mean that the slices are isomorphic under the mapping provided by the component-correspondence relation.

4.1. Program Differencing for Single-Procedure Programs

This section discusses the three differencing problems D.1, D.2, and D.3 for single-procedure programs. In the single-procedure case, differencing problem D.3 can be solved by first applying an algorithm for either D.1 or D.2 (whichever is appropriate) and then applying an algorithm that solves slicing problem S.3; consequently, in this section we concentrate primarily on differencing problems D.1 and D.2.

[8]One way to track the migration of components is to have the editor attach tags to program components according to the following principles:

(1) When a copy of a program is made—*e.g.*, when a copy of *Old* is made in order to create *New*—each component in the copy is given the same tag as the corresponding component in the original program.

(2) The operations on program components supported by the editor are insert, delete, and move. A newly inserted component is given a previously unused tag; the tag of a component that is deleted is never reused; a component that is moved from one position to another retains its tag.

(3) The tags on components persist across different editing sessions and machines.

(4) Tags are allocated by a single server, so that two different editors cannot allocate the same new tag.

A tagging facility with such properties can be supported by language-based editors, such as those that can be created by such systems as MENTOR [10], GANDALF [29], and the Synthesizer Generator [32].

```
Old
program
  P := 3.14
  rad := 3
  if DEBUG then
    rad := 4
  fi
  area := P* (rad*rad)
  circ := 2*P*rad
  output(area)
  output(circ)
end
```

```
New1
program
  PI := 3.14                  ← TEXTUAL
  if DEBUG then
    rad := 4
  else
    rad := 3                  ← SEMANTIC
  fi
  area := PI* (rad*rad)       ← TEXTUAL
  circ := 2*PI*rad            ← TEXTUAL
  output(area)
  output(circ)
end
```

```
New2
program
  P := 3.1416                 ← SEMANTIC
  rad := 3
  if DEBUG then
    rad := 4
  fi
  area := P* (rad*rad)        ← SEMANTIC
  circ := 2*P*rad             ← SEMANTIC
  output(area)                ← SEMANTIC
  output(circ)                ← SEMANTIC
end
```

Figure 8. Program *Old* and two *New* programs. Each *New* program is annotated to show its differences with respect to *Old*.

4.1.1. D.1: Finding differences when given a correspondence between *Old* and *New*

When given a correspondence between the components of *Old* and *New*, it is possible to determine (a superset of) the components with different behaviors by comparing slices of *Old* and *New* with respect to corresponding points. Let G_{Old} and G_{New} denote the program dependence graphs for programs *Old* and *New*, respectively. If the slice of variant G_{New} at vertex v differs from the slice of G_{Old} at v, then, by the Slicing Theorem (Section 3.1), G_{New} and G_{Old} may compute a different sequence of values at v. In other words, vertex v is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of G_{New} with respect to G_{Old}, which we denote by $AP_{New,\,Old}$, as the subset of vertices of G_{New} whose slices in G_{Old} and G_{New} differ:

$$AP_{New,\,Old} =_{df} \{\, v \in V(G_{New}) \mid b(G_{Old}, v) \neq b(G_{New}, v) \,\}.$$

(By definition, the slice of a PDG with respect to a vertex not in its vertex set is the empty graph; therefore, any vertex of G_{New} that has no corresponding vertex in G_{Old} is included in $AP_{New,\,Old}$.) $AP_{New,\,Old}$ is a superset of the components of *New* with different behaviors in *New* than in *Old*.

The straightforward way to determine the members of $AP_{New,\,Old}$ (in quadratic time) would be to compare slices of G_{Old} and G_{New} with respect to every vertex of G_{New}. There is a more efficient way to determine $AP_{New,\,Old}$ by first finding the set of *directly* affected points of G_{New} with respect to G_{Old}. This set, denoted by $DAP_{New,\,Old}$, consists of those vertices v of G_{New} with *no* corresponding vertex in G_{Old} or with a different set of incoming edges than the corresponding vertex in G_{Old}.[9] $AP_{New,\,Old}$ can be determined from $DAP_{New,\,Old}$ by taking a forward slice of G_{New} with respect to $DAP_{New,\,Old}$.[10]

$$AP_{New,\,Old} = f\,(G_{New}, DAP_{New,\,Old}).$$

This method determines $AP_{New,\,Old}$ in time linear (rather than quadratic) in the size of G_{New}.

Remark. From the discussion of slicing problem S.3 in Section 3.1, it follows that the vertices of the slice $b(G_{New}, AP_{New,\,Old})$ define a projection of *New* that captures *New*'s changed behavior. This provides a solution to differencing problem D.3.

[9]For this comparison, a def-order edge is considered to be a ternary hyperedge whose target is the "witness" vertex.

[10]The PDG slicing operation defined in Section 3.1 with respect to a single vertex is easily extended to slicing with respect to a set of vertices: $b(G, (\bigcup_i s_i)) =_{df} \bigcup_i b(G, s_i)$ This operation can still be performed in time linear in the size of the (final) slice by performing a single traversal of the PDG starting from the set of points (rather than computing all of the individual slices and then taking the union of the resulting graphs).

4.1.2. D.2: Finding differences when not given a correspondence between *Old* and *New*

If one is not given a correspondence between the components of *Old* and *New*, there are two techniques that can be used to determine (a superset of) the components of *New* and *Old* that have different behaviors: *slice-isomorphism testing* and *partitioning*.

Slice-isomorphism testing

Two PDG slices are *isomorphic* if there is a 1-to-1 and onto map between their vertex sets that preserves adjacency and labels. Although there is no known polynomial-time graph-isomorphism algorithm for arbitrary unlabeled graphs [14], there are efficient algorithms for restricted classes of graphs, such as graphs of bounded valence [27]. Slice-isomorphism testing also concerns a restricted class of graphs: the vertices and edges in a program dependence graph are labeled, and the labeling, in conjunction with a graph's def-order edges, permits a vertex's incoming control and flow edges to be totally ordered. This property makes it possible to use depth-first search (traversing edges from targets to sources) to test whether two slices (possibly in different PDGs) are isomorphic. The running time of the algorithm is linear in the size of the smaller of the two slices being tested [21].

Because program points with isomorphic slices are guaranteed to have equivalent behavior, differencing problem D.2 can be solved by using the slice-isomorphism-testing algorithm of [21] to partition the components of *New* and *Old* into equivalence classes of isomorphic slices. The solution to D.2 is the set of components of *New* that are in a class that does not include any component of *Old*.

A naive partitioning technique based on slice-isomorphism testing would compare all pairs of slices from *New* and *Old*; in the worst case this would require time $O(n^3)$, where n is the size of the larger of the two program dependence graphs. A more efficient technique for performing such a partitioning would exploit the fact that a slice can be linearized in a canonical fashion. Given this insight, partitioning can be performed in time proportional to the sum of the sizes of all the slices in the programs, which is $O(n^2)$ in the worst case.

The key to this more efficient partitioning is to group isomorphic slices into equivalence classes, assigning each class a unique representative. Each vertex of the programs' graphs in turn is associated with the representative for its slice's isomorphism class; thus, two vertices have isomorphic slices iff they are associated with the same representative. The partitioning is performed as follows:

(1) A dictionary of linearized slices is maintained. Associated with each different slice is the unique representative for that equivalence class.

(2) For each program dependence graph G and each vertex v of G, the canonical linearization of slice $b(G, v)$ is computed. The linearized slice is looked up in the dictionary; if the slice is in the dictionary, the unique representative for that equivalence class is associated with vertex v; if the slice is not

in the dictionary, it is inserted along with a new unique representative.

Assuming that a lookup can be performed in time proportional to the size of the slice (*e.g.*, using hashing) the total time for the partitioning is proportional to the sum of the sizes of the programs' slices (*i.e.*, $O(n^2)$).

Partitioning

An alternative method for partitioning the components of *New* and *Old* into classes whose members have the same execution behavior is to use the Sequence-Congruence Algorithm described in [43, 44]. This algorithm applies to the PRGs (program representation graphs) of one or more programs. The algorithm partitions the vertices of the graph(s) so that two vertices are in the same partition only if the program components that they represent have equivalent behaviors. (In addition to the use described here for identifying the differences between two versions of a program, the Sequence-Congruence Algorithm can be used as a subroutine in an algorithm for integration problem I.1; this is discussed in Section 5.1.1.)

The Sequence-Congruence Algorithm uses a technique (which we will call the Basic Partitioning Algorithm) adapted from [2], which is based on an algorithm for minimizing a finite-state machine [15]. This technique finds the coarsest partition of a graph that is consistent with a given initial partition of the graph's vertices. The algorithm guarantees that two vertices v and v' are in the same class after partitioning iff they are in the same initial partition, and, for every predecessor u of v, there is a corresponding predecessor u' of v' such that u and u' are in the same class after partitioning.

The Sequence-Congruence Algorithm operates in two passes. Both passes use the Basic Partitioning Algorithm, but apply it to different initial partitions and make use of different sets of edges. An initial partition of the PRG is created based on the operators that are used in the vertices. The first pass refines this partition by applying the Basic Partitioning Algorithm to the flow-dependence subgraphs of the programs' PRGs. The second pass starts with the final partition produced by the first pass and refines it by applying the Basic Partitioning Algorithm to the control-dependence subgraphs of the program's PRGs. The time required by the Sequence-Congruence Algorithm is $O(n \log n)$, where n is the size of the programs' PRGs (*i.e.*, number of vertices + number of edges).

Example. If the Sequence-Congruence Algorithm were applied to the programs from Figure 8, the three instances of component "*rad* := 3" would be put into the same initial partition. However, the instance from New_1 would be separated from the other two instances by the algorithm's second pass because its control predecessor (the predicate of the *if* statement) is not in the same partition as the control predecessors of the instances from *Old* and New_2 (the *Entry* vertex). The fact that the component from New_1 is not in the same final partition as the other two components means that its behavior might be different than theirs. In this case its behavior is indeed different; the component in New_1 only executes if variable *DEBUG* is *false*, while the components in *Old* and New_2 always execute.

Although the instances of component "*rad* := 3" from *Old* and New_1 are in different final partitions, the two instances of "**output**(*area*)" from *Old* and New_1, which are (transitively) flow dependent on the instances of "*rad* := 3", are in the *same* final partition (which is appropriate since they have equivalent behaviors).

Comparison of the two methods

It is instructive to compare the two methods that have been described above.

(1) The Sequence-Congruence Algorithm is more powerful than the technique based on slice-isomorphism testing. For instance, for the example programs shown in Figure 8, the Sequence-Congruence Algorithm is able to determine that statement **output**(*area*) has the same behavior in *Old* and New_1. The technique based on slice-isomorphism testing places the two instances of statement **output**(*area*) from *Old* and New_1 in different slice-isomorphism classes, thus indicating that the behavior of this statement is potentially different in the two programs.

(2) The Sequence-Congruence Algorithm is more efficient than the technique based on slice-isomorphism testing (*i.e.*, $O(n \log n)$ versus expected $O(n^2)$).

(3) The technique based on slice-isomorphism testing has the advantage of being extendible to handle programs with multiple procedures, as described in Section 4.2.2. It might also be possible to extend the Sequence-Congruence Algorithm to handle procedures, however no such extension is currently known.

4.2. Program Differencing for Multi-Procedure Programs

The criterion for when two program points have equivalent behavior that we have used so far (*i.e.*, two points have equivalent behavior if they produce the same sequences of values when their respective programs are executed on the same initial state) is not appropriate for multi-procedure program differencing. To understand this, consider creating program *New* by making a copy of the program of Figure 3, and reversing the order of the two calls to *Mult3*. If program points that produce different sequences of values are considered to have inequivalent behavior, then the behavior of the statement "*result* := *op* 1**op* 2**op* 3" (in procedure *Mult3*) in *Old* differs from its behavior in *New*. (Because the order of the two calls is reversed, the order of the two elements in the sequence would also be reversed.) However, because the two calls to *Mult3* are part of independent computations, reversing their order would normally be considered to be a semantics-preserving transformation, and one would not expect *any* component of *New* to be classified as a semantic change with respect to *Old*.

There are several different criteria for equivalent behavior that one might use in the context of multi-procedure differencing. In this section, we use the following criterion: Two program points have equivalent behavior if, when their respective procedures are called with the same actual parameters, the points produce the same sequence of values at "top-level"; that is, if the procedures are recursive, we are concerned only with the sequences of values produced in the outermost invocations.

Under this criterion, reversing the order of the two calls to *Mult3* does not change the behavior of statement "*result* := *op* 1**op* 2**op* 3" (as desired). In general, to produce a semantic difference in a procedure Q under this criterion for equivalent behavior, it is necessary to change a component of Q itself, or a component of a procedure called (directly or transitively) by Q.

It is certainly possible to use other criteria for behavioral equivalence as the basis for a program-differencing tool. (One alternative, in which the criterion for equivalent behavior is not restricted to only "top-level" behavior, is explored in [7].) However, for this paper we concentrate on the simpler criterion for equivalent behavior given above; this has the advantage that it simplifies the discussion considerably, yet still gives the flavor of the issues that must be addressed in algorithms for semantics-based program differencing. In practice, it would probably be useful for a differencing tool to support multiple criteria (with the choice of which criterion to apply under the control of the user).

4.2.1. D.1: Finding differences when given a correspondence between *Old* and *New*

In this section, we assume that a correspondence between the components of two multi-procedure programs *Old* and *New* has been provided. As with single-procedure programs, it is possible to determine (a superset of) the components with different behaviors by comparing slices of *Old* and *New* with respect to corresponding points. However, under the new criterion for equivalent behavior specified above, it is only necessary to compare *b2* slices of corresponding components, rather than full backward slices. (Recall from Section 3.2 that a *b2* slice is obtained using just the second pass of the interprocedural slicing algorithm. As explained in Section 3.2, a *b2* slice taken with respect to a given component *c* in procedure P follows edges within P and in procedures called from P (directly or transitively), but does *not* follow call or parameter-in edges "up" to calling procedures.)

The significance of a *b2* slice for program differencing stems from the following proposition:

Proposition. *Suppose $b2(S_1, p_1)$ is isomorphic to $b2(S_2, p_2)$. Let P_1 and P_2 be the procedures in S_1 and S_2 that contain p_1 and p_2, respectively. Suppose P_1 and P_2 are called with the same actual parameters and suppose both computations terminate normally. Then the sequence of values produced at p_1 and p_2 in the outermost invocations of P_1 and P_2 are equal.*

In other words, when two procedure components have isomorphic *b2* slices they are guaranteed to produce the same sequence of values at "top-level" whenever their respective procedures are called with the same actual parameters.

Given two SDGs S_{New} and S_{Old}, the directly affected points of S_{New} with respect to S_{Old}, denoted by $DAP_{New, Old}$, is defined similarly to way directly affected points were defined for single-procedure programs in Section 4.1. $DAP_{New, Old}$ consists of vertices of S_{New} for which either there exists no corresponding vertex in S_{Old} or whose set of incoming edges is different from the set of incoming edges at the corresponding vertex in S_{Old}. However, for the purposes of identifying directly affected points in SDGs, when the sets of incoming edges at corresponding vertices of S_{New} and S_{Old} are compared, all incoming summary, parameter-in, and call edges are ignored. Thus, only when there are differences in the incoming control, flow, def-order, or parameter-out edges is a vertex of S_{New} considered to be directly affected.

Because summary edges represent transitive dependences rather than direct dependences, changes in the set of incoming summary edges do not represent direct modifications to the system; therefore, they are ignored for purposes of classifying vertices as directly affected points.

The reason parameter-in and call edges are ignored is related to the fact that these edges are ignored (*i.e.*, not traversed) during a *b2* slice. Because parameter-in and call edges are also ignored during an *f1* slice, we can determine the set of all vertices of S_{New} with different *b2* slices by taking an *f1* slice with respect to $DAP_{New, Old}$. In other words,

$$\{ v \in V(S_{New}) \mid b2(S_{New}, v) \neq b2(S_{Old}, v) \} = f1(S_{New}, DAP_{New, Old}).$$

4.2.2. D.2: Finding differences when not given a correspondence between *Old* and *New*

If one is not given a correspondence between the components of two multi-procedure programs *Old* and *New*, it is still possible to identify procedure components with equivalent execution behaviors by comparing *b2* slices of the programs' SDGs. Although SDGs include some additional kinds of vertices and edges that are not found in PDGs, the slice-isomorphism testing algorithm for single-procedure programs can be extended to test *b2* slices for isomorphism: For each vertex in a *b2* slice, the vertex's incoming edges can be given a canonical total ordering, which is the property that makes it possible to use depth-first search (traversing edges from targets to sources) to test whether two slices (possibly in different SDGs) are isomorphic. Thus, it is possible to test whether two *b2* slices are isomorphic in time linear in the size of the smaller of the two slices being tested, and it is possible to partition the components of one or more procedures into equivalence classes in time proportional to the sum of the sizes of their *b2* slices [21].

As discussed in Section 4.2.1, when two procedure components have isomorphic *b2* slices they are guaranteed to produce the same sequence of values at "top-level" whenever their respective procedures are called with the same actual parameters. Thus, one way to solve differencing problem D.2. is to perform the following test:

For all points p_1 in S_{New} and p_2 in S_{Old}, test whether $b2(S_{New}, p_1)$ is isomorphic to $b2(S_{Old}, p_2)$.

There are two reasons why it is better to take a different approach in an actual tool for program differencing:

(i) Efficiency: the cost of the above method is proportional to the sum of the sizes of the programs' *b2* slices (*i.e.*, quadratic in the size of the larger of the two SDGs).

(ii) Usefulness: in most cases, one is interested in determining whether the components in a given procedure P_1 from S_{New} have different execution behavior from the components in some other given procedure, say P_2 from S_{Old}. Ordinarily, it is not important to know whether the various components in P_1 exhibit behavior that is equivalent to that exhibited at a point *somewhere* in S_{Old} (*i.e.*, not necessarily restricted to one procedure P_2).

Consequently, we believe that it is more useful to have the user of a differencing tool supply the names of two procedures, P_1 from S_{New} and P_2 from S_{Old}, in which components are to be tested for equivalent behavior. (The facility for finding differences in two multi-procedure programs implemented in our prototype system takes this form.)

5. PROGRAM INTEGRATION

In this section, we discuss techniques to solve the two integration problems introduced in Section 1:

I.1. Given a program *Base* and two or more variants, together with a correspondence between the programs' components, determine whether the changes made to create the variants from *Base* are compatible. If the changes are compatible, combine the variants to form a single, integrated program; otherwise identify the source of the interference.

I.2. Same problem as I.1, but with *no* correspondence between the components of *Base* and the variants.

The need to integrate several versions of a program into a common one arises in many situations:

(1) A system may be customized by a user while simultaneously being upgraded by a maintainer. When the next release of the system is sent to the user, he must integrate his customized version of the system and the newly released version with respect to the earlier release so as to incorporate both his customizations and the upgrades.

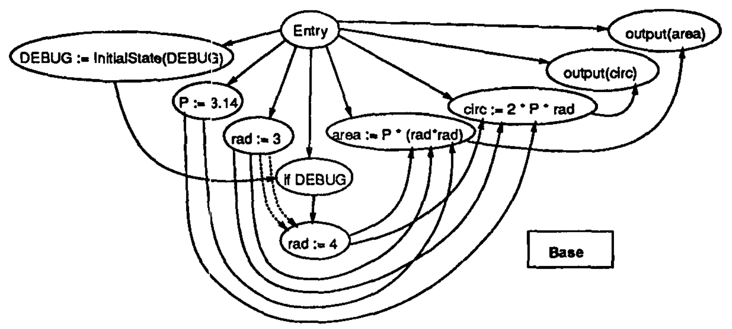
(2) While systems are being created, program development is often a cooperative activity that involves multiple programmers. If a task can be decomposed into independent pieces, the different aspects of the task can be developed and tested independently by different programmers. However, if such a decomposition is not possible, the members of the programming team must work with multiple, separate copies of the source files, and the different versions of the files must ultimately be integrated to produce a common version.

(3) Suppose a tree or dag of related versions of a program exists (to support different machines or different operating systems, for instance), and the goal is to make the same enhancement or bug-fix to all of them. If, for example, the change is made to the root version—by manually modifying a copy of the root program—the process of installing the change in all other versions requires a succession of program integrations.

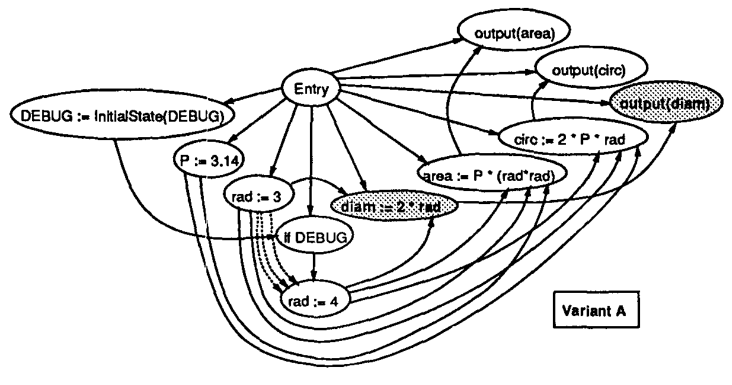
Anyone who has had to reconcile divergent lines of development will recognize that it is a tedious and time consuming task to merge programs by hand and will appreciate the need for automatic assistance.

At present, the only available tools for integration implement an operation for merging files as strings of text (the UNIX *diff3* utility is one example). This approach has the advantage that the current tools are as applicable to merging documents, data files, and other text objects as they are to merging programs. However, current tools are necessarily of limited effectiveness for integrating *programs*—and are even dangerous—because the manner in which two programs are merged is not *safe*; one has no guarantees about the way the program that results from a purely *textual* merge behaves in relation to the behaviors of the programs that are the arguments to the merge. For example, if one variant contains changes only on lines 5–10, while the other variant contains

```
program
  P := 3.14
  rad := 3
  if DEBUG then rad := 4 fi
  area := P* (rad*rad)
  circ := 2*P*rad
  output(area)
  output(circ)
end
```



```
program
  P := 3.14
  rad := 3
  if DEBUG then rad := 4 fi
  diam := 2*P
  area := P* (rad*rad)
  circ := 2*P*rad
  output(diam)
  output(area)
  output(circ)
end
```



```
program
  P := 3.1416
  rad := 3
  if DEBUG then rad := 4 fi
  area := P* (rad*rad)
  circ := 2*P*rad
  output(area)
  output(circ)
end
```

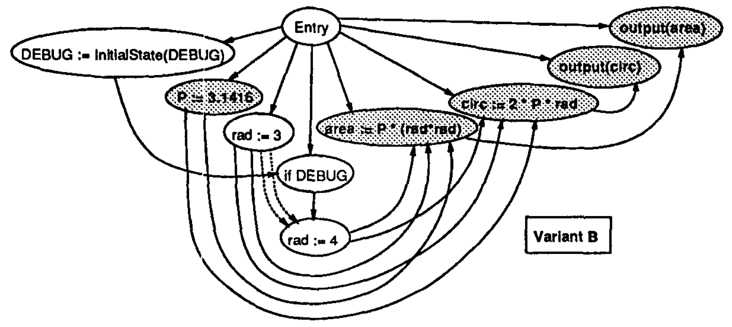


Figure 9. An example of the first part of Step 1 of the HPR Algorithm. Programs *Base*, *A*, and *B*, and their PDGs are shown. Shading is used to indicate the affected points of *A* and *B* with respect to *Base*.

changes only on lines 15–20, *diff3* would deem these changes to be interference-free; however, just because changes are made at different places in a program is no reason to believe that the changes are free of undesirable interactions. The merged program produced by such a tool must, therefore, be checked carefully for conflicts that might have been introduced by the merge.[11]

There has been related work on integrating functions [6], logic programs [25], and specifications [11]; however, different models of "integration" have been used in each case. For example, in Berzins's work on integrating functions [6], two variant programs, *A* and *B*, are merged without regard to *Base*. Thus, his work treats only the integration of program *extensions*, not program *modifications*, where the distinction between the two is as follows:

> A program extension extends the domain of a partial function without altering any of the initially defined values, while a modification redefines values that were defined initially [6].

The function that results from the merge preserves the (entire)

[11]Several managers in industry have told us that their mechanism to avoid integration conflicts is based on the modular structure of systems. They assign overall responsibility for a given module of a system to a particular programmer, and institute a policy that any changes to a module must be cleared with the person responsible. However, this assumption is true only if all changes preserve the module's semantics (*i.e.*, they change the *way* the module performs its task without changing its functionality). If modifications do not preserve the module's semantics, the notion that module boundaries protect against interference—even in conjunction with the above policy—is as flawed as the notion used in *diff3*. Both rely on the incorrect assumption that "disjoint changes are interference free."

behavior of *both*; functions A and B cannot be merged if they conflict at any point where both are defined.

The 1–1 join operation defined by Lakhotia and Sterling is also a two-way merge [25]. However, in Lakhotia and Sterling's work there is no notion of interference, and the characterization of the semantic properties of the merged program was left as an open question in [25]. Feather's work on integrating specifications [11] does take *Base* into account, but although his integration algorithm preserves syntactic modifications, it does not guarantee any semantic properties of the integrated specification.

The alternative that we have pursued is to create a semantics-based tool for program integration that makes use of knowledge of the programming language to determine whether the changes made to *Base* to produce variants A and B have undesirable semantic interactions. Only if there is no such interference should the tool produce a merged program.

In version I.1 of the integration problem, we assume that we are furnished a way to identify corresponding components of *Base*, A, and B. As explained in Section 4, this correspondence could be established either by using a special editor to create the variants from (copies of) *Base*, where the editor keeps track of the "migration" of components as variants A and B are edited, or by first applying a *syntactic* matching algorithm to the three programs, such as that described in [45].

5.1. Program Integration for Single-Procedure Programs

While our long-term goal is to design a semantics-based program-integration tool for a full-fledged programming language, our early work used a simplified model of program integration so as to make the problem amenable to theoretical study [18]. This model possesses the essential features of the problem, and thus permitted us to conduct our studies without being overwhelmed by inessential details.

The original model of semantics-based program integration has the following three requirements:

(1) Programs are written in a simplified programming language that has only assignment statements, output statements, if-then-else statements, and while loops. The language does not include input statements; however, a program can use a variable before assigning to it, in which case the variable's value comes from the initial state.

(2) When an integration algorithm is applied to base program *Base* and variant programs A and B, and if integration succeeds—producing program M—then for any initial state σ on which *Base*, A, and B all terminate normally, M must have the following properties:
   (i) M terminates normally on σ.
   (ii) For any component c in variant A whose behavior is different in A and *Base*, component c is in M and its behavior in M agrees with component c in A.[12]
   (iii) For any component c in variant B whose behavior is different in B and *Base*, component c is in M and its behavior in M agrees with component c in B.
   (iv) For any program component c that has the same behavior in *Base*, A, and B, component c is in M and also exhibits that same behavior.

(3) Program M is to be created only from components that occur in programs *Base*, A, and B.

An informal statement of property (2) is: changes in the behavior of A and B with respect to *Base* must be preserved in the integrated program, along with the behavior that is the same in *Base*, A, and B.

Properties (1) and (3) of the integration model are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. *Any* program M that satisfies Properties (1), (2), and (3) integrates *Base*, A, and B; if no such program exists then A and B interfere with respect to *Base*. However, Property (2) is not decidable, even under the restrictions given by Properties (1) and (3); consequently, any program-integration algorithm will sometimes report interference—and thus fail to produce an integrated program—even though there is actually *no* interference (*i.e.*, even when there is *some* program that meets the criteria given above).

[12] For the purposes of this section we use the definition of behavior from Section 1, namely the sequence of values produced at a program point.

5.1.1. I.1: Program integration given a correspondence among components of *Base*, *A*, and *B*

The first algorithm that meets the requirements of the model given above was formulated by Horwitz, Prins, and Reps [18]. That algorithm—which we call the *HPR algorithm*—is the first algorithm for semantics-based program integration. The HPR algorithm represents a fundamental advance over text-based program-integration algorithms (such as the UNIX utility *diff3*) by providing the first theoretical foundation for building a semantics-based program-integration tool. Changes in *behavior* rather than changes in text are detected, and are preserved in the integrated program. Although in general it is undecidable to determine exactly the set of components with changed behavior, the HPR algorithm computes a safe approximation to this set by using slicing and differencing operations on the program dependence graphs of programs *Base*, A, and B.

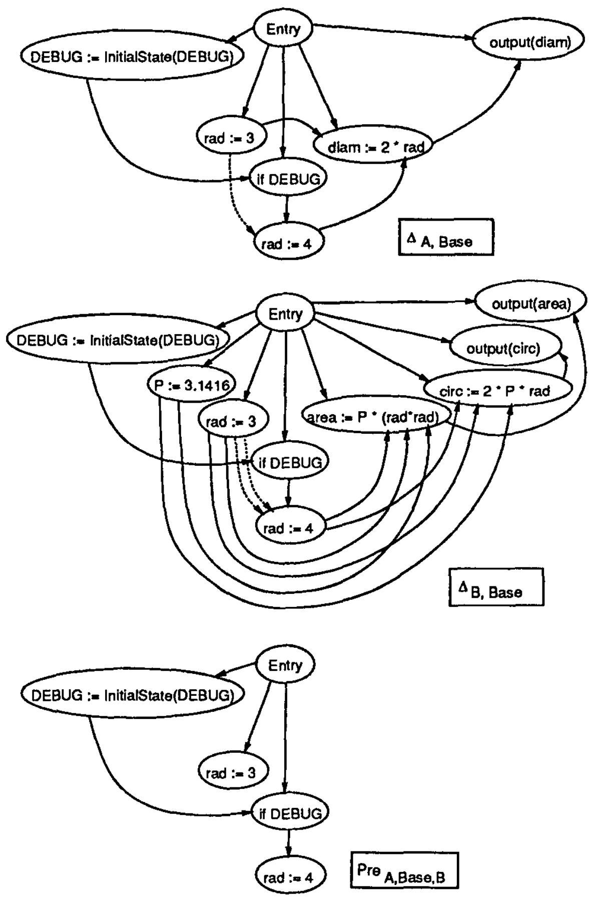


Figure 10. Step 1 of the HPR algorithm, continued. The subgraphs $\Delta_{A,\,Base}$, $\Delta_{B,\,Base}$, and $Pre_{A,\,Base,\,B}$ are shown.

Given PDGs G_{Base}, G_A, and G_B (for programs *Base*, *A*, and *B*, respectively), the HPR algorithm performs three steps. The first step identifies three subgraphs that represent the changed behavior of *A* with respect to *Base*, the changed behavior of *B* with respect to *Base*, and the behavior that is the same in all three programs. The second step combines these subgraphs to form a merged dependence graph *M*. The third step determines whether *A* and *B* interfere with respect to *Base*; if there is no interference, an integrated program is produced from graph *M*.

Step 1: Determining changed and preserved slices

The *affected points* of variants *A* and *B* with respect to *Base* (as defined in Section 4.1.1) are used to identify subgraphs $\Delta_{A,\,Base}$ and $\Delta_{B,\,Base}$ of G_A and G_B that represent the variants' changed behaviors with respect to *Base*:

$$AP_{A,\,Base} =_{df} \{\, v \in V(G_A) \mid b(G_{Base}, v) \neq b(G_A, v) \,\}$$
$$AP_{B,\,Base} =_{df} \{\, v \in V(G_B) \mid b(G_{Base}, v) \neq b(G_B, v) \,\}$$

$$\Delta_{A,\,Base} =_{df} b(G_A, AP_{A,\,Base})$$
$$\Delta_{B,\,Base} =_{df} b(G_B, AP_{B,\,Base}).$$

A vertex that has the same slice in all three programs is guaranteed to exhibit the same behavior. Thus, we define the *preserved points* of G_{Base} (with respect to G_A and G_B), denoted by $PP_{A,\,Base,\,B}$, to be the subset of vertices of G_{Base} with identical slices in G_{Base}, G_A, and G_B:

$$PP_{A,\,Base,\,B} =_{df} \{\, v \in V(G_{Base}) \mid b(G_A, v) = b(G_{Base}, v) = b(G_B, v) \,\}.$$

Thus, the slices common to *Base*, *A*, and *B* are captured by the slice $b(G_{Base}, PP_{A,\,Base,\,B})$, and so we define $Pre_{A,\,Base,\,B}$—the projection of *Base* that exhibits behavior common to all three programs—as follows:

$$Pre_{A,\,Base,\,B} =_{df} b(G_{Base}, PP_{A,\,Base,\,B}).$$

Example. Figure 9 shows three programs: *Base*, *A*, and *B*, and their PDGs. The affected points of G_A and G_B are shown as shaded vertices. (*Base* is the program from Figure 2 that computes a circle's area and circumference; variant *A* adds the computation of the diameter; variant *B* changes the initialization of variable *P*.) Figure 10 shows the subgraphs that represent $\Delta_{A,\,Base}$, $\Delta_{B,\,Base}$, and $Pre_{A,\,Base,\,B}$.

Step 2: Forming the merged graph

The merged graph G_M is formed by taking the graph union of the slices that characterize the changed behavior of *A*, the changed behavior of *B*, and the behavior of *Base* preserved in both *A* and *B*. The given component-correspondence relation is used to determine which vertices and edges from the three slices are "identical" for the purposes of the union; that is, if vertex *v* (or edge *e*) occurs in one slice, and a corresponding vertex (edge) occurs in another slice, only one copy of the vertex (edge) is included in the merged graph.

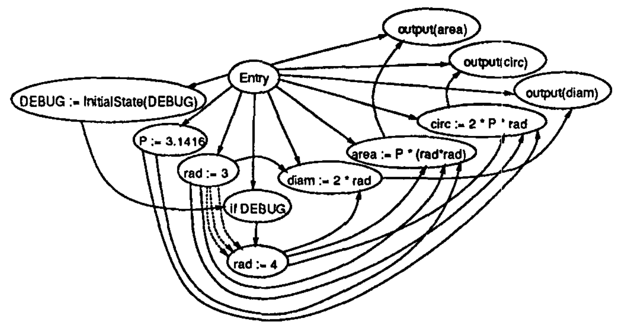


Figure 11. The merged graph: $\Delta_{A,\,Base} \cup_g \Delta_{B,\,Base} \cup_g Pre_{A,\,Base,\,B}$.

$$G_M =_{df} \Delta_{A,\,Base} \cup_g \Delta_{B,\,Base} \cup_g Pre_{A,\,Base,\,B}.$$

Example. Figure 11 shows the merged graph for the example integration problem.

Step 3: Testing for interference

There are two possible ways in which the graph G_M can fail to represent a satisfactory integrated program; we refer to them as "Type I interference" and "Type II interference." The criterion for Type I interference is based on a comparison of slices of G_A, G_B, and G_M. The slices $b(G_A, AP_{A,\,Base})$ and $b(G_B, AP_{B,\,Base})$ represent the changed behaviors of *A* and *B*, respectively. There is Type I interference if G_M does not preserve these slices; that is, there is Type I interference if either

$$b(G_M, AP_{A,\,Base}) \neq b(G_A, AP_{A,\,Base}) \quad \text{or}$$
$$b(G_M, AP_{B,\,Base}) \neq b(G_B, AP_{B,\,Base}).$$

The final step of the HPR algorithm involves reconstituting a program from the merged program dependence graph. However, it is possible that there is no such program—the merged graph can be an *infeasible* program dependence graph; this is Type II interference. (See [18] or [3] for a discussion of reconstructing a program from the merged program dependence graph and the inherent difficulties of this problem.) If neither kind of interference occurs, one of the programs that corresponds to the graph G_M is returned as the result of the integration operation.

Example. The example integration problem has neither Type I nor Type II interference. One of the programs that corresponds to the merged graph is shown in Figure 12.

One of the most important aspects of the HPR algorithm is that it provides *semantic* guarantees about how the behavior of the integrated program relates to the behaviors of *Base*, *A*, and *B*. In particular, it has been shown that the HPR algorithm meets the semantic criterion for program integration given at the beginning of Section 5.1 [33,44].

Accommodating Semantics-Preserving Transformations

One limitation of the HPR algorithm is that it incorporates no notion of a semantics-preserving transformation. This limitation causes the algorithm to be overly conservative in its definition of interference. For example, if one variant changes the *way* a computation is performed (without changing the values computed) while the other variant adds code that uses the result of the computation, the HPR algorithm would classify those changes as interfering. This section discusses a different integration algorithm—developed by Yang, Horwitz, and Reps, and called the YHR algorithm—that is an improvement on the HPR algorithm in that it is capable of accommodating semantics-preserving transformations.

Example. Figure 13 shows two example integration problems that illustrate the limitations of the HPR algorithm with regard to semantics-preserving transformations. The HPR algorithm will

```
program
  P := 3.1416
  rad := 3
  if DEBUG then rad := 4 fi
  diam := 2*P
  area := P*(rad*rad)
  circ := 2*P*rad
  output(diam)
  output(area)
  output(circ)
end
```

Figure 12. The result of integrating programs *Base*, *A*, and *B* from Figure 9.

Base	Variant *A*	Variant *B*	Integrated Program

program
P := 3.14
rad := 3
if *DEBUG*
  then *rad* := 4
fi
area := *P**(*rad***rad*)
output(*area*)
	program
	$\boxed{PI := 3.14}$
	if *DEBUG*
	  then *rad* := 4
	  else $\boxed{rad := 3}$
	fi
	$\boxed{area := PI*(rad*rad)}$
	output(*area*)
		program
		P := 3.14
		rad := 3
		if *DEBUG*
		  then *rad* := 4
		fi
		area := *P**(*rad***rad*)
		$\boxed{height := 10}$
		$\boxed{vol := height*area}$
		output(*area*)
		$\boxed{\textbf{output}(vol)}$
			program
			PI := 3.14
			if *DEBUG*
			  then *rad* := 4
			  else *rad* := 3
			fi
			area := *PI**(*rad***rad*)
			height := 10
			vol := *height***area*
			output(*area*)
			output(*vol*)

program
k := 0; *i* := 1
while *i*≤100 **do**
  j := *i**2
  while *j*<1000 **do**
    k := *k*+*i**10+*j*
    j := *j*+1
  od
  i := *i*+1
od
output(*k*)
	program
	k := 0; *i* := 1
	$\boxed{twoi := 2}$
	while *i*≤100 **do**
	  $\boxed{j := twoi}$
	  while *j*<1000 **do**
	    k := *k*+*i**10+*j*
	    j := *j*+1
	  od
	  $\boxed{twoi := twoi+2}$
	  i := *i*+1
	od
	output(*k*)
		program
		k := 0; *i* := 1
		while *i*≤100 **do**
		  j := *i**2
		  $\boxed{teni := i*10}$
		  while *j*<1000 **do**
		    $\boxed{k := k+teni+j}$
		    j := *j*+1
		  od
		  i := *i*+1
		od
		output(*k*)
			program
			k := 0; *i* := 1
			twoi := 2
			while *i*≤100 **do**
			  j := *twoi*
			  teni := *i**10
			  while *j*<1000 **do**
			    k := *k*+*teni*+*j*
			    j := *j*+1
			  od
			  twoi := *twoi*+2
			  i := *i*+1
			od
			output(*k*)

Figure 13. Two example integration problems that illustrate the limitations of the HPR algorithm with regard to semantics-preserving transformations. Modifications in variants *A* and *B* are enclosed in boxes. In both cases the HPR algorithm would report interference even though there is no interference according to the integration criteria stated in Section 5.1. An integrated program that satisfies the criteria is shown in each case.

report interference in both cases; however, there is *no* interference according to the integration criteria given in Section 5.1, and an integrated program that satisfies the criteria is shown in each case. In the first example, variant *A* changes the computation of *area* by renaming variable *P* to *PI*, and moving the assignment *rad* := 3 inside the conditional, while variant *B* adds an assignment to variable *vol* that uses the value of *area*. In the second example, variants *A* and *B* both perform semantics-preserving transformations: variant *A* changes the way *j* is computed by performing a reduction in strength (replacing a multiplication with an addition), while variant *B* changes the assignment to *k* (which uses the value of *j*) by moving a loop-invariant computation outside the loop.

The limitations of the HPR algorithm illustrated in Figure 13 are due to the way the HPR algorithm identifies affected points, and to the way program fragments are extracted from *Base*, *A*, and *B*, and combined to form the merged program. For example, consider the first integration problem of Figure 13. The change made to *Base* to create variant *B* was the addition of the computation of *vol*. This new code must be included in the merged program; however, the HPR algorithm would extract from variant *B* the *entire* program fragment needed to compute the value of *vol* (*i.e.*, the entire slice of the program with respect to the new components). This fragment includes the statement "*area* := *P** (*rad***rad*)", which is undesirable since the way *area* is computed (though not its value) has been changed in variant *A* (by renaming *P* to *PI* and by moving one assignment to *rad* inside the conditional). It would be preferable to extract from variant *B* only the assignments to *height* and *vol*, combining this fragment with the changed fragment from *A*. However, to do this requires being able to recognize that the value of *area* is the same in variant *A* as in *Base*, which the HPR algorithm is unable to do.

The YHR algorithm is an integration algorithm designed to overcome these limitations of the HPR algorithm. The important differences between the two algorithms are the following:

(1) The HPR algorithm uses comparison of slices to identify the affected points of variants *A* and *B* with respect to *Base*. In contrast, the YHR algorithm is parameterized in terms of a method for identifying *congruent* components of *Base*, *A*, and *B*. (A precise definition of congruence can be found in [46]; roughly, two components are congruent if they and all of their corresponding control dependence ancestors have equivalent behavior.) The partitioning algorithm discussed in Section 4.1.2 is one method that can be used to identify congruent components. Congruent components could also be identified by the programmer, or by the editor front-end of a program-transformation system.

(2) The HPR algorithm includes the slice with respect to every affected point in the merged graph. In contrast, the YHR algorithm uses a new operation, called *limited slicing*, to extract only partial slices with respect to affected points.

(3) The HPR algorithm uses program dependence graphs to represent *Base*, *A*, and *B*. In contrast, the YHR algorithm uses program representation graphs. The ϕ vertices in program representation graphs are crucial to the success of the YHR algorithm. Suppose *Base* includes a loop or conditional in which the value of some variable *x* is computed, and variant *A* changes the way *x* is computed, without changing the final value of *x*. Further suppose that *B* adds a new component that uses the final value of *x*. The ϕ-exit or ϕ-if vertex "*x* := *x*" at the end of the code that computes *x* is a point with identical behaviors in *Base*, *A*, and *B*. This point provides a "stopping place" for the limited slice in *B* with respect to the new component, and permits *A* and *B* to be integrated successfully.

Given sufficiently precise congruence information, the YHR algorithm would successfully integrate both examples shown in Figure 13. However, the partitioning algorithm of Section 4.1.2 is only powerful enough to permit the YHR algorithm to succeed on the first example. To permit the YHR algorithm to succeed on the second example, either more powerful congruence-identifying techniques are needed, or information about where semantics-preserving transformations have been applied would have to be supplied.

Given program representation graphs for programs *Base*, *A*, and *B*, the YHR algorithm performs the following steps:

(1) Use an auxiliary method to obtain congruence information for the components of the three programs.

(2) Use the results of Step (1) to classify the vertices of each program's PRG to reflect how the behavior and text of the vertex in that program relates to the behavior and text of the corresponding vertices in the other two programs (as in the HPR algorithm, the YHR algorithm assumes that a correspondence between the vertices of *Base*, *A*, and *B* is given).

(3) Use the classification of Step (2) and the limited slicing operation to extract subgraphs that represent the changed and preserved computations of *A* and *B* with respect to *Base*.

(4) Combine the subgraphs to form a merged graph.

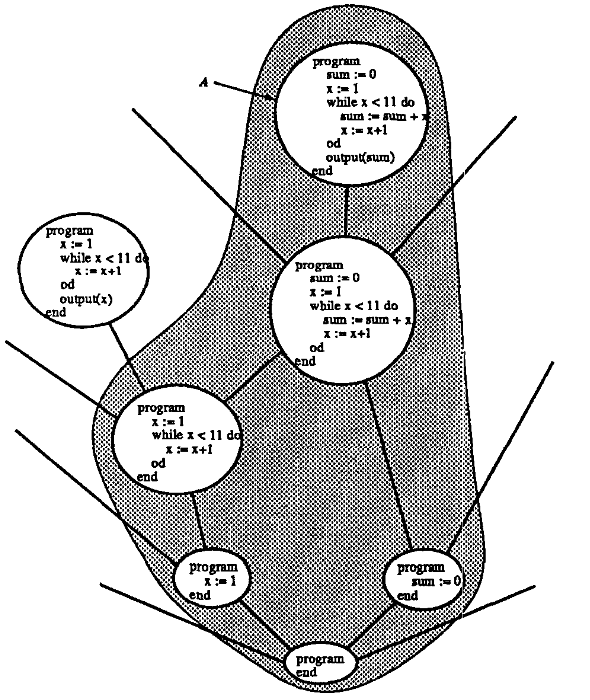
(5) Determine whether the merged graph represents a program; if so, produce the program.

The algorithm may determine that the variant programs interfere in either Step (2), Step (3), or Step (5).

Details of the YHR algorithm can be found in [44,46].

5.1.2. I.2: Integration without a component-correspondence relation

A significant limitation of the HPR and YHR algorithms is that neither algorithm can perform integrations unless a correspondence relation on the components of *Base*, *A*, and *B* is supplied. If the correspondence relation is supplied by a special editor (as is the case with our implementation), integration can only be performed on program variants created within the system; program variants created *outside* the system using ordinary text editors such as *vi* and *emacs* cannot be integrated.



```
program
  sum := 0
  x := 1
  while x < 11 do
    sum := sum + x
    x := x+1
  od
  output(sum)
end
```

```
program
  x := 1
  while x < 11 do
    x := x+1
  od
  output(x)
end
```

```
program
  sum := 0
  x := 1
  while x < 11 do
    sum := sum + x
    x := x+1
  od
end
```

```
program
  x := 1
  while x < 11 do
    x := x+1
  od
end
```

```
program
  x := 1
end
```

```
program
  sum := 0
end
```

```
program
end
```

Figure 14. The shaded region indicates the set in *P* that represents *DC*(*A*). Here, to keep the figure comprehensible, elements of the partial order are shown as *programs*, although the "is-a-slice-of" relation depicted (≤) is really a partial order on the programs' *PDGs*.

We recently made progress towards the goal of being able to integrate programs in the absence of a known correspondence relation on the components of *Base*, *A*, and *B*. This advance was an unanticipated result of our recent work on how to establish the algebraic properties of program integration [36,38], which introduced a lattice-theoretic framework for studying program integration. One instance of this framework yields an integration method that is a very close relative of the HPR algorithm. However, there is a slightly different instance of that framework that offers the possibility of eliminating the need for a component-correspondence relation in a program-integration system.

This material is organized as follows: we first summarize the lattice-theoretic reformulation of the HPR algorithm; we then describe how these ideas offer the possibility of a solution to the problem of performing integrations without a component-correspondence relation.

A lattice-theoretic framework for program integration

In [36,38], the HPR algorithm is reformulated as an operation in a *Brouwerian algebra* constructed from sets of dependence graphs in which vertices are tagged to indicate the corresponding vertices of *Base*, *A*, and *B*. A Brouwerian algebra is a particular kind of lattice; it has a greatest element $\top$, and, in addition to the binary operations join ($\sqcup$) and meet ($\sqcap$), it has a third binary operation, called "pseudo-difference," denoted by $\dot{-}$, which is characterized by the law $x \dot{-} y \sqsubseteq z$ iff $x \sqsubseteq y \sqcup z$.[13] Thus, the elements of a Brouwerian algebra obey the following nine axioms:

Idempotency	Commutativity	Associativity
$x \sqcup x = x$	$x \sqcup y = y \sqcup x$	$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
$x \sqcap x = x$	$x \sqcap y = y \sqcap x$	$(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
Absorptivity	Pseudo-difference	
$x \sqcup (x \sqcap y) = x$	$x \dot{-} y \sqsubseteq z$ iff $x \sqsubseteq y \sqcup z$	
$x \sqcap (x \sqcup y) = x$		

The *integration* of elements *x* and *y* with respect to element *base*, denoted by $x[base]y$, is defined by

$$x[base]y =_{df} (x \dot{-} base) \sqcup (x \sqcap base \sqcap y) \sqcup (y \dot{-} base).$$

The reason these definitions are of interest is because there is a Brouwerian algebra in which the integration operation $x[base]y$ is closely related to the operation of integrating via the HPR algorithm. This Brouwerian algebra is developed from a partial order on dependence graphs, denoted by $\leq$, that represents the relation "is-a-slice-of."[14] We say that a dependence graph *g* is a *single-point slice* iff there exists a vertex $v \in V(g)$ such that $b(g, v) = g$. Notice that if *x* and *y* are single-point slices, $y \leq x$ holds iff there exists a vertex *v* such that $y = b(x, v)$ (*i.e.*, if *y* is a single-point subslice of *x*). Let G_1 denote the set of all program dependence graphs that are single-point slices. (The relation $\leq$ on elements of G_1 is illustrated in Figure 14.) Given a set *A* of single-point slices, the *downwards-closure* of *A*, *DC*(*A*), is defined to be the set of single-point slices that are dominated by members of *A*:

$$DC(A) =_{df} \{ y \in G_1 \mid \exists x \in A . (y \leq x) \}.$$

It follows that $DC(A \cup B) = DC(A) \cup DC(B)$ and that

[13]The symbol $\sqsubseteq$ denotes the partial order on elements given by $x \sqsubseteq y$ iff $x \sqcap y = x$ (or, equivalently, $x \sqsubseteq y$ iff $x \sqcup y = y$).

[14]We extend the definition of slicing with respect to a vertex set (see footnote 10 of Section 4.1.1) to that of *slicing with respect to a PDG* as follows: let *x* and *y* be two PDGs; the slice of *x* with respect to *y*, denoted by $b(x, y)$, is defined to be the slice $b(x, V(y))$, where $V(y)$ denotes the vertex set of *y*. We say that *y is a slice of x* iff $b(x, y) = y$. The symbol $\leq$ denotes the partial order "is-a-slice-of" on PDGs (*i.e.*, $y \leq x$ iff *y* is a slice of *x*).

$DC(A \cap B) = DC(A) \cap DC(B)$. A set A is said to be *downwards closed* iff $DC(A) = A$.

In [36,38], a program is represented by the set of all of its single-point slices. P, the domain of program representations, is taken to be the collection of all downwards-closed sets of single-point slices. The elements of P are ordered by "subset-of." The shaded region of Figure 14 illustrates the element of P that represents the program

```
program
  sum := 0
  x := 1
  while x < 11 do
    sum := sum + x
    x := x + 1
  od
  output(sum)
end
```

Let $\top$ denote the set of all single-point slices, and $\perp$ the empty set. Let $\cup$, $\cap$, and $-$ denote the set-theoretic union, intersection, and difference operators. P is closed with respect to $\cup$ and $\cap$. P is not closed under $-$, but is closed under the pseudo-difference operator $\dot{-}$ defined as follows:

$$X \dot{-} Y =_{df} DC(X - Y).$$

Figure 15 illustrates the operation of pseudo-difference on two arbitrary elements of P. In general, an element of P can contain multiple maximal elements; these are indicated by the multiple peaks of sets X and Y in Figure 15. The value of $X \dot{-} Y$ is the downwards closure of $X - Y$; thus, $X \dot{-} Y$ includes both of the shaded regions shown in Figure 15.

One can show that $(P, \cup, \cap, \dot{-}, \top)$ is a Brouwerian algebra [36,38]. One can also show that the HPR integration algorithm corresponds very closely to the integration operation in $(P, \cup, \cap, \dot{-}, \top)$, namely

$$(x \dot{-} base) \cup (x \cap base \cap y) \cup (y \dot{-} base).$$

To be more precise, let $Rep : PDG \to P$ be the function that maps a PDG G to the (downwards-closed) set of all single-point slices of G:

$$Rep =_{df} \lambda pdg . \{ s \in G_1 \mid s \le pdg \}.$$

The relationship between the integration operation in $(P, \cup, \cap, \dot{-}, \top)$ and the HPR algorithm is captured by the following proposition:

Proposition. *If the integration of PDGs A and B with respect to Base via the HPR algorithm passes the Type I interference test, then*

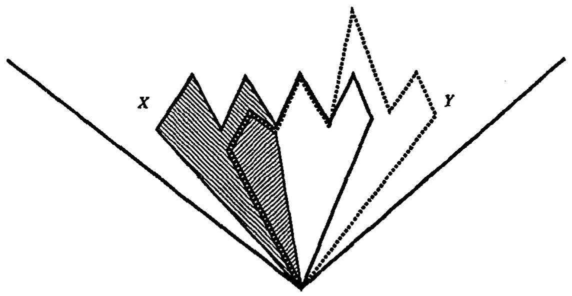


Figure 15. The value of $X \dot{-} Y$ is the downwards closure of $X - Y$. Thus, $X \dot{-} Y$ includes both of the shaded regions.

$$Rep(\Delta_{A,Base} \cup_g Pre_{A,Base,B} \cup_g \Delta_{B,Base}) \\ = Rep(A)[Rep(Base)]Rep(B).$$

A lattice for integration without tags

In the Brouwerian algebra $(P, \cup, \cap, \dot{-}, \top)$, the elements are downwards-closed sets of *tagged* single-point slices. However, it is possible to construct Brouwerian algebras whose elements are sets of dependence graphs that do *not* have tags on their components—for example, the lattice whose elements are downwards-closed sets of *untagged* single-point slices is also a Brouwerian algebra. Representations of such lattice elements can be easily constructed for programs that have been created with ordinary text editors.

Working with sets of untagged single-point slices has several consequences:

(1) Because slice vertices no longer have tags that can be used to identify corresponding components in different graphs, we must work with the notion of *isomorphism* between single-point slices. However, as discussed in Section 4.1.2, it is possible to test in *linear time* whether two single-point slices are isomorphic. Furthermore, by using hashing techniques the slice-set manipulations needed to perform operations in the algebra of downwards-closed sets of untagged single-point slices can be performed in linear expected time (*i.e.*, expected time linear in the sum of the sizes of the argument sets).

(2) The class of integration problems that can be handled successfully (*i.e.*, without interference being reported) in the lattice whose elements are sets of untagged slices is strictly larger than the class that can be handled in the lattice whose elements are sets of tagged slices. (The latter coincides with the class handled by the HPR algorithm.)

(3) The integration algorithm based on downwards-closed sets of untagged slices can produce a different answer than the algorithm based on downwards-closed sets of tagged slices (both in terms of the final program that is the result of an integration, as well as in the notion of when an integration fails due to interference). The reason is that more programs map to the same lattice element. In the lattice of downwards-closed sets of untagged slices, if a program has multiple slices that are isomorphic, the corresponding slice set will have only one copy of the duplicated slice. For example, we show below two programs and the set of *untagged* slices to which both programs correspond:

```
program          program
[1] x := 0       [1] x := 0
[2] y := x       [2] y := x
[3] w := x       [4] x := 0
end              [3] w := x
                 end
```

```
⎧ program , program , program , program ⎫
⎪ end         x := 0    x := 0    x := 0 ⎪
⎨           end         y := x    w := x ⎬
⎪                     end       end     ⎪
⎩                                        ⎭
```

Because the two occurrences of $x := 0$ in the second program have different tags (indicated by [1] and [4]), the two programs correspond to *different* elements in the lattice of downwards-closed sets of *tagged* slices:

```
⎧ program , program    , program    , program    ⎫
⎪ end       [1] x := 0   [1] x := 0   [1] x := 0 ⎪
⎨           end          [2] y := x   [3] w := x ⎬
⎪                        end          end        ⎪
⎩                                                ⎭
```

```
{ program , program   , program   , program   , program     }
  end       [1] x := 0  [1] x := 0  [4] x := 0  [4] x := 0
            end         [2] y := x  end         [3] w := x
                        end                     end
```

Nevertheless, for both lattices, whenever the set that results from evaluating $A\,[Base\,]B$ is feasible, the corresponding program meets the semantic criterion for integration given in Section 5.1. In other words, integration in the lattice whose elements are downwards-closed sets of *untagged* single-point slices also qualifies as an algorithm for semantics-based program integration.

Unfortunately, using the lattice whose elements are sets of untagged single-point slices for program integration does not completely solve the problem of integration in the absence of a component-correspondence relation. What is lacking is an algorithm for reconstituting the resulting program from the set of slices that result from an integration. It is an open question whether a practical method for this problem exists; this question is the subject of on-going research.

5.2. Program Integration for Multi-Procedure Programs

In this section we discuss a solution to integration problem I.1 (integration given a component-correspondence relation) for multi-procedure programs. There is currently no known solution to problem I.2 (integration without a component-correspondence relation) for multi-procedure programs; this problem is the subject of on-going research.

We first consider two straightforward extensions of the HPR algorithm to handle multi-procedure integration; however, neither algorithm proves to be satisfactory. The first algorithm fails to satisfy Property 2(i) of the integration model from the beginning of Section 5.1; the second algorithm does satisfy Property 2(i) of the integration model, but is unacceptable because it reports interference in cases where there is an intuitively acceptable integrated program. The latter example leads us to reformulate the integration model to capture better the goals of multi-procedure integration. After discussing the revised model for multi-procedure integration, we outline our multi-procedure integration algorithm. Details can be found in [8].

5.2.1. Straightforward extensions of the HPR algorithm

Our first candidate algorithm for multi-procedure integration applies the HPR algorithm separately to each of the procedures that make up a program (*i.e.*, for each procedure P in *Base*, A, and B, variant A and variant B's versions of P are integrated with respect to *Base*'s version of P). However, this technique is unsatisfactory because there are many examples in which an integrated program is produced, but that program fails to terminate normally on an initial state on which *Base*, A, and B all terminate normally (that is, the technique produces an integrated program that violates condition 2(i) of the integration model given in Section 5.1).

Example. Figure 16 shows programs *Base*, A, and B, and the integrated program that is produced by applying the HPR algorithm to the three versions of *Main* and to the three versions of P (the changes made to A and B are shown enclosed in boxes). This example motivates the need for a multi-procedure integration algorithm to take into account potential interactions between modifications that occur in different procedures in the variants.

The need to determine the potential effects of a change made in one procedure on components of other procedures suggests the use of interprocedural slicing. Thus, our second candidate algorithm for multi-procedure integration is a direct extension of the HPR algorithm: it performs the steps of the HPR algorithm exactly as given in Section 5.1.1, except that each *intra*procedural slicing operation is reinterpreted as an *inter*procedural slicing operation.

Although this reinterpretation does yield a multi-procedure integration algorithm that satisfies the integration model from Section 5.1, the algorithm obtained is unsatisfactory because it fails (*i.e.*, reports interference) on many examples for which, intuitively, integration should succeed. This is illustrated by the example shown in Figure 17, on which the direct extension of the HPR algorithm reports interference. Because the backward slices $b(S_{Base}, \{x := x+1\})$, $b(S_A, \{x := x+1\})$, and $b(S_B, \{x := x+1\})$ are pairwise unequal, the statement "$x := x+1$" is an affected point in both variants; therefore, the slices $b(S_A, \{x := x+1\})$ and $b(S_B, \{x := x+1\})$ are both included in the merged SDG S_M. However, because *both* slices are included in S_M, they are both "corrupted" in S_M; that is, although $b(S_A, \{x := x+1\})$ and $b(S_B, \{x := x+1\})$ are sub-*graphs* of S_M, neither $b(S_A, \{x := x+1\})$ nor $b(S_B, \{x := x+1\})$ is a sub-*slice* of S_M. For example, the slice $b(S_B, \{x := x+1\})$ includes actual-in vertex "$x_{in} := b$" for the second call on *Incr*, which has an incoming dependence edge from statement "$b := 4$." The slice $b(S_A, \{x := x+1\})$ also includes actual-in vertex "$x_{in} := b$" for the second call on *Incr*, but with an incoming dependence edge from "$b := 2$." Therefore, in the slice $b(S_M, \{x := x+1\})$, actual-in vertex "$x_{in} := b$" for the second call on *Incr* has incoming dependence edges from *both* "$b := 4$" and "$b := 2$." Consequently, graph S_M is found to have Type I interference, and the integration algorithm reports interference.

Further examination of the example in Figure 17 reveals that extending the programming language with procedures and call statements has introduced the same issue about the appropriate criterion for "changed behavior" that was discussed in the case of multi-procedure differencing in Section 4.2. It is certainly true that statement "$x := x+1$" produces three different sequences of values in *Base*, A, and B. However, statement "$x := x+1$" in variant A exhibits different behavior from *Base* only on the *first* invocation of *Incr*, and statement "$x := x+1$" in B exhibits different behavior from *Base* only on the *second* invocation of *Incr*. Thus, for this example it would seem desirable for the integration algorithm to

Base	Variant A	Variant B	Integrated Program
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*
$x := 1$	$x := 1$	$x := 1$	$x := 1$
call $P(x)$	**call** $P(x)$	**call** $P(x)$	**call** $P(x)$
output(x)	**output**(x)	$\boxed{x := 1/x}$	$x := 1/x$
end	**end**	**output**(x)	**output**(x)
		end	**end**
procedure $P(a)$	**procedure** $P(a)$	**procedure** $P(a)$	**procedure** $P(a)$
$a := a+1$	$\boxed{a := a-1}$	$a := a+1$	$a := a-1$
end	**end**	**end**	**end**

Figure 16. This example illustrates how using the HPR algorithm to integrate individual procedures can lead to a violation of property 2(i) of the integration model of Section 5.1. Although programs *Base*, A, and B all terminate normally, the integrated program performs a division by zero and thus fails to terminate normally. (The boxes indicate the modifications made to variants A and B.)

Base	Variant *A*	Variant *B*	Proposed Integrated System
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*
$a:=1$	$\boxed{a:=3}$	$a:=1$	$a:=3$
$b:=2$	$b:=2$	$\boxed{b:=4}$	$b:=4$
call *Incr* (a)	**call** *Incr* (a)	**call** *Incr* (a)	**call** *Incr* (a)
call *Incr* (b)	**call** *Incr* (b)	**call** *Incr* (b)	**call** *Incr* (b)
output(a, b)	output(a, b)	output(a, b)	output(a, b)
end	**end**	**end**	**end**
procedure *Incr* (x)	**procedure** *Incr* (x)	**procedure** *Incr* (x)	**procedure** *Incr* (x)
$x:=x+1$	$x:=x+1$	$x:=x+1$	$x:=x+1$
return	**return**	**return**	**return**

Figure 17. The program shown on the right incorporates the changed behaviors of both A and B as well as the preserved behavior of *Base*, A and B. However, a direct extension of the HPR algorithm that uses *inter*procedural slicing operations in place of *intra*procedural slicing operations would report interference on this example. (The boxes indicate the modifications made to variants A and B.)

succeed and return the program labeled "Proposed Integrated Program" in Figure 17; when this program is executed, it exhibits the changed behavior from variant A during the first invocation of *Incr* in addition to the changed behavior from variant B during the second invocation of *Incr*.

However, the program labeled "Proposed Integrated Program" in Figure 17 fails to meet Property (2) of the integration model from Section 5.1 (because the sequences of values produced at statement "$x:=x+1$" in *Base*, A, and B are pairwise unequal). This example suggests that the integration model from Section 5.1, which was originally introduced as a model for the integration of *single-procedure* programs, needs to be revised to characterize better the goals of multi-procedure integration. In particular, the integration model should capture the notion of changed execution behavior at a finer level of granularity.

In Section 4.2 we used a simplified criterion for equivalent behavior based on the "top-level" sequence of values produced at a point. For program integration, we use a more restrictive criterion based on the concept of *roll-out* [28]—the exhaustive in-line expansion of call statements to produce a program without procedure calls (in the presence of recursion, roll-out leads to an infinite program). The criterion used for program integration, and a multi-procedure integration model are presented in [8]. Roughly, the multi-procedure integration model requires that the rolled-out version of the integrated program must capture all of the changed and preserved behaviors of the rolled-out versions of the variants with respect to the rolled-out version of *Base*.

It should be stressed that our multi-procedure integration algorithm does not actually *perform* any roll-outs; the algorithm works on SDGs, which are *finite* representations of programs. Roll-out is simply a conceptual device introduced to formulate a satisfactory model of program integration.

A "rolled-out" program contains many occurrences—possibly an infinite number—of each statement in the original program. In the (possibly infinite) program obtained by roll-out, different occurrences of a given component of some procedure P correspond to invocations of P in different calling contexts. Consequently, one occurrence of a given component can have a different behavior in *roll−out*(A) than in *roll−out*(*Base*), while another occurrence of the component has a different behavior in *roll−out*(B) than in *roll−out*(*Base*) without there being interference.

Example. For each of the programs shown in Figure 17, the (finite) program obtained using roll-out contains two different occurrences of statement "$x:=x+1$," corresponding to the first and second invocations of procedure *Incr*. The change made in Variant A affects the behavior only at the first occurrence of "$x:=x+1$;" the change made in Variant B affects the behavior only at the second occurrence of "$x:=x+1$." When the program labeled "Proposed Integrated Program" in Figure 17 is rolled out, the resulting program captures both changes; when the program is executed, it exhibits the changed behavior from variant A at the first occurrence of "$x:=x+1$" as well as the changed behavior from variant B at the second occurrence of "$x:=x+1$."

To satisfy the new multi-procedure integration model, we developed an algorithm that is similar to the HPR algorithm in that it identifies slices of the programs' system dependence graphs that represent changed and preserved behavior, combines the graphs to form a merged SDG, and tests the merged graph for interference. However, these steps are defined so as to be consistent with the criterion for equivalent behavior based on roll-out. For example, the set of affected points is divided into two parts: the *strongly* affected points of variant A with respect to *Base* (denoted by $SAP_{A,\,Base}$) are those points with different behavior in *Base* and A according to the criterion discussed in Section 4.2—the points may produce different sequences of values when their respective procedures are called with the same actual parameters. The *weakly* affected points of A with respect to *Base* (denoted by $WAP_{A,\,Base}$) are those points that may produce different sequences of values in A because a new call was added, or because the actual parameters of an existing call were changed. The subgraph of A's SDG that captures the changed behavior of A with respect to *Base* (namely, $\Delta_{A,\,Base}$) is computed using different kinds of backward slices with respect to A's strongly and weakly affected points:

$$\Delta_{A,\,Base} =_{df} b(\,S_A,\, SAP_{A,\,Base})\,) \cup b2(\,S_A,\, WAP_{A,\,Base})$$

The preserved points of *Base* with respect to A and B are those points with equivalent behaviors according to the criterion of Section 4.2:

$$Pre_{A,\,Base,\,B} =_{df} b2(S_{Base},\ \{\, v \mid b2(S_A,\, v) = b2(S_{Base},\, v) = b2(S_B,\, v)\,\})$$

The merged SDG is formed by taking the union of the three graphs that represent the variants' changed and preserved behaviors:

$$S_M =_{df} \Delta_{A,\,Base} \cup_g \Delta_{B,\,Base} \cup_g Pre_{A,\,Base,\,B}$$

The details of the multi-procedure integration algorithm can be found in [8].

6. IMPLEMENTATION

The techniques for slicing, differencing, and integration that are described in this paper—as well as a few others not described—have been implemented in a prototype system, called the Wisconsin Program Integration System [35]. The Wisconsin Program Integration System is coupled to a program editor created using the Synthesizer Generator, a meta-system for creating interactive, language-based program editors [32]. Program analysis is carried out according to the editor's defining attribute grammar; the information gathered in this way is used to construct system dependence graphs. Commands are available in the editor to invoke the operations of slicing, differencing, and integration. The results of the slicing and differencing operations are displayed on the screen by highlighting appropriate components of the programs in a contrasting color (or, on black-and-white monitors, in a contrasting typeface). When an integration command is invoked, the integration algorithm is applied to the system dependence graphs that correspond to the programs indicated by the user. The system reports whether the variant programs interfere, and—if there is no interference—the integrated program is constructed and placed in a new editing buffer. When interference is detected (*i.e.*, integration fails), the system provides an interactive facility to help the user diagnose the cause of interference [34].

The Wisconsin Program Integration System can be obtained by contacting the authors. It is being distributed under license by the Computer Sciences Department at the University of Wisconsin–Madison. The distribution consists of the source code for the system together with a reference manual that documents how to use the system [37]. The system is written in C and SSL (the specification language of the Synthesizer Generator) and runs under UNIX on a variety of workstations. Further information about configuration requirements is available on request from the authors.

Acknowledgements

We would like to thank Thomas Ball, Samuel Bates, Victor Barger, David Binkley, Thomas Bricker, Phillip Pfeiffer, Jan Prins, G. Ramalingam, Genevieve Rosay, and Wuu Yang for their contributions to the work reported in this paper.

References

1. Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. Thesis, Department of Math. Sciences, Rice University, Houston, TX (April 1983).
2. Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988). Data Flow Analysis
3. Ball, T., Horwitz, S., and Reps, T., "Correctness of an algorithm for reconstituting a program from a dependence graph," Technical Report 947, Department of Computer Sciences, University of Wisconsin—Madison (July 1990).
4. Bannerjee, U., "Speedup of ordinary programs," Ph.D. Thesis, University of Illinois, Urbana, IL (October 1979).
5. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," *6th annual ACM Symposium on Principles of Programming Languages*, pp. 29-41 (January 1979).
6. Berzins, V., "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).
7. Binkley, D., Horwitz, S., and Reps, T., "Identifying semantic differences in programs with procedures," Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI (September 1991).
8. Binkley, D., "Multi-procedure program integration," Ph.D. Thesis and Technical Report 1038, Department of Computer Sciences, University of Wisconsin, Madison, WI (August 1991).
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
10. 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).
11. Feather, M.S., "Detecting interference when merging specification evolutions," Unpublished report, Information Sciences Institute, University of Southern California, Marina del Rey, CA (1989).
12. 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).
13. Gallagher, K.B., "Using program slicing in software maintenance," Ph.D. dissertation and Technical Report CS-90-05, Computer Science Department, University of Maryland, Baltimore Campus, Baltimore, MD (January 1990).
14. Hoffmann, C.M., *Group-Theoretic Algorithms and Graph Isomorphism, Lecture Notes in Computer Science*, Vol. 136, Springer-Verlag, New York, NY (1982).
15. Hopcroft, J.E., "An *n* log *n* algorithm for minimizing the states of a finite automaton," *The Theory of Machines and Computations,* pp. 189-196 (1971).
16. 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 15th ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).
17. 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).
18. 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).
19. Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plains, NY, June 1990), pp. 234-246 (June 1990).
20. 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).
21. Horwitz, S. and Reps, T., "Efficient comparison of program slices," *Acta Informatica* **28** pp. 713-732 (1991).
22. Hwang, J.C., Du, M.W., and Chou, C.R., "Finding program slices for recursive procedures," in *Proceedings of IEEE COMPSAC 88,* (Chicago, IL, Oct. 3-7, 1988), IEEE Computer Society, Washington, DC (1988).
23. Kuck, D. J., Muraoka, Y., and Chen, S. C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Transactions on Computers C-21,* pp. 1293-1310 (December 1972).
24. 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 (1981).
25. Lakhotia, A. and Sterling, L., "Composing recursive logic programs with clausal join," *New Generation Computing* **6**(2) pp. 211-225 (1988).
26. 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).
27. Luks, E., "Isomorphism of bounded valence can be tested in polynomial time," pp. 42-49 in *Proceedings of the Twenty-First IEEE Symposium on Foundations of Computer Science* (Syracuse, NY, October 1980), IEEE Computer Society, Washington, DC (1980).
28. Mogensen, T. and Holst, C.K., "[Partial Evaluation] Terminology," *New Generation Computing* **6** pp. 303-307 (1988).
29. 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).
30. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).
31. Podgurski, A. and Clarke, L.A., "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Transactions on Software Engineering* **SE-16**(9) pp. 965-979 (September 1990).
32. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A system for constructing language-based editors,* Springer-Verlag, New York, NY (1988).
33. Reps, T. and Yang, W., "The semantics of program slicing," Technical Report 777, Department of Computer Sciences, University of Wisconsin—Madison (June 1988).

34. Reps, T. and Bricker, T., "Illustrating interference in interfering versions of programs," *Proceedings of the 2nd International Workshop on Software Configuration Management,* (Princeton, NJ, October 24-27, 1989), *ACM SIGSOFT Software Engineering Notes* 17(7) pp. 46-55 (November 1989).

35. Reps, T., "Demonstration of a prototype tool for program integration," Technical Report 819, Department of Computer Sciences, University of Wisconsin—Madison (January 1989).

36. Reps, T., "Algebraic properties of program integration," in *Proceedings of the Third European Symposium on Programming,* (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science,* ed. N. Jones,Springer-Verlag, New York, NY (May 1990).

37. Reps, T., "The Wisconsin program-integration system reference manual," Unpublished document, Department of Computer Sciences, University of Wisconsin, Madison, WI (April 1990).

38. Reps, T., "Algebraic properties of program integration," To appear in *Science of Computer Programming,* ().

39. Rosen, B., Wegman, M.N., and Zadeck, F.K., "Global value numbers and redundant computations," pp. 12-27 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

40. Shapiro, R. M. and Saint, H., "The representation of algorithms," Technical Reprot CA-7002-1432, Massachusetts Computer Associates (February 1970).

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

42. Wolfe, M. J., "Optimizing supercompilers for supercomputers," Ph.D. Thesis, University of Illinois, Urbana, IL (October 1982).

43. Yang, W., Horwitz, S., and Reps, T., "Detecting program components with equivalent behaviors," Technical Report 840, Department of Computer Sciences, University of Wisconsin, Madison, WI (April 1989).

44. Yang, W., "A new algorithm for semantics-based program integration," Ph.D. Thesis, Department of Computer Sciences, University of Wisconsin, Madison, WI (1990).

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

46. Yang, W., Horwitz, S., and Reps, T., "A program integration algorithm that accommodates semantics-preserving transformations," To appear in *ACM Transactions on Software Engineering and Methodology,* (1992).