A NEW PROGRAM INTEGRATION ALGORITHM

by

Wuu Yang, Susan Horwitz & Thomas Reps

Computer Sciences Technical Report #899

December 1989
A New Program Integration Algorithm

WUU YANG, SUSAN HORWITZ, and THOMAS REPS
University of Wisconsin—Madison

Program integration attempts to construct a merged program from several related but different variants of a base program. The merged program must include the changed computations of the variants as well as the computations of the base program that are preserved in all variants.

A fundamental problem of program integration is determining the sets of changed and preserved computations of each variant. This paper first describes a new algorithm for partitioning program components (in one or more programs) into disjoint equivalence classes so that two components are in the same class only if they have the same execution behavior. This partitioning algorithm can be used to identify changed and preserved computations, and thus forms the basis for the new program-integration algorithm presented here. The new program-integration algorithm is strictly better than the original algorithm of Horwitz, Prins, and Reps: integrated programs produced by the new algorithm have the same semantic properties relative to the base program and its variants as do integrated programs produced by the original algorithm, the new algorithm successfully integrates program variants whenever the original algorithm does, but there are classes of program modifications for which the new algorithm succeeds while the original algorithm reports interference.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Tools and Techniques – programmer workbench; D.2.3 [Software Engineering]: Coding – program editors; D.2.6 [Software Engineering]: Programming Environments; D.2.7 [Software Engineering]: Distribution and Maintenance – enhancement, restructuring, version control; D.2.9 [Software Engineering]: Management – programming teams, software configuration management; D.3.4 [Programming Languages]: Processors – compilers, interpreters, optimization; E.1 [Data Structures] graphs

General Terms: Algorithms, Design

Additional Key Words and Phrases: coarsest partition, control dependence, data congruence, data dependence, dataflow analysis, flow dependence, program dependence graph, program integration, program representation graph, sequence congruence, static-single-assignment form

1. INTRODUCTION

Given a base program Base and a set of variant programs, each created by modifying a copy of Base, the goal of program integration is to determine whether the variants incorporate interfering changes, and if not, to create a single program that includes the changes introduced in the variants as well as the portions of Base that are preserved in all variants. Although text-merging tools that address this problem have existed for years, when used for merging programs they are unsafe, in the sense that they do not protect against...
unwanted interactions between the parts of the integrated program that are incorporated from different variants. Thus, one has no guarantees about how the execution behavior of the integrated program relates to the behaviors of the programs that are the arguments to the merge.

The first algorithm to provide any such guarantees was given by Horwitz, Prins, and Reps in [Horwitz88, Horwitz89]. This algorithm—referred to hereafter as the HPR algorithm—guarantees that the following semantic property holds for the integrated program in cases where the algorithm determines that the variant programs do not interfere [Reps89]:¹

If the HPR algorithm is applied to base program Base and variant programs A and B,² and if integration succeeds, producing program M, then for any initial state σ on which Base, A, and B all terminate normally,³ M has the following properties:

1. M terminates normally on σ.
2. For any variable x that has final value ν after executing A on σ, and a different final value ν' after executing Base on σ, x has final value μ after executing M on σ (i.e., M agrees with A on x).
3. For any variable y that has final value ν after executing B on σ, and a different final value ν' after executing Base on σ, y has final value μ after executing M on σ (i.e., M agrees with B on y).
4. For any variable z that has the same final value ν after executing Base, A, and B on σ, μ has the same final value ν after executing M on σ (i.e., M agrees with Base, A, and B on z).

More informally: changes in the behavior of A and B with respect to Base are detected and preserved in the integrated program, along with the unchanged behavior of all three.

Properties (1)–(4) can be taken to define a semantic criterion for integration and interference: any program M that satisfies (1)–(4) integrates Base, A, and B; if no such program exists then A and B interfere with respect to Base. However, this criterion is not decidable; it requires being able to determine, for all possible initial states, which variables of a variant program have the same final values as their counterparts in the base program. Thus, any program-integration algorithm must use techniques that compute a safe approximation to this set of variables. (In this case safe means that two inequivalent variables must never be identified as being equivalent.) Consequently, any program-integration algorithm will sometimes fail to produce an integrated program even though there is actually no interference (i.e., even when there is some program that meets the integration criterion given above).

As a practical matter, it is desirable to place further restrictions on how the integrated program M is constructed from Base, A, and B:

1. M must be constructed from components of A and B and no other components.
2. Each component of M must behave in exactly the same way as one of its counterparts in A or B.

¹The HPR algorithm applies to programs written in a simple language that includes scalar variables, assignment statements, conditional statements, while loops, and final output statements (called end statements). By definition, only those variables listed in the end statement have values in the final state. 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.
²Both the HPR algorithm and the new algorithm can accommodate any number of variants; for the sake of exposition, we consider the common case of two variants.
³There are two ways in which a program may fail to terminate normally: (1) the program has a non-terminating loop, or (2) a fault such as division by zero occurs.
Thus, a fundamental problem is how to determine which components of a variant program might produce different values than the analogous components of the base program. (We call such components affected components.)

The HPR algorithm uses program slices [Weiser84] to find affected components. If a component c’s slice in the base program differs from its slice in a variant, then the way c’s values are computed differs in the base program and the variant, and thus the values themselves might differ. Therefore, any component whose slice in the base program differs from its slice in a variant is considered to be an affected component by the HPR algorithm.

The goal of the work described in this paper was to find an appropriate way to extend the HPR algorithm with a sharper technique for identifying affected components. We recognized that an idea introduced by Alpern, Wegman, and Zadeck in [Alpern88], which uses a certain graph representation of programs to find “equivalence classes” of program components, provided a possible basis for extending the integration method in this way. The algorithm of Alpern, Wegman, and Zadeck first optimistically groups possibly equivalent components in an initial partition, and then finds the coarsest partition of the components that is consistent with the initial partition and the edges of the graph.

However, their equivalence-testing algorithm was not suitable for our purposes; the property that holds for components in the same “equivalence class” is that components of a single program that are in the same final partition produce the same value at certain moments during program execution [Alpern88]. There are two reasons why this is not the appropriate property for our purposes: (1) for integration, it is necessary to be able to identify equivalent components in several programs simultaneously; (2) equivalent components must produce identical sequences of values. Consequently, we developed a new algorithm that uses the partitioning idea to find equivalent components, called the Sequence-Congruence Algorithm [Yang89]. The affected components determined using the Sequence-Congruence Algorithm are a subset of the affected components determined using program slicing (but are still a safe approximation to the exact set of affected components).

This paper describes a new program-integration algorithm that uses the Sequence-Congruence Algorithm to find affected components. The new integration algorithm is quite different from the HPR algorithm. In addition to using a different method for determining affected components, it uses a different underlying graph representation of programs and uses different criteria to extract changed and preserved components from the variants to be assembled into the merged program.

Despite these differences, the new algorithm shares with the HPR algorithm the same characterization of the execution behavior of the integrated program in terms of the behaviors of the base program and the two integrands. In addition, it can be shown that the new algorithm is strictly better than the HPR algorithm in the following sense.

1. The new algorithm succeeds whenever the HPR algorithm succeeds.

2. There are classes of program modifications for which the new algorithm succeeds but the HPR algorithm reports interference.

The kinds of changes that cause components to be (pessimistically) classified as affected using program slicing but classified as unaffected using the Sequence-Congruence Algorithm include changing variable

---

*The slice of a program with respect to a component c (where a program component is an assignment statement, a predicate, or an end statement) is the set of program components that might affect (either directly or transitively) the values of the variables used at c [Weiser84, Ottenstein84].
names, inserting or deleting statements that copy values from a constant to a variable or from one variable to another, and some instances of moving assignments into or out of conditional statements. Examples of these three kinds of changes are given in Figure 1.

Figure 1 shows three sets of programs, each set containing a base program and two variants. In all three cases, the slice with respect to the assignment to variable *area* in variant *A* differs from the corresponding slice in *Base*. Thus, the HPR algorithm would classify that assignment as an affected component (although in fact the value assigned to *area* is the same in variant *A* as in *Base*). This classification, in conjunction with the fact that variant *B* introduces new code that uses the value of *area* (namely, the assignment to *vol*)

<table>
<thead>
<tr>
<th>Base</th>
<th>Variant A</th>
<th>Variant B</th>
<th>Integrated Program Produced by the New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P := 3.14$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rad := 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>area := $P \times (rad^{**2})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end(area)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$PI := 3.14$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rad := 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>area := $PI \times (rad^{**2})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end(area)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P := 3.14$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rad := 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>area := $P \times (rad^{**2})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>height := 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vol := height*area</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end(area, vol)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$PI := 3.14$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rad := 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>area := $PI \times (rad^{**2})$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>height := 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vol := height*area</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>end(area, vol)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| program |
| $P := 3.14$ |
| rad := 2 |
| area := $P \times (rad^{**2})$ |
| end(area) |
| program |
| rad := 2 |
| area := $3.14 \times (rad^{**2})$ |
| end(area) |
| program |
| $P := 3.14$ |
| rad := 2 |
| area := $P \times (rad^{**2})$ |
| height := 4 |
| vol := height*area |
| end(area, vol) |
| program |
| rad := 2 |
| area := $3.14 \times (rad^{**2})$ |
| height := 4 |
| vol := height*area |
| end(area, vol) |

| program |
| $P := 3.14$ |
| rad := 2 |
| if DEBUG |
| then rad := 4 |
| else rad := 2 |
| fi |
| area := $P \times (rad^{**2})$ |
| end(area) |
| program |
| $P := 3.14$ |
| if DEBUG |
| then rad := 4 |
| else rad := 2 |
| fi |
| area := $P \times (rad^{**2})$ |
| height := 4 |
| vol := height*area |
| end(area, vol) |
| program |
| $P := 3.14$ |
| if DEBUG |
| then rad := 4 |
| else rad := 2 |
| fi |
| area := $P \times (rad^{**2})$ |
| height := 4 |
| vol := height*area |
| end(area, vol) |

Figure 1. Three example integration problems that illustrate the three kinds of changes that cause the HPR algorithm to report interference, but for which the new algorithm produces the integrated programs shown. The first example illustrates variable renaming (*P* is renamed *PI* in variant *A*); the second example illustrates a value being used directly *vs.* being passed through a variable; the third example illustrates an assignment being moved into a conditional.
leads the HPR algorithm to determine that the variants incorporate interfering changes. In fact, there is no interference in any of these examples, and the new integration algorithm would succeed in all cases, producing the integrated programs as shown.

The remainder of this paper defines and discusses the Sequence-Congruence Algorithm and the new program-integration algorithm. Both algorithms use a graph representation of programs called the Program Representation Graph (first defined in [Yang89]), which combines features of program dependence graphs [Kuck81, Ferrante87, Horwitz88, Horwitz89] and static single assignment forms [Shapiro70, Alpern88, Cytron89, Rosen88]. Program Representation Graphs are defined in Section 2. Section 3 describes the Sequence-Congruence Algorithm. The Sequence-Congruence Algorithm can be applied to the Program Representation Graphs of one or more programs; the algorithm partitions the vertices of the graph(s) into disjoint equivalence classes so that two vertices are in the same class only if the program components that they represent have equivalent behaviors (a definition of equivalent behavior is given in Section 3). Section 4 presents the new integration algorithm. Section 5 proves that when the new integration algorithm successfully produces an integrated programs, that program satisfies the semantic criterion given above. Section 6 shows that the new integration algorithm is strictly better than the HPR algorithm. Section 7 discusses the relation between the result reported in this paper and previous work.

2. PROGRAM REPRESENTATION GRAPHS

Both the Sequence-Congruence Algorithm and the new program-integration algorithm use a graph representation of programs called a Program Representation Graph. Program Representation Graphs (PRGs) are currently defined only for programs in a limited language that includes scalar variables, assignment statements, conditional statements, while loops, and final output statements called end statements. PRGs combine features of program dependence graphs [Kuck81, Ferrante87, Horwitz88, Horwitz89] and static single assignment forms [Shapiro70, Alpern88, Cytron89, Rosen88].

A program's PRG is defined in terms of an augmented version of the program's control-flow graph. The standard control-flow graph includes a special Entry vertex and one vertex for each if or while predicate, and each assignment statement. The control-flow graph is augmented as follows. First, a final-use vertex, labeled "FinalUse(x)_", is added for each variable x named in the program's end statement. The relative order of these vertices is arbitrary; however, they must appear sequentially, following all other vertices of the control-flow graph. Second, as in static single assignment forms, the control-flow graph is augmented by adding special "φ vertices" so that each use of a variable in an assignment statement, a predicate, or the end statement is reached by exactly one definition.

(1) For each variable x that is defined within either (or both) branches of an if statement and is live at the end of the if statement, a "φiff" vertex labeled "φiff: x := x" is added to the control-flow graph immediately following the if statement. If there is more than one such vertex, their relative order is arbitrary.

(2) For each variable x that is defined within a while loop, and is live immediately before the loop predicate (i.e., may be used before being redefined either inside the loop or after the loop), a "φenter" vertex labeled "φenter: x := x" is added to the control-flow graph inside the loop, before the loop predicate. If there is more than one such vertex, their relative order is arbitrary.

(3) For each variable x that is defined within a while loop and is live after the loop, a "φexit" vertex labeled "φexit: x := x" is added to the control-flow graph immediately after the loop. If there is more than one such vertex, their relative order is arbitrary.
Finally, for each variable $x$ that may be used before being defined (i.e., there is an $x$-definition clear path in the control-flow graph from the $Entry$ vertex to a vertex that uses $x$), an initial-definition vertex, labeled "$x := Init\_State(x)$," is added to the control-flow graph after the $Entry$ vertex. This vertex represents the assignment to $x$ of a value from the initial state. If there is more than one such vertex, their relative order is arbitrary; however, they must appear sequentially, following the $Entry$ vertex and preceding all other vertices in the control-flow graph.

**Example.** Figures 2(a) and 2(b) show a program and its augmented control-flow graph.

The vertices of a program’s Program Representation Graph (PRG) are the same as the vertices in the augmented control-flow graph (an $Entry$ vertex, one vertex for each predicate, and each assignment statement, and for each initial definition, final use, $\phi_f$, $\phi_{\text{enter}}$, and $\phi_{\text{exit}}$ vertex). The edges of the PRG represent control and flow dependences.

---

**Figure 2.** (a) A program; (b) its augmented control-flow graph; (c) its Program Representation Graph. In the Program Representation Graph, control dependence edges are shown using bold arrows and the edges are shown without their labels (in this example, all control dependence edges would be labeled true); data dependence edges are shown using arcs.
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 that if the program component represented by vertex \( v \) is evaluated during program execution and its value matches the label on the edge, then, (assuming termination of all loops) the component represented by \( w \) will eventually execute, while if the component represented by \( v \) is evaluated and its 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 \text{true}.)

Algorithms for computing control dependences in languages with unrestricted control flow are given in [Ferrante87, Cytron89]. For the restricted language under consideration here, control dependence edges reflect the nesting structure of the program (i.e., there is an edge labeled \text{true} from the vertex that represents a \text{while} predicate to all vertices that represent statements inside the loop; there is an edge labeled \text{true} from the vertex that represents an \text{if} predicate to all vertices that represent statements in the true branch of the \text{if}; and an edge labeled \text{false} to all vertices that represent statements in the false branch; there is an edge labeled \text{true} from the Entry vertex to all vertices that represent statements or predicates that are not inside any \text{while} loop or \text{if} statement). In addition, there is a control dependence edge labeled \text{true} from every vertex that represents a \text{while} predicate to itself.

Flow dependence edges represent the 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 component that uses the value of variable \( x \), and there is an \( x \)-definition clear path from \( v \) to \( w \) in the augmented control-flow graph.

\textit{Example.} Figure 2(c) shows the Program Representation Graph of the program of Figure 2(a). Control dependence edges are shown using bold arrows and their labels have been omitted (in this example, all control dependence edges would be labeled \text{true}); data dependence edges are shown using arcs.

Textually different programs may have identical Program Representation Graphs. However, we have shown that if two programs have the same graph, then the programs are semantically equivalent [Yang90].

THEOREM. (EQUIVALENCE THEOREM FOR PROGRAM REPRESENTATION GRAPHS). Suppose \( P \) and \( Q \) are programs such that the Program Representation Graph of \( P \) is identical to the Program Representation Graph of \( Q \). If \( \sigma \) is a state on which \( P \) terminates normally, then for any state \( \sigma' \) that agrees with \( \sigma \) on all variables for which the graphs contain initial-definition vertices, (1) \( Q \) terminates normally on \( \sigma' \), (2) \( P \) and \( Q \) compute the same sequence of values at each corresponding program component, and (3) the final states of \( P \) and \( Q \) agree on all variables for which the graphs contain final-use vertices.

3. THE SEQUENCE-CONGRUENCE ALGORITHM

The Sequence-Congruence Algorithm can be applied to the Program Representation Graphs of one or more programs. The algorithm partitions the vertices of the graph(s) into disjoint equivalence classes so that two vertices are in the same class only if the program components that they represent have equivalent behaviors in the following sense:

\textit{Definition.} (Equivalent behavior of program components.) Two components \( c_1 \) and \( c_2 \) of (not necessarily distinct) programs \( P_1 \) and \( P_2 \) respectively, have equivalent behaviors if and only if all four of the following hold:

1. For all initial states \( \sigma \) such that both \( P_1 \) and \( P_2 \) terminate normally when executed on \( \sigma \), the sequence of values produced at component \( c_1 \) when \( P_1 \) is executed on \( \sigma \) is identical to the sequence of values produced at component \( c_2 \) when \( P_2 \) is executed on \( \sigma \).
(2) For all initial states \( \sigma \) such that neither \( P_1 \) nor \( P_2 \) terminates normally when executed on \( \sigma \), either the sequence of values produced at component \( c_1 \) is an initial sub-sequence of the sequence of values produced at \( c_2 \) or vice versa.

(3) For all initial states \( \sigma \) such that \( P_1 \) terminates normally on \( \sigma \) but \( P_2 \) fails to terminate normally on \( \sigma \), the sequence of values produced at \( c_2 \) is an initial sub-sequence of the sequence of values produced at \( c_1 \).

(4) For all initial states \( \sigma \) such that \( P_2 \) terminates normally on \( \sigma \) but \( P_1 \) fails to terminate normally on \( \sigma \), the sequence of values produced at \( c_1 \) is an initial sub-sequence of the sequence of values produced at \( c_2 \).

By "the sequence of values produced at a component" we mean: For an assignment statement (including initial-definition statements and \( \phi \) statements), the sequence of values assigned to the left-hand-side variable; for a predicate, the sequence of boolean values to which the predicate evaluates; and for a variable named in the end statement, the final value of that variable. Note that a fault such as integer overflow is considered to be a special "value" in the above definition. Thus, suppose a fault occurs during the \( k^{th} \) evaluation of \( c_1 \). Then program \( P_2 \) cannot terminate normally and the same fault must occur during the \( k^{th} \) evaluation of \( c_2 \), if \( c_2 \) is evaluated \( k \) times.

A component's execution behavior depends on three factors: the operator in the component, the operands available when the operator is applied, and the predicates that control the execution of the operation. It is not unreasonable to assume that vertices with different operators, inequivalent operands, or inequivalent controlling predicates will have inequivalent execution behaviors (although there do exist program components that have equivalent behavior but have different operators, inequivalent operands, or inequivalent controlling predicates).

The Sequence-Congruence Algorithm is based on the above assumption. Given one or more programs, the Algorithm divides components of the programs that have different operators, inequivalent operands, or inequivalent controlling predicates into disjoint equivalence classes. Initially, components with different operators are put into different partitions. Flow dependences and control dependences are used to refine the initial partition. Components that are in the same final equivalence classes will have the same operators, equivalent operands, and equivalent controlling predicates.

The Sequence-Congruence Algorithm consists of two passes. Both passes use an algorithm called the Basic Partitioning Algorithm that was adapted from [Alpern88, Aho74], and is based on an algorithm of [Hopcroft71] for minimizing a finite state machine. Figure 3 shows the Basic Partitioning Algorithm where the \( m \)-successors of a vertex \( u \) are the vertices \( v \) such that there is an edge \( u \to v \) of type \( m \) (the type of an edge is defined below). The Basic Partitioning Algorithm 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 if and only if they are in the same initial partition, and, for every predecessor \( u \) of \( v \), there is an analogous predecessor \( u' \) of \( v' \) such that \( u \) and \( u' \) are in the same class after partitioning.

The two passes of the Sequence-Congruence Algorithm apply the Basic Partitioning Algorithm to different initial partitions, and make use of different sets of edges. The first pass creates an initial partition based on the operators used at the vertices. Flow dependence edges (and some additional edges) are used in the first pass to refine the initial partition. The second pass starts with the final partition produced by the first pass; control dependence edges are used to further refine this partition.

The operator in a statement or a predicate vertex is determined from the expression part of the vertex. For example, statement "\( x := a + b * c \)" has the same operator as statement "\( y := d + e * f \)" but a different
The Basic Partitioning Algorithm:

The initial partition is \( B[1], B[2], \ldots, B[p] \)
\[
\text{WAITING} := \{ 1, 2, \ldots, p \}
\]
\[
q := p
\]
while \( \text{WAITING} \neq \emptyset \) do
    select and delete an integer \( i \) from \( \text{WAITING} \)
    for each type \( m \) of edge do
        \( \text{FOLLOWER} := \emptyset \)
        for each vertex \( u \) in \( B[i] \) do
            \( \text{FOLLOWER} := \text{FOLLOWER} \cup m\text{-successor}(u) \)
        od
        for each \( j \) such that \( B[j] \cap \text{FOLLOWER} \neq \emptyset \) and \( B[j] \subseteq \text{FOLLOWER} \) do
            \( q := q + 1 \)
            create a new class \( B[q] \)
            \( B[q] := B[j] \cap \text{FOLLOWER} \)
            if \( j \in \text{WAITING} \)
                then add \( q \) to \( \text{WAITING} \)
            else if \( \text{size}(B[j]) \leq \text{size}(B[q]) \)
                then add \( j \) to \( \text{WAITING} \)
            else add \( q \) to \( \text{WAITING} \)
            fi
        od
    od

Figure 3. The Basic Partitioning Algorithm. This algorithm, which is adapted from [Alpern88, Aho74], 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 if and only if they are in the same initial partition and for every predecessor \( u \) of \( v \) there is an analogous predecessor \( u' \) of \( v' \) such that \( u \) and \( u' \) are in the same class after partitioning.

Operator than statement "\( z := g \ast h \)"; that is, the structure of the expression in the vertex defines the operator. The expression "\( a + b \ast c \)" uses the operator that takes three arguments \( a, b, \) and \( c, \) and returns the value of "\( a + b \ast c \)".

A predicate is simple if it consists of a single boolean variable; an assignment statement is simple if its right-hand-side expression consists of a single variable. Both vertices that represent simple predicates and vertices that represent simple assignments are referred to as simple vertices. The operator in a simple vertex is the identity operator, that is, an operator that takes one argument and returns the value of the argument. Examples of simple vertices include: "if \( P \)," "\( y := x \)," and

The operator in a vertex whose expression consists of a single constant is the constant operator that takes no argument and always returns the value of that constant (i.e., there is a different operator for each constant value).
Two vertices that are the same kind of $\phi$ vertex (i.e., $\phi_{\text{enter}}$, $\phi_{\text{exit}}$, or $\phi_f$) or that have the same operators must have the same number of incoming control and flow dependence edges. Thus, we can speak of the "analogous" flow (or control) predecessors of the two vertices. To be more specific, we assign types to edges in the PRGs; the notion of analogous flow (or control) predecessors of two vertices is then defined in terms of the types of edges. (Note that the numbers for the edge types specified below are chosen arbitrarily; these numbers are used only to distinguish different types of edges.)

Due to the presence of $\phi$ vertices in PRGs, each use of a variable in a non-$\phi$ vertex is reached by exactly one definition (either an original assignment statement, an initial-definition assignment, or a $\phi$ assignment). Therefore, if the operator in a non-$\phi$ vertex is an $n$-ary operator, there are exactly $n$ incoming flow dependence edges for this vertex. These flow dependence edges are assigned types 1, 2, ..., $n$, one for each operand. Edge-type numbers for other kinds of edges in a PRG start at $m + 1$, where $m$ is the greatest number of flow edges incident on some non-$\phi$ vertex. In what follows, we will assume that $m = 3$, and start numbering other edges at 4.

A vertex $u$ labeled "$\phi_f$: $x := x'$" has two incoming flow dependence edges: one represents the value that flows to $u$ from or via the true branch of the associated if statement; the other represents the value that flows to $u$ from or via the false branch. The flow dependence edges incident on a $\phi_f$ vertex are assigned types 4 and 5, respectively. For example, consider the following program fragment:

```plaintext
<T1>   x := 1
       if $P$ then
<T2>   x := 2
       fi
<T3>   $\phi_f$: $x := x$
```

The definition at T1 reaches T3 via the false branch of the if statement, so the flow dependence edge from T1 to T3 has type 5. The definition at T2 reaches T3 from the true branch, so the flow dependence edge from T2 to T3 has type 4.

A vertex $u$ labeled "$\phi_{\text{enter}}$: $x := x'$" has two incoming flow dependence edges: one represents the value that flows to $u$ from outside the associated loop (due to an assignment to $x$ before the loop); the other represents the value that flows to $u$ from inside the loop. These flow dependence edges are assigned types 6 and 7, respectively.

A vertex $u$ labeled "$\phi_{\text{exit}}$: $x := x'$" has one incoming flow dependence edge; the source of this flow dependence edge is the associated $\phi_{\text{enter}}$ vertex. The flow dependence edge incident on a $\phi_{\text{exit}}$ vertex is assigned type 8.

All vertices except $\phi_{\text{enter}}$ and while predicate vertices have exactly one incoming control dependence edge. The control dependence edges that form self-loops on while predicates are assigned type 9. The incoming control dependence edge of a $\phi_{\text{enter}}$ vertex $u$ whose source is not the associated while predicate for $u$ is assigned type 10 or 11 depending on whether the label on the control dependence edge is true or false. All other control dependence edges are assigned type 12 or 13 depending on whether the label on the control dependence edge is true or false.

The analogous flow (or control) predecessors of two vertices $u_1$ and $u_2$ are two vertices $v_1$ and $v_2$ such that the flow (or control, respectively) dependence edges $u_1 \rightarrow v_1$ and $u_2 \rightarrow v_2$ have the same type.

Figure 4 presents the Sequence-Congruence Algorithm, which operates on one or more Program Representation Graphs. When the algorithm operates on more than one PRG, the multiple PRGs are treated as one graph; thus, when we refer below to "the graph," we mean the collection of PRGs.
The Sequence-Congruence Algorithm:

Pass 1:
- Add an if-edge from every if predicate to each associated $\phi_f$ vertex.
- Add a while-edge from every while predicate to each associated $\phi_{exit}$ vertex.
- Merge non-$\phi$ vertices that use identity operators with their flow predecessors.
- Create an initial partition using the operators in the vertices as explained in the text.
- Apply the Basic Partitioning Algorithm to refine the initial partition, ignoring all control dependence edges.
- Remove all if and while edges.
- Undo all merge operations.

Pass 2:
- Apply the Basic Partitioning Algorithm to the partition obtained from the first pass, using only control dependence edges to further refine the partition.

**Figure 4.** The Sequence-Congruence Algorithm. The Sequence-Congruence Algorithm consists of two passes. Both passes make use of the Basic Partitioning Algorithm presented in Figure 3; only the starting partition and the edges considered in the two passes are different.

**Pass 1:**
For the first pass, some additional edges are added to the graph: an edge from every if predicate to each associated $\phi_f$ vertex and an edge from every while predicate to each associated $\phi_{exit}$ vertex are added to the PRGs. These added edges are assigned types 14 and 15, respectively. Also, for the first pass, non-$\phi$ vertices with identity operators are merged with their (single) flow predecessors. To merge vertex $v$ with vertex $u$, replace every edge $v \rightarrow x$ with edge $u \rightarrow x$, remove edge $u \rightarrow v$, and remove vertex $v$. (This merge operation will be undone before the second pass, but vertices $u$ and $v$ will remain in the same partition.)

The initial partition is based on the operators in the vertices. Initially, there is a class for all the Entry vertices; for each variable $x$ there is a class for all the initial-definition vertices for $x$; there is a class for all non-$\phi$ vertices that have the same operators; for each nesting level of while loops, there is a class for all the $\phi_{exit}$ vertices at this nesting level; there is a class for all the $\phi_{exit}$ vertices; there is a class for all the $\phi_f$ vertices.

The initial partition is refined by the Basic Partitioning Algorithm; however, all control dependence edges are ignored in the first pass.

At the end of the first pass, the edges added at the beginning of the first pass — those of types 14 and 15 — are discarded. Also, all merge operations performed at the beginning of the first pass are undone.

**Pass 2:**
The second pass considers only control dependence edges, and applies the Basic Partitioning Algorithm again to refine the partition obtained from the first pass.

The time required by the Sequence-Congruence Algorithm is $O(N \log N)$, where $N$ is the sum of the sizes of the Program Representation Graphs (i.e., number of vertices + number of edges) to which the algorithm is applied.
Definition. Vertices are sequence-congruent if they are in the same class after the second pass of partitioning.

The Sequence-Congruence Theorem [Yang89] states that program components represented by sequence-congruent vertices have equivalent execution behaviors in the sense defined at the beginning of Section 3. This Theorem establishes the ability of the Sequence-Congruence Algorithm to detect program components with equivalent execution behaviors.

Theorem. (Sequence-Congruence Theorem). If two vertices are sequence-congruent, then the program components represented by the two vertices have equivalent behaviors.

Example. Figure 5 shows the final partition created by applying the Sequence-Congruence Algorithm to

---

**Figure 5.** Partitioning Example. The final partition created by the Sequence-Congruence Algorithm for the programs of the third example of Figure 1.
the third set of programs in Figure 1. Although the three occurrences of "rad := 2" are in the same initial partition, the component from variant \( A \) is in a different final partition than the analogous components from \( \text{Base} \) and variant \( B \). This is because "rad := 2" is executed unconditionally in \( \text{Base} \) and in variant \( B \); thus, the sequence of values produced at this component in those programs is never empty. However, the sequence of values produced at this component in variant \( A \) is empty if the initial value of \( \text{DEBUG} \) is true. Note that the component \( \text{"area := \text{Pow}((\text{rad}**2))"} \) from variant \( A \) is in the same final partition as the analogous components of \( \text{Base} \) and \( B \) (and is thus guaranteed to assign the same value to the variable \( \text{area} \)) even though the slice of \( A \) with respect to this component is not the same as the slice with respect to the analogous components of \( \text{Base} \) and \( B \).

4. THE NEW INTEGRATION ALGORITHM

Given a base program \( \text{Base} \) and variant programs \( A \) and \( B \), the new integration algorithm performs the following steps:

1. Apply the Sequence-Congruence Algorithm to the Program Representation Graphs of the three programs.

2. Use the sequence-congruence classes produced in Step (1) to classify the vertices of each PRG.

3. Use the classification of Step (2) to compute subgraphs that represent the changed and preserved computations of the variant programs with respect to the base program.

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).

4.1. Classification of Vertices

There are two kinds of changes that can be introduced by a variant program: a change in execution behavior, or a change in text that does not affect execution behavior. The new integration algorithm attempts to preserve both kinds of changes in the integrated program. The non-\( \varphi \) vertices\(^6\) in each of the three programs (\( \text{Base} \), \( A \), and \( B \)) are classified as defined below 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.

The first problem is, given a vertex in one program, which are the corresponding vertices in the other two programs? The partition produced by the Sequence-Congruence Algorithm cannot always provide an answer, since one sequence-congruence class may include several vertices from each program (i.e., the partition does not define a one-to-one correspondence). The HPR algorithm relies on editor-supplied tags; it is assumed that programs are created using a special editor that provides unique tags for newly inserted components, and maintains tags when components are moved within a program or when a copy of a program is made. Components with the same tag are considered to be the "same component" in different variants of the base program.

\(^{\text{Note that when the Sequence-Congruence Algorithm is applied to the second set of programs in Figure 1, the assignment statements to \text{area} in \text{Base} and in variant \( A \) are in different sequence congruence classes. To be able to discover they are sequence-congruent, we need one straightforward enhancement to the Sequence-Congruence Algorithm [Yang89]: For every constant \( c \) used in the program, we create a new variable \text{Const}_c and a new assignment statement \text{Const}_c \leftarrow c \) at the very beginning of the program and replace every use of \( c \) in the program with \text{Const}_c.}}\)

\(^{\text{Note that only non-\( \varphi \) vertices are classified. This will be explained further in Section 4.5.}}\)
The new program-integration algorithm also assumes that program components are tagged (tags may be provided by the editor, or may be supplied by some other mechanism—the source of the tags is not relevant to the algorithm itself). Given this assumption, the correspondence between components of the three programs is established as follows: Two components \( c_1 \) and \( c_2 \) correspond (or \( c_1 \) and \( c_2 \) are corresponding components) if and only if all of the following hold:

1. \( c_1 \) and \( c_2 \) are sequence-congruent;
2. \( c_1 \) and \( c_2 \) have the same tag;
3. if \( c_1 \) and \( c_2 \) are assignment statements, they assign to the same variable.

Corresponding components are considered the same components in different programs. That is, we can assign to each component an identity, which consists of three parts: its sequence-congruence class, its tag, and the variable that is assigned to at the vertex. Thus, two components correspond if and only if they have the same identity; hence, corresponding components are considered the same component in different versions of a program.

Using this definition of corresponding components, each non-\( \phi \) vertex of Base, \( A \), and \( B \) is classified as defined below.

Every non-\( \phi \) vertex in \( A \) is in one of five sets: \( \text{New}_A \), \( \text{Modified}_A \), \( \text{Modified}_B \), \( \text{Unchanged} \), or \( \text{Intermediate}_A \).

1. A vertex is in \( \text{New}_A \) if there is no corresponding vertex in Base. Vertices in \( \text{New}_A \) represent program components that have been added to Base to create \( A \), or have been moved to a context that has changed their execution behaviors.
2. A vertex is in \( \text{Modified}_A \) if there is a corresponding vertex in Base, but the vertex’s text in \( A \) differs from the text of the corresponding vertex in Base. Vertices in \( \text{Modified}_A \) represent components of \( A \) that have been textually changed but whose execution behaviors have not been changed.
3. A vertex is in \( \text{Modified}_B \) if there are corresponding vertices in both Base and \( B \), and the vertex’s text in \( A \) is the same as the text of the corresponding vertex in Base, but differs from the text of the corresponding vertex in \( B \).
4. A vertex is in \( \text{Intermediate}_A \) if there is a corresponding vertex in Base and the vertex’s text in \( A \) is the same as the text of the corresponding vertex in Base, but there is no corresponding vertex in \( B \) (either because the vertex was deleted from \( B \), or because the vertex’s execution behavior was changed, or because the vertex’s left-hand-side variable was changed).
5. A vertex is in \( \text{Unchanged} \) if there are corresponding vertices in both Base and \( B \), and all three vertices have the same text. Vertices in \( \text{Unchanged} \) represent components that are textually and behaviorally identical in all three programs.

Vertices in \( B \) are similarly classified into the sets \( \text{New}_B \), \( \text{Modified}_B \), \( \text{Modified}_A \), \( \text{Unchanged} \), and \( \text{Intermediate}_B \). Vertices in Base are similarly classified into the sets \( \text{Modified}_A \), \( \text{Modified}_B \), \( \text{Intermediate}_A \), \( \text{Intermediate}_B \), \( \text{Unchanged} \), and \( \text{Deleted} \). (A vertex in Base is in \( \text{Deleted} \) if neither \( A \) nor \( B \) contains a corresponding vertex. Vertices in \( \text{Deleted} \) represent program components of Base that have been deleted or whose left-hand-side variable and/or behavior have been changed in both \( A \) and \( B \).)

\(^5\)Since \( \phi \) statements are not part of the source program, they cannot be tagged by the editor. Their tags can, however, be generated systematically from the tags of the associated predicates and the names of the variables that are assigned to by the \( \phi \) statements.
Note that it is possible for a vertex in \( \text{New}_A \) to have a corresponding vertex in \( B \) that is in \( \text{New}_B \) and for a vertex in \( \text{Modified}_A \) to have a corresponding vertex in \( B \) that is in \( \text{Modified}_B \). For instance, consider the following three programs:

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Variant ( A )</th>
<th>Variant ( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt;T1&gt; \ x := 0 )</td>
<td>( &lt;T1&gt; \ x := 0 )</td>
<td>( &lt;T1&gt; \ x := 0 )</td>
<td></td>
</tr>
<tr>
<td>( &lt;T2&gt; \ y := x )</td>
<td>( &lt;T2&gt; \ y := 0 )</td>
<td>( &lt;T2&gt; \ y := 0 )</td>
<td></td>
</tr>
<tr>
<td>( &lt;T3&gt; \ z := x )</td>
<td>( &lt;T4&gt; \ x := 1 )</td>
<td>( &lt;T4&gt; \ x := 1 )</td>
<td></td>
</tr>
<tr>
<td>( &lt;T4&gt; \ x := 1 )</td>
<td>( &lt;T3&gt; \ z := x )</td>
<td>( &lt;T3&gt; \ z := 1 )</td>
<td></td>
</tr>
</tbody>
</table>

The assignment \( T3 \) in \( A \) is in \( \text{New}_A \) because the value assigned to \( z \) at \( T3 \) in \( A \) differs from that assigned to \( z \) at \( T3 \) in \( \text{Base} \); Similarly, the assignment \( T3 \) in \( B \) is in \( \text{New}_B \). However, the two assignment statements \( T3 \) in \( A \) and \( B \) correspond. The assignment \( T2 \) in \( A \) is in \( \text{Modified}_A \) because the two assignment statements \( T2 \) in \( A \) and \( \text{Base} \) produce the same value, have the same tag, and they assign to the same variable \( y \) but their texts differ. Similarly, the assignment \( T2 \) in \( B \) is in \( \text{Modified}_B \). The assignment \( T2 \) in \( \text{Base} \) is in both \( \text{Modified}_A \) and \( \text{Modified}_B \). The three assignment statements \( T2 \) in \( A, B, \) and \( \text{Base} \) correspond.

The classification process may discover that \( A \) and \( B \) interfere with respect to \( \text{Base} \) by identifying corresponding vertices \( \nu_A \) and \( \nu_B \) in \( A \) and \( B \), respectively, such that the text of \( \nu_A \) differs from the text of \( \nu_B \) and, if there is a corresponding vertex \( \nu_{\text{Base}} \) in \( \text{Base} \), the texts of \( \nu_A, \nu_B, \) and \( \nu_{\text{Base}} \) are pairwise distinct. Since a vertex in the merged PRG can have only one text, it is not possible to preserve the changed text of this component from both \( A \) and \( B \). This can occur either for a vertex in \( \text{New}_A \) (with a corresponding vertex in \( \text{New}_B \)) or for a vertex in \( \text{Modified}_A \) (with a corresponding vertex in \( \text{Modified}_B \)). In the example given above, the fact that the text of the assignment tagged \( T3 \) in \( B \) differs from the text of the assignment tagged \( T3 \) in \( A \) causes interference.

4.2. Computing Changed and Preserved Computations

The program produced by a successful integration must include the changed computations introduced by the variants as well as the computations of the base program that are preserved in both variants. The identification of changed and preserved computations is done differently in the HPR algorithm and the new integration algorithm.

4.2.1. Limited slices

In the HPR algorithm, two program components are assumed to have different execution behaviors if their slices are different. To ensure that an affected component included in the integrated program retains its behavior, the HPR algorithm includes in the integrated program the entire slice with respect to the affected component.

In contrast, the Sequence-Congruence Algorithm is able to identify behaviorally equivalent vertices that have unequal program slices. Therefore, an affected component's behavior can sometimes be retained in the integrated program without including its entire slice; only the "neighborhood" of the component is needed. This neighborhood is formalized as a limited slice.
Definition. Let $R$ be the Program Representation Graph of $Base$, $A$, or $B$, and let $S$ be a set of ($\phi$ and non-$\phi$) vertices in $R$. The limited slice of $R$ with respect to $S$, denoted by $R\cap S$, is defined as the smallest subgraph of $R$ such that if there is a path from a vertex $u$ to a vertex of $S$ and all non-$\phi$ vertices along this path, excluding the two endpoints, belong to either Intermediate, Intermediate, or Deleted, then all vertices and edges on this path are included in $R\cap S$.

It is easy to see that the limited slice with respect to a set of vertices is equivalent to the union of the limited slices with respect to the individual vertices.

4.2.2. Changed and preserved computations

The affected components of a variant are the components that are textually different from the corresponding components of $Base$, or that have no corresponding component in $Base$. The changed computations of a variant are computed by taking a limited slice of the variant with respect to its affected components. ($R_A$ denotes $A$'s PRG.)

$Affected_A = New_A \cup Modified_A$

$ChangedComps_A = R_A \cap Affected_A$

$Affected_B$ and $ChangedComps_B$ are defined similarly.

The preserved computations of $Base$, $A$, and $B$ are computed by examining the limited slices of the three programs with respect to the vertices $u$ in the set $Unchanged$. Note that these limited slices may not be equal8; although $u$ itself is behaviorally and textually identical in $Base$, $A$, and $B$, the values of the variables used at $u$ may be computed differently in the three programs. Interference is reported at this point if there is some vertex $u$ in $Unchanged$ such that the limited slices with respect to $u$ in $Base$, $A$, and $B$, are pairwise unequal. Otherwise, for each vertex $u \in Unchanged$, the preserved limited slice with respect to $u$, $Preserved(u)$, is determined as follows:

<table>
<thead>
<tr>
<th>Relationship of limited slices</th>
<th>Preserved $(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A \cap u = R_B \cap u$</td>
<td>$R_A \cap u$</td>
</tr>
<tr>
<td>$(R_A \cap u = R_{Base} \cap u)$ and $(R_A \cap u \neq R_B \cap u)$</td>
<td>$R_B \cap u$</td>
</tr>
<tr>
<td>$(R_B \cap u = R_{Base} \cap u)$ and $(R_B \cap u \neq R_A \cap u)$</td>
<td>$R_A \cap u$</td>
</tr>
<tr>
<td>$R_A \cap u$, $R_B \cap u$, and $R_{Base} \cap u$ are pairwise unequal</td>
<td>interference</td>
</tr>
</tbody>
</table>

The preserved computations, $Preserved$, is the union of $Preserved(u)$ for all $u \in Unchanged$.

$Preserved = \bigcup_{u \in Unchanged} Preserved(u)$

---

8Two limited slices are equal if there is an isomorphism under which related vertices correspond (i.e., related vertices have identical tags and left-hand-side variables, and are sequence-congruent).
4.3. Forming the Merged Graph

The merged graph, \( R_M \), is formed by taking the union of the graphs that represent the changed computations of \( A \) and \( B \), and the graphs that represent the preserved computations of \( Base, A, \) and \( B \):

\[
R_M = \text{ChangedComps}_A \cup \text{ChangedComps}_B \cup \text{Preserved}.
\]

For the purposes of this union, two vertices are "the same" (i.e., only one copy of the vertex is included in the merged graph) if and only if the two vertices correspond. It is possible that both \( \text{ChangedComps}_A \) and \( \text{ChangedComps}_B \) will include corresponding vertices with different text. This can only happen, however, if the two vertices are both classified \( \text{Modified}_A \) or both classified \( \text{Modified}_B \). In the former case, the text of the vertex incorporated in the merged graph is the text from \( A \); in the latter case, it is the text from \( B \). If vertices from the sets \( \text{New}_A \) and \( \text{New}_B \) have corresponding vertices in both \( A \) and \( B \), these vertices must have the same text; else interference would have been reported during vertex classification; if vertices from the sets \( \text{Modified}_A \) and \( \text{Modified}_B \) have corresponding vertices in both \( A \) and \( B \), these vertices must have the same text; else interference would have been detected during vertex classification; vertices from the set \( \text{Intermediate}_A \) cannot have corresponding vertices from \( B \) (and similarly for \( \text{Intermediate}_B \)); vertices from the set \( \text{Unchanged} \) have the same text in both \( A \) and \( B \); corresponding \( \phi \) vertices must have the same text.

4.4. Reconstituting a Program From the Merged Graph

The final step of the program integration algorithm is to determine whether the merged graph corresponds to some program, and if so, to produce the program. If the merged graph is infeasible (does not correspond to any program), the algorithm reports interference.

Determining whether a Program Dependence Graph is feasible has been shown to be NP-complete [Horwitz88a]; a similar result can be shown for Program Representation Graphs. The crux of the problem is to order each predicate's control children. A backtracking algorithm that operates on Program Dependence Graphs has been written and proved correct [Ball89]; this algorithm is easily adaptable to work on Program Representation Graphs. Although the algorithm is, in the worst case, exponential in the number of pairs of assignments to the same variable, we believe that it will be acceptable in practice.

Example. Figure 6 illustrates the new integration algorithm using the third set of example programs in Figure 1. Figure 6 shows the sets of vertices \( \text{Affected}_A \), \( \text{Affected}_B \), and \( \text{Unchanged} \); the graph fragments \( \text{ChangedComps}_A \), \( \text{ChangedComps}_B \), and \( \text{Preserved} \); and the merged graph. This merged graph is feasible, and corresponds to the program shown in Figure 1 as the result of integrating the third set of programs.

4.5. Discussion of Classification of Vertices

In Section 4.1, we mentioned that only non-\( \phi \) vertices are classified into the categories \( \text{New}, \text{Modified}, \text{Intermediate}, \text{Unchanged}, \) and \( \text{Deleted} \). The reason \( \phi \) vertices are not classified in these categories is because in a (feasible) \( \text{PRG} \) \( \phi \) vertices exist only if they have flow successors. If \( \phi \) vertices are treated in the same way as non-\( \phi \) vertices, the merged graph may not be a feasible \( \text{PRG} \) even if there is no interference. For instance, consider the example in Figure 7 (the \( \phi \) vertices are shown explicitly).

In Figure 7, if \( \phi \) vertices were treated in the same way as non-\( \phi \) vertices, the \( \phi \) vertices would be classified as \( \text{Unchanged} \) and would be included in \( \text{Preserved} \) and the merged graph would be as in \( M_1 \), which is not a feasible \( \text{PRG} \) (because the \( \phi \) vertex in \( M_1 \) has no flow successor). Thus, a false interference would be reported in this case. However, our new program integration algorithm will successfully produce the integrated program \( M_2 \).
Figure 6. The new integration algorithm is illustrated using the third set of example programs from Figure 1. Note the absence of any incoming control edge to vertex \( \text{rad} := 2 \) in \textit{Preserved}. 
<table>
<thead>
<tr>
<th>Base</th>
<th>Variant A</th>
<th>Variant B</th>
<th>M 1</th>
<th>M 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>program</strong></td>
<td><strong>program</strong></td>
<td><strong>program</strong></td>
<td><strong>program</strong></td>
<td><strong>program</strong></td>
</tr>
<tr>
<td>$x := 1$</td>
<td>$x := 1$</td>
<td>$x := 1$</td>
<td>$x := 1$</td>
<td>$x := 1$</td>
</tr>
<tr>
<td>If $P$ then</td>
<td>If $P$ then</td>
<td>If $P$ then</td>
<td>If $P$ then</td>
<td>If $P$ then</td>
</tr>
<tr>
<td>$x := 2$</td>
<td>$x := 2$</td>
<td>$x := 2$</td>
<td>$x := 2$</td>
<td>$x := 2$</td>
</tr>
<tr>
<td>fi</td>
<td>fi</td>
<td>fi</td>
<td>fi</td>
<td>fi</td>
</tr>
<tr>
<td>$\phi: x := x$</td>
<td>$\phi: x := x$</td>
<td>$\phi: x := x$</td>
<td>$\phi: x := x$</td>
<td>end</td>
</tr>
<tr>
<td>$y := x + 2$</td>
<td>$y := x + 2$</td>
<td>$y := x + 3$</td>
<td>$z := x + 3$</td>
<td>end</td>
</tr>
<tr>
<td>$z := x + 3$</td>
<td>end($y$, $z$)</td>
<td>end($z$)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7. An integration example that demonstrates why $\phi$ vertices are not classified to avoid certain interference.

5. THE INTEGRATION THEOREM

As with the HPR algorithm, we can prove a theorem for the new integration algorithm about how the execution behavior of the integrated program relates to the execution behaviors of the programs that are the arguments to the merge. The theorem asserts that when the new integration algorithm successfully integrates the variant programs (with respect to the base program), the merged program preserves the changed behaviors of the variants as well as the behaviors common to all three.

**Theorem.** (Integration Theorem). If $A$ and $B$ are two variants of $Base$ for which the new integration algorithm succeeds (and produces a merged program $M$), then for any initial state $\sigma$ on which $A$, $B$, and $Base$ all terminate normally:

1. $M$ terminates normally on $\sigma$.
2. For any program component $v_A$ in $A$, if $v_A \in \text{Affected}_A$ then there is a program component $v$ in $M$ such that $v$ and $v_A$ produce the same sequence of values during the respective executions of $M$ and $A$ on $\sigma$.
3. For any program component $v_B$ in $B$, if $v_B \in \text{Affected}_B$ then there is a program component $v$ in $M$ such that $v$ and $v_B$ produce the same sequence of values during the respective executions of $M$ and $B$ on $\sigma$.
4. For any program component $v_{Base}$ in $Base$, if $v_{Base} \in \text{Unchanged}$ then there is a program component $v$ in $M$ such that $v$ and $v_{Base}$ produce the same sequence of values during the respective executions of $M$ and $Base$ on $\sigma$.

Note that this theorem meets (and generalizes) the semantic integration criterion stated in the Introduction. For example, if there is a variable $x$ whose final value after executing $A$ on $\sigma$ differs from its final value after executing $Base$ on $\sigma$, then (1) there is a final-use vertex $v_A$ for variable $x$ in $A$, and (2) $v_A \in \text{Affected}_A$. Thus, $x$'s final value after executing $M$ on $\sigma$ is equal to the value of $x$ after executing $A$ on $\sigma$.

In addition to the final values of variables, the Theorem also asserts that, if the new integration algorithm successfully produces a merged program, the changed behaviors of program components are preserved in the merged program. That is, if a component $c$ in a variant behaves differently from the component in $Base$ that has the same tag as $c$ (if such a component exists in $Base$), then $c$'s behavior will be preserved in the
merged program.

In addition to behavioral changes, the new integration algorithm also attempts to preserve textual changes: \( \text{Affected}_A \) and \( \text{Affected}_B \) include those components whose texts, rather than execution behaviors, have been changed (i.e., components in the sets \( \text{Modified}_A \) and \( \text{Modified}_B \)) and the limited slices with respect to components in \( \text{Modified}_A \) and \( \text{Modified}_B \) are always included in the merged graph. Thus, textual changes made in \( A \) and \( B \) are also preserved in the merged program when the new integration algorithm successfully produces a merged program.

In what follows, we use \( R_A, R_B, R_{Base} \), and \( R_M \) to denote the respective program representation graphs of \( A, B, \) Base, and \( M \). Every (ϕ or non-ϕ) vertex \( v \) of \( R_M \) is taken from either \( R_A \) or \( R_B \) or both (it is possible that \( v \) appears in \( R_{Base} \) as well); this vertex in \( R_A \) or \( R_B \) is called an originating vertex of \( v \). A vertex \( v \) of \( R_M \) inherits an “identity” from its originating vertices. ("Identity" is based on tag, left-hand-side variable, and sequence-congruence class, but not the entire text in the vertex. A vertex \( v \) in \( R_M \) may have a different text from one of its originating vertices, although the text of \( v \) must match one of its originating vertices.) Modulo their having different texts, \( v \) and its originating vertices can be considered to be the same vertex in different graphs. Note that, by the construction of \( R_M \), if both \( v_1 \) and \( v_2 \) are originating vertices of \( v \), then \( v_1 \) and \( v_2 \) must be corresponding vertices (in particular, \( v_1 \) and \( v_2 \) must be sequence-congruent).

Similarly, every edge \( u \rightarrow v \) of \( R_M \) is taken from either \( R_A \) or \( R_B \) or both. (It is possible that the edge \( u \rightarrow v \) appears in \( R_{Base} \) as well.) Since each control or flow dependence edge is identified by its two endpoints, when we say an edge \( u \rightarrow v \) of \( R_M \) is taken from \( R_A \) (or \( R_B \)), we mean that there are originating vertices \( u' \) and \( v' \) of \( u \) and \( v \), respectively, and an identical control or flow dependence edge \( u' \rightarrow v' \) in \( R_A \) (or \( R_B \), respectively). It can be shown (by cases on the classification of \( v' \)) that \( v' \) and \( v \) have the same text.

The proof of the Integration Theorem proceeds by considering the sequence-congruence classes formed when the Sequence-Congruence Algorithm is applied to \( R_M \) together with \( R_A, R_B, \) and \( R_{Base} \). We show that every vertex of \( R_M \) is placed in the same sequence-congruence class as its originating vertices; the Integration Theorem then follows from the Sequence-Congruence Theorem and the fact that, by the construction of \( R_M \), every vertex in \( \text{Affected}_A, \text{Affected}_B, \) and \( \text{Unchange} \) is an originating vertex of some vertex in \( R_M \).

Recall that the Sequence-Congruence Algorithm consists of two partitioning passes. A key observation about the Sequence-Congruence Algorithm is that each pass can be decomposed into repeated phases. In each phase we consider only edges of a single type. For instance, in the first phase of the first pass, we use only edges of type 1 to perform partitioning; in the second phase, we use only edges of type 2, etc. After all types of edges (except control dependence edges) have been considered in separate phases, edges of type 1 are taken into account again in a new phase. This process is repeated again and again until a stable partition is reached. The second pass of partitioning is performed in a similar way, except that only control dependence edges are considered during partitioning.

We use \( R^i \) to denote the subgraph of \( R \) obtained by retaining only edges of type \( i \) in the program representation graph \( R \). For each type \( i \), if we ignore the control dependence edge from a while predicate to itself, \( R^i \) is a forest because there is no cycle in \( R \) that consists of edges of a single type \( i \), and there is at most one incoming edge of type \( i \) for any vertex in \( R \). We use \( \text{root}(v, R^i) \) to denote the root of the tree that contains \( v \) in \( R^i \). We use \( \text{level}(v, R^i) \) to denote the length of the path from \( \text{root}(v, R^i) \) to \( v \) in \( R^i \).

Based on the above observation, the following lemma asserts that when the Sequence-Congruence Algorithm is applied to \( R_A, R_B, R_{Base}, \) and \( R_M \) simultaneously, every vertex of \( R_M \) is sequence-congruent to its originating vertices.

**LEMMA.** If \( A \) and \( B \) are two variants of Base for which the new integration algorithm succeeds (and produces a merged program \( M \)), then every vertex of \( R_M \) is sequence-congruent to its originating vertices.
PROOF. We use the above repeated phases to partition $R_A, R_B, R_{Base}$, and $R_M$. Based on the above observation, it suffices to show that every vertex in $R_M$ is in the same class as its originating vertices at the end of every phase of both partitioning passes.

First we show that every vertex in $R_M$ is in the same class as its originating vertices in the initial partition. Suppose $v$ is a vertex in $R_M$ and $v'$ is its originating vertex in $R_A$ or $R_B$ whose text is the same as $v$ ($v'$ must exist because the text of $v$ is taken from one of its originating vertices). Without loss of generality, assume $v'$ is in $R_A$.

If $v$ is not a simple vertex, then since the texts in $v$ and $v'$ are the same, $v$ and $v'$ are in the same class in the initial partition. If there is another vertex $v''$ in $B$ or $Base$ that is also an originating vertex of $v$, then $v'$ and $v''$ are corresponding vertices, which means that $v'$ and $v''$ are always in the same class. Hence $v$ and $v''$ are also in the same initial class. Therefore, every non-simple vertex $v$ of $R_M$ is in the same class as its originating vertices in the initial partition.

If $v$ is a simple vertex, let $u$ be a non-simple vertex in $R_M$ such that there is a flow dependence path $u \rightarrow_f v$ in $R_M$ and all vertices on this path except $u$ are simple assignment vertices (i.e., statements of the form $x := y$). We prove by induction over the length of the flow dependence path $u \rightarrow_f v$ that $v$ and $v'$ are in the same initial class. (Induction is needed here because the flow dependence path $u \rightarrow_f v$ in $R_M$ may not be entirely from $R_A$ nor entirely from $R_B$.)

**Base case.** Suppose the length of the flow dependence path $u \rightarrow_f v$ is 1. If the edge $u \rightarrow_f v$ in $R_M$ is taken from $R_A$, then there exists an identical edge $u' \rightarrow_f v'$ in $R_A$ such that $u'$ and $v'$ are originating vertices of $u$ and $v$, respectively. Since $u'$ is an originating vertex of $u$ and $u$ is not a simple vertex, $u$ and $u'$ are in the same class in the initial partition, as shown above. Because by hypothesis $v$ and $v'$ have the same text, $v'$ is simple; therefore $v'$ and $u'$ are in the same initial class. Because vertices $u'$ and $u$ are in the same initial class and because $u$ and $v$ are in the same initial class, $v'$ and $v$ are in the same initial class.

If the edge $u \rightarrow_f v$ in $R_M$ is taken from $R_B$, then there exists an identical edge $u_B \rightarrow_f v_B$ in $R_B$ such that $u_B$ and $v_B$ are originating vertices of $u$ and $v$, respectively; note that because the edge $u \rightarrow v$ was taken from $B$, vertices $v_B$ and $v$ have the same text. By the same argument as in the previous paragraph, $v$ and $v_B$ are in the same initial class. Because (1) $v$ and $v_B$ are in the same initial class and (2) $v_B$ and $v'$ are corresponding vertices, $v$ and $v'$ are in the same initial class.

In either case, $v$ and $v'$ are in the same class in the initial partition.

**Induction step.** Suppose the length of the flow dependence path $u \rightarrow_f v$ is $n$ for some $n > 1$. Let $w$ be the immediate flow predecessor of $v$. If the edge $w \rightarrow_f v$ in $R_M$ is taken from $R_A$, then there exists an identical edge $w' \rightarrow_f v'$ in $R_A$ such that $w'$ and $v'$ are originating vertices of $w$ and $v$, respectively. Since $w'$ is an originating vertex of $w$ and the length of the flow dependence path $u \rightarrow_f w$ is $n - 1$, by the induction hypothesis, $w$ and $w'$ must be in the same initial class. By assumption, $v$ and $v'$ have the same text, thus $v'$ is simple and $w'$ and $v'$ are in the same initial class. Because (1) $v$ and $w$ are in the same initial class, (2) $w$ and $w'$ are in the same initial class, and (3) $w'$ and $v'$ are in the same initial class, $v$ and $v'$ are in the same initial class.

If the edge $w \rightarrow_f v$ in $R_M$ is taken from $R_B$, then there exists an identical edge $w_B \rightarrow_f v_B$ in $R_B$ such that $w_B$ and $v_B$ are originating vertices of $w$ and $v$, respectively. Since $w_B$ is an originating vertex of $w$ and the length of the flow dependence path $u \rightarrow_f w$ is $n - 1$, by the induction hypothesis, $w$ and $w_B$ are in the same initial class. Note that because the edge $w \rightarrow v$ was taken from $B$, vertices $v$ and $v_B$ have the same text. That is, both $v$ and $v_B$ are simple vertices. Because (1) $v$ and $w$ are in the same initial class, (2) $w$ and $w_B$ are in the same initial class, and (3) $w_B$ and $v_B$ are in the same initial class, we know that $v$ and $v_B$ are in the same initial class. Because (1) $v$ and $v_B$ are in the same initial class and (2) $v_B$ and $v'$ are corresponding
vertices, \(v\) and \(v'\) are in the same initial class.

In either case, \(v\) and \(v'\) are in the same initial class. This completes the induction.

If there is another vertex \(v''\) in \(B\) or \(Base\) that is also an originating vertex of \(v\), then \(v'\) and \(v''\) are corresponding vertices, which means that \(v'\) and \(v''\) are always in the same class. Hence \(v\) and \(v''\) are also in the same initial class. Therefore, every simple vertex \(v\) of \(R_M\) is in the same class as its originating vertices in the initial partition.

We conclude that every vertex, simple or non-simple, of \(R_M\) is in the same class as its originating vertices in the initial partition.

Next we want to show that every vertex of \(R_M\) stays in the same class as its originating vertices at the end of each phase of both partitioning passes.

Fix a pass and a phase of the pass. Let \(i\) be the type of edge under consideration during this phase. We want to prove that if every vertex of \(R_M\) is in the same class as its originating vertices at the beginning of this phase, then every vertex of \(R_M\) is still in the same class as its originating vertices at the end of this phase.

Suppose \(v\) is a vertex in \(R_M\) and \(v'\) is its originating vertex in \(R_A\) or \(R_B\) whose text is the same as \(v\). Because the texts in \(v\) and \(v'\) are the same, \(v\) and \(v'\) have the same number of incoming flow and control dependence edges. In particular, \(v\) has an incoming edge of type \(i\) if and only if \(v'\) has an incoming edge of the same type. We prove by induction over \(\text{level}(v, R_M^i)\) that, if every vertex of \(R_M\) is in the same class as its originating vertices at the beginning of this phase, then every vertex of \(R_M\) is still in the same class as its originating vertices at the end of this phase (it is sufficient to show that \(v\) and \(v'\) are still in the same class at the end of this phase.)

**Base case.** Suppose \(\text{level}(v, R_M^i) = 0\); that is, either \(v\) has no incoming edge of type \(i\) in \(R_M\) or edges of type \(i\) are the self-loops on while predicates. First assume \(v\) has no incoming edge of type \(i\) in \(R_M\). Because \(v\) and \(v'\) must have the same incoming edges, \(v'\) has no incoming edge of type \(i\). By assumption, \(v\) and \(v'\) were in the same class at the beginning of this phase. Because they have no incoming edges of type \(i\), they cannot be separated during this phase. Therefore, \(v\) and \(v'\) are still in the same class at the end of this phase.

Next assume that edges of type \(i\) are the self-loops on while predicates. In this case, both \(v\) and \(v'\) are while predicate vertices. By assumption, \(v\) and \(v'\) were in the same class at the beginning of this phase. Because \(v\) and \(v'\) both have self-loops, they cannot be separated during this phase. Therefore, \(v\) and \(v'\) are still in the same class at the end of this phase.

In either case, \(v\) and \(v'\) are still in the same class at the end of this phase.

**Induction step.** Suppose that \(\text{level}(v, R_M^i) = n\) for some \(n > 0\). Our induction hypothesis is that if every vertex of \(R_M\) is in the same class as its originating vertices at the beginning of this phase, then, for all vertices \(u\) with \(\text{level}(u, R_M^i) < n\), \(u\) is still in the same class as its originating vertices at the end of this phase.

Since \(v\) has an incoming edge of type \(i\), so does \(v'\). Let \(u\) be the immediate predecessor of \(v\) in \(R_M^i\). Due to the construction of \(R_M\), the edge \(u \rightarrow v\) is taken either from \(R_A\) or from \(R_B\).

First assume the edge \(u \rightarrow v\) in \(R_M\) is taken from \(R_A\). Thus, there is an identical edge \(u' \rightarrow v'\) in \(R_A\). Because \(u'\) is an originating vertex of \(u\), by assumption, \(u\) and \(u'\) are in the same class at the beginning of this phase. Because there is an edge \(u \rightarrow v\) in \(R_M^i\), \(\text{root}(u, R_M^i)\) is the same vertex as \(\text{root}(v, R_M^i)\) and \(\text{level}(u, R_M^i)\) is \(n-1\). Because \(\text{level}(u, R_M^i) = n-1\), by the induction hypothesis, \(u\) and \(u'\) are still in the same class at the end of this phase. Because \(u\) and \(u'\) are always in the same class during this phase, \(v\) and \(v'\) are in the same class at the end of this phase.
Next assume the edge \( u \rightarrow v \) in \( R_M \) is taken from \( R_B \). Thus, there is an identical edge \( u'' \rightarrow v'' \) in \( R_B \). By the same argument as in the previous paragraph, \( v \) and \( v'' \) are in the same class at the end of this phase. Because \( v' \) and \( v'' \) are sequence-congruent, \( v' \) and \( v'' \) are always in the same class during partitioning. Since \( v \) and \( v'' \) are in the same class and \( v'' \) and \( v' \) are always in the same class, \( v \) and \( v' \) are in the same class at the end of this phase.

In either case, we conclude that \( v \) and \( v' \) are in the same class at the end of this phase. This completes the induction.

If there is another vertex \( v'' \) in \( B \) or \( Base \) that is also an originating vertex of \( v \), then \( v' \) and \( v'' \) are corresponding vertices, which means that \( v' \) and \( v'' \) are always in the same class. Hence \( v \) and \( v'' \) are also in the same class at the end of this phase.

We have proved that (1) every vertex of \( R_M \) is in the same class as its originating vertices in the initial partition and (2) if every vertex of \( R_M \) is in the same class as its originating vertices at the beginning of a phase, then every vertex of \( R_M \) is still in the same class at its originating vertices at the end of the phase. Therefore, every vertex of \( R_M \) is in the same equivalence class as its originating vertices when the Sequence-Congruence Algorithm terminates; that is, every vertex of \( R_M \) is sequence-congruent to its originating vertices. \( \Box \)

**Theorem.** (Integration Theorem). If \( A \) and \( B \) are two variants of \( Base \) for which the new integration algorithm succeeds (and produces a merged program \( M \)), then for any initial state \( \sigma \) on which \( A \), \( B \), and \( Base \) all terminate normally:

1. \( M \) terminates normally on \( \sigma \).
2. For any program component \( v_A \) in \( A \), if \( v_A \in \text{Affected}_A \) then there is a program component \( v \) in \( M \) such that \( v \) and \( v_A \) produce the same sequence of values during the respective executions of \( M \) and \( A \) on \( \sigma \).
3. For any program component \( v_B \) in \( B \), if \( v_B \in \text{Affected}_B \) then there is a program component \( v \) in \( M \) such that \( v \) and \( v_B \) produce the same sequence of values during the respective executions of \( M \) and \( B \) on \( \sigma \).
4. For any program component \( v_{Base} \) in \( Base \), if \( v_{Base} \in \text{Unchanged} \) then there is a program component \( v \) in \( M \) such that \( v \) and \( v_{Base} \) produce the same sequence of values during the respective executions of \( M \) and \( Base \) on \( \sigma \).

**Proof.** Note that every vertex in \( \text{Affected}_A \), \( \text{Affected}_B \), and \( \text{Unchanged} \), is an originating vertex of some vertex of \( R_M \). This is because for each vertex \( v \) in these classes, either \( R_A/v \) or \( R_B/v \) is included in \( R_M \). Thus, we only need to show that \( M \) terminates normally on \( \sigma \). The remaining assertions of the theorem follow directly from the previous lemma and the Sequence-Congruence Theorem.

Suppose \( M \) does not terminate normally on \( \sigma \). Then either there is a non-terminating loop or a fault such as division by zero occurs during the execution of \( M \).

First suppose a fault occurs during the execution of \( M \). Let \( u \) be the component where the fault occurs. By the construction of \( R_M \), \( u \) must have an originating vertex in either \( R_A \) or \( R_B \). Without loss of generality, assume \( u \) has an originating vertex \( u_A \) in \( R_A \). By the previous lemma, \( u \) and \( u_A \) are sequence-congruent. Since \( A \) terminates normally on the initial state \( \sigma \) but \( M \) does not, by the Sequence-Congruence Theorem, the sequence of values produced at \( u \) is an initial sub-sequence of the sequence of values produced at \( u_A \). In particular, the fault value occurs as the last element of the sequence of values produced at \( u \); thus, the fault value must be in the sequence of values produced at \( u_A \). The presence of this value means that, in fact, \( A \) does not terminate normally on the initial state \( \sigma \), which contradicts the assumption that \( A \) ter-
minates normally. Therefore, no fault can occur during the execution of $M$.

Next suppose there is a non-terminating loop during the execution of $M$. Let $u$ be the predicate of the non-terminating loop. Without loss of generality assume $u$ is taken from $R_A$; that is, $u$ has an originating vertex $u_A$ in $R_A$. By the previous lemma, $u$ and $u_A$ are sequence-congruent. Since $A$ terminates normally on $\sigma$ but $M$ does not, by the Sequence-Congruence Theorem, the sequence of values produced at $u$ is an initial sub-sequence of the sequence of values produced at $u_A$. Because $A$ terminates normally, the sequence of values produced at $u_A$ is finite. Therefore, the sequence of values produced at $u$ is also finite. Thus, the loop of $u$ cannot execute an infinite number of iterations, which contradicts the assumption that $u$ is the predicate of a non-terminating loop. Therefore, there cannot be a non-terminating loop in $M$.

Because no fault can occur during the execution of $M$ and because there cannot be a non-terminating loop in $M$, $M$ terminates normally on the initial state $\sigma$. $\square$

6. COMPARISON WITH THE HPR ALGORITHM

The HPR program-integration algorithm [Horowitz88, Horowitz89] operates on Program Dependence Graphs (PDGs) rather than Program Representation Graphs (PRGs). Since PDGs and PRGs are very similar in nature, it is possible to modify the HPR algorithm to operate on PRGs and to show that the modified algorithm is equivalent to the HPR algorithm [Yang90]. The comparison made in this section is based on the modified algorithm rather than the original HPR algorithm [Horowitz88, Horowitz89].

In this section, we first describe the modified HPR algorithm that operates on PRGs. Since the HPR algorithm makes use of program slices, Section 6.1 demonstrates how slices can be extracted from PRGs and gives a characterization of program slicing. The modified HPR algorithm, presented in Section 6.2, is a straightforward translation of the original HPR algorithm; the only difference is that it uses PRGs instead of PDGs. In Section 6.3, we compare the new program-integration algorithm with the modified HPR algorithm. We are able to show that, given the same set of component tags, whenever the HPR algorithm succeeds in integrating a base program and a set of variants, the new integration algorithm will also succeed, and will produce a program whose execution behavior has the same characterization as the one produced by the HPR algorithm.

6.1. Feasibility Lemma for Program Representation Graphs

The HPR integration algorithm makes use of slices of Program Dependence Graphs. In order to modify the HPR algorithm to work on Program Representation Graphs, we first define slices of Program Representation Graphs.

Definition. A slice of a Program Representation Graph $R$ with respect to a set of ($\phi$ and non-$\phi$) vertices $S$, denoted by $R/S$, is the subgraph of $R$ induced by all vertices that can reach an element of $S$ via a path of control and/or flow dependence edges.

Note that a slice of $R$ with respect to a vertex that does not appear in $R$ is the empty graph. A slice of the example program of Figure 2 is shown in Figure 8. The slice is taken with respect to the statement "$\text{rad} := 4.$"

We say a graph is a feasible PRG if it is the PRG of some program. It has been shown in [Reps88] that a slice of a PDG is a feasible PDG. For the same result to hold for PRGs, it is necessary to impose the restriction that the slice be taken with respect to a set of non-$\phi$ vertices [Yang90].

Lemma. (Feasibility Lemma for Program Representation Graphs [Yang90]). For any program $P$, if $R_Q$ is the slice of $P$'s Program Representation Graph with respect to a set of non-$\phi$ vertices, then $R_Q$ is a feasible Program Representation Graph.
Figure 8. A slice of the program representation graph shown in Figure 2(c). The slice is taken with respect to the statement \( \text{rad} := 4 \).

6.2. The Modified HPR Algorithm

The HPR integration algorithm operates on Program Dependence Graphs rather than Program Representation Graphs. In order to compare the two integration algorithms, the HPR algorithm is modified to work on Program Representation Graphs. The modified integration algorithm is a straightforward translation of the HPR algorithm. It takes as input a base program \( \text{Base} \), and two variant programs \( A \) and \( B \). Whenever the changes made to \( \text{Base} \) to create \( A \) and \( B \) do not “interfere” (as defined below), the modified algorithm produces a merged program \( M \) that incorporates the changed computation threads of \( A \) and \( B \) as well as the preserved computation thread common to all three versions. We have shown that the HPR and the modified integration algorithms are equivalent in the sense that they produce equivalent merged programs or they both report that there is interference [Yang90].

There are three steps in the modified algorithm. The first step determines slices that represent safe approximations to the changed computation threads of \( A \) and \( B \) and the computation threads of \( \text{Base} \) preserved in both \( A \) and \( B \); the second step combines these slices to form the merged graph \( R_M \); the third step tests \( R_M \) for interference.

\textbf{Step 1: Determining changed and preserved computation threads}

If the slice of variant \( R_A \) at non-\( \Phi \) vertex \( v \) is not identical to the slice of \( R_{\text{Base}} \) at \( v \), then \( R_A \) and \( R_{\text{Base}} \) may compute different values at \( v \). In other words, vertex \( v \) is a site that potentially exhibits different behavior in the two programs. Thus, we define the \textit{affected components} of \( R_A \) with respect to \( R_{\text{Base}} \), denoted by \( AF_{A, \text{Base}} \), to be the subset of non-\( \Phi \) vertices of \( R_A \) whose slices in \( R_{\text{Base}} \) and \( R_A \) are not identical: \( AF_{A, \text{Base}} = \{ v : a \text{ non-}\Phi \text{ vertex in } R_A \mid R_{\text{Base}} / v \not= R_A / v \} \). \( AF_{B, \text{Base}} \) is defined similarly. It follows that the slices \( R_A / AF_{A, \text{Base}} \) and \( R_B / AF_{B, \text{Base}} \) capture the respective changed computation threads of \( A \) and \( B \).

The \textit{preserved components} common to \( A \), \( B \), and \( \text{Base} \), denoted by \( PR_{A, B, \text{Base}} \), are those non-\( \Phi \) vertices whose slices in \( R_A \), \( R_B \), and \( R_{\text{Base}} \) are identical: \( PR_{A, B, \text{Base}} = \{ v : a \text{ non-}\Phi \text{ vertex in } R_{\text{Base}} \mid R_A / v = R_B / v = R_{\text{Base}} / v \} \). Thus, the preserved computation thread common to \( A \), \( B \), and \( \text{Base} \) is captured by the program slice that appears in all three: \( R_{\text{Base}} / PR_{A, B, \text{Base}} \).
Step 2: Forming the merged graph

The merged graph, \( R_M \), is formed by unioning the three slices that represent the changed and the preserved computation threads:

\[
R_M = (R_A / AF_{A, Base}) \cup (R_B / AF_{B, Base}) \cup (R_{Base} / PR_{A, B, Base}).
\]

Step 3: Testing for interference

There are two possible ways by which the graph \( R_M \) may fail to represent a satisfactory integrated program; both types of failure are referred to as "interference." The first interference criterion is based on a comparison of slices of \( R_A \), \( R_B \), and \( R_M \). The slices \( R_A / AF_{A, Base} \) and \( R_B / AF_{B, Base} \) represent the changed computation threads of programs \( A \) and \( B \) with respect to \( Base \). \( A \) and \( B \) interfere if \( R_M \) does not preserve these slices; that is, the merged graph \( R_M \) must satisfy the following two equations: \( R_M / AF_{A, Base} = R_A / AF_{A, Base} \) and \( R_M / AF_{B, Base} = R_B / AF_{B, Base} \).

The second interference criterion arises because the merged graph may not be feasible; if the graph is infeasible, \( A \) and \( B \) interfere. As discussed in Section 4.4, determining whether a graph is a feasible \( PRG \) is an NP-complete problem.

If neither kind of interference occurs, one of the programs whose \( PRGs \) are identical to the merged graph \( R_M \) is returned as the result of the integration operation.

6.3. Comparison Theorem

As discussed in the Introduction, whenever the HPR algorithm succeeds in integrating a base program and a set of variants, the execution behavior of the integrated program can be characterized in terms of the behaviors of the base program and the variants. Given the same set of component tags, the new integration algorithm will also succeed, and will produce a program whose execution behavior has the same characterization.

However, for the same argument programs it is possible for the two algorithms to produce different integrated programs. This situation is illustrated by the following integration example.

<table>
<thead>
<tr>
<th>Base</th>
<th>Variant A</th>
<th>Variant B</th>
<th>Integrated Program Produced by the HPR Algorithm</th>
<th>Integrated Program Produced by the New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := 1 )</td>
<td>( x := 1 )</td>
<td>( x := 1 )</td>
<td>( x := 1 )</td>
<td>( x := 1 )</td>
</tr>
<tr>
<td>( y := x+2 )</td>
<td>( w := x+2 )</td>
<td>( y := x+2 )</td>
<td>( w := x+2 )</td>
<td>( w := x+2 )</td>
</tr>
<tr>
<td>( z := y+3 )</td>
<td>( z := y+3 )</td>
<td>( z := y+3 )</td>
<td>( z := y+3 )</td>
<td>( z := y+3 )</td>
</tr>
<tr>
<td>end(x)</td>
<td>end(x)</td>
<td>end(x)</td>
<td>end(x)</td>
<td>end(x)</td>
</tr>
</tbody>
</table>

The discrepancy between the two integrated programs is due to the assignment to \( z \) in variant \( A \). The assignment to \( z \) in \( A \) is considered to be an affected component by the HPR algorithm because the slice with respect to this assignment in \( A \) is not equal to its counterpart in \( Base \). Therefore, the assignment is included in the integrated program by the HPR algorithm. However, the Sequence-Congruence Algorithm discovers that the execution behaviors of the respective assignments in \( A \) and \( Base \) are, in fact, the same.
This assignment is, therefore, not an affected component of \( A \). This assignment statement is not a preserved component because it has been deleted in \( B \). Because no affected components depend on this assignment to \( x \) in \( A \), this assignment is not included in the integrated program produced by the new algorithm.

Although the two integration algorithms may produce different results even in cases where both succeed, it can be shown that the program produced by the new algorithm is always a slice of the program produced by the HPR algorithm. This is stated as the following Comparison Theorem.

**Theorem. (Comparison Theorem).** When the HPR algorithm successfully integrates \( A, B, \) and Base, the new algorithm also succeeds and the integrated program produced by the new algorithm is a slice of the integrated program produced by the HPR algorithm.

In this section, we use \( R_A, R_B, \) and \( R_{\text{Base}} \) to denote the respective Program Representation Graphs of \( A, B, \) and Base. We use \( R_{\text{old}} \) and \( R_{\text{new}} \) to denote the respective merged graphs produced by the modified HPR algorithm and the new algorithm.

The HPR algorithm requires that (1) tags be unique within a given program variant and (2) if vertices in different variants of a program have the same tag, then they also have the same texts. Since the two integration algorithms should be compared under the same conditions, both conditions will also be assumed in our discussion of the new integration algorithm in this section. In particular, vertices with the same tag will always have the same text, and hence the sets Modified\(_A\) and Modified\(_B\) in the new integration algorithm are always empty. From now on, issues about the text associated with a vertex will be ignored.

The two integration algorithms use different methods for establishing a correspondence among program components. In particular, vertices that have the same tag but are not sequence-congruent are corresponding vertices under the HPR algorithm, but not under the new algorithm. In order to clarify this difference, we first prove Lemma 6.1, which shows that when \( A, B, \) and Base can be integrated by the HPR algorithm, there can be at most one vertex in \( R_{\text{new}} \) with a given tag. Again under the assumption that \( A, B, \) and Base can be integrated by the HPR algorithm, Lemma 6.2 shows that \( R_{\text{new}} \) is a subgraph of \( R_{\text{old}} \) and Lemma 6.3 shows that \( R_{\text{new}} \) is a slice of \( R_{\text{old}} \). The proof of the Comparison Theorem follows from Lemma 6.3 and the Feasibility Lemma.

**Lemma 6.1.** Suppose \( A, B, \) and Base can be integrated by the HPR algorithm. Then there is at most one vertex with a given tag in \( R_{\text{new}} \).

**Proof.** First we have to show that when \( A, B, \) and Base can be integrated by the HPR algorithm, the new integration algorithm will produce a merged graph. That is, the new integration algorithm will not report interference in step (2) or in step (3) (see Section 4).

Interference in step (2) is due to conflicting text in corresponding components. However, we have already assumed, for the purposes of this section, that components with the same tag always have the same text. Thus, interference due to conflicting text will not happen. Interference in step (3) can happen only when there is a component \( u \in \text{Unchanged} \) such that \( R_A/\!\!/u, R_B/\!\!/u, \) and \( R_{\text{Base}}/\!\!/u \) are pairwise unequal. However, if \( R_A/\!\!/u, R_B/\!\!/u, \) and \( R_{\text{Base}}/\!\!/u \) are pairwise unequal, then \( R_A/\!\!/u, R_B/\!\!/u, \) and \( R_{\text{Base}}/\!\!/u \) are pairwise unequal. Thus, the HPR algorithm will also report interference, which contradicts the assumption that \( A, B, \) and Base can be integrated by the HPR algorithm. Thus, interference in step (3) cannot happen either.

Therefore, the new integration algorithm will produce a merged graph \( R_{\text{new}} \).

We are assuming that two vertices with the same tag have identical text. Thus, when \( R_{\text{new}} \) is created — by the union of three subgraphs — two vertices in these different graphs that both have the same tag and are sequence-congruent are corresponding vertices. Such vertices will be identified as the "same vertex" in performing the graph union and hence will not lead to multiple vertices with the same tag in \( R_{\text{new}} \). Thus,
what remains to be shown is that there cannot be two non-sequence-congruent vertices in $R_{new}$ with the same tag.

We prove this by contradiction. Suppose $A, B$, and $Base$ can be integrated by the HPR algorithm. Let $v_1$ and $v_2$ be two vertices in $R_{new}$ that have the same tag but are not sequence-congruent. Without loss of generality, assume that $v_1$ is taken from $A$ and $v_2$ from $B$.

First assume that there is no vertex in $R_{Base}$ that has the same tag as $v_1$ and $v_2$. Hence, $R_A/v_1 \neq R_{Base}/v_1$ and $R_B/v_2 \neq R_{Base}/v_2$ (note that, by definition, $R_{Base}/v_1$ and $R_{Base}/v_2$ are empty graphs).

If $v_1$ is a non-$\phi$ vertex then $v_1 \in AF_{A, Base}$. Because the HPR algorithm successfully integrates $A, B$, and $Base$, it must be that $R_A/v_1 = R_{old}/v_1$. On the other hand, if $v_1$ is a $\phi$ vertex then there must be a non-$\phi$ vertex $v'_1 \in AF_{A, Base}$ such that $v_1$ is in the slice $R_A/v'_1$. Since $v_1' \in AF_{A, Base}$, $R_A/v_1' = R_{old}/v_1'$ and therefore, $R_A/v_1 = R_{old}/v_1$. Thus, regardless of whether $v_1$ is a $\phi$ vertex or a non-$\phi$ vertex, we have $R_A/v_1 = R_{old}/v_1$.

By the same argument, $R_B/v_2 = R_{old}/v_2$.

Note that the HPR algorithm considers $v_1$ and $v_2$ to be the “same vertex” in performing graph union. Thus, $R_A/v_1 = R_{old}/v_1 = R_B/v_2$. However, since $v_1$ and $v_2$ are not sequence-congruent, $R_A/v_1 \neq R_B/v_2$. This is a contradiction. Therefore, there cannot be two non-sequence-congruent vertices $v_1$ and $v_2$ in $R_{new}$ with the same tag if there is no vertex with that tag in $R_{Base}$.

Next assume that there is a vertex in $R_{Base}$ that has the same tag as $v_1$ and $v_2$. Let $v_{Base}$ be such a vertex in $R_{Base}$. Because $v_1$ and $v_2$ are not sequence-congruent, $R_A/v_1 \neq R_B/v_2$. Hence, $R_A/v_1 \neq R_{Base}/v_{Base}$ or $R_B/v_2 \neq R_{Base}/v_{Base}$. Without loss of generality, we may assume that $R_A/v_1 \neq R_{Base}/v_{Base}$.

Because $R_A/v_1 \neq R_{Base}/v_{Base}$, by the same argument as above, $R_A/v_1 = R_{old}/v_1$. There are two cases depending on whether $R_B/v_2 = R_{Base}/v_{Base}$.

Case 1. $R_B/v_2 \neq R_{Base}/v_{Base}$. Because $R_B/v_2 \neq R_{Base}/v_{Base}$, by the same arguments as above, the slice $R_B/v_2$ must be included in $R_{old}$ and $R_B/v_2 = R_{old}/v_1$ for otherwise the HPR algorithm would report interference. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the fact that $R_A/v_1 \neq R_B/v_2$.

Case 2. $R_B/v_2 = R_{Base}/v_{Base}$. By assumption, $v_2$ is included in $R_{new}$. There are two ways in which $v_2$ can be included in $R_{new}$.

1. There is a vertex $w_B \in Affected_B$ such that $v_2$ is included in $R_B/w_B$. Since $w_B \in Affected_B$, $w_B \in New_B$ and hence $w_B \in AF_{B, Base}$ in the HPR algorithm. Because the HPR algorithm successfully integrates $A, B$, and $Base$, $R_{old}/w_B = R_B/w_B$. Therefore, $R_{old}/v_2 = R_B/v_2$. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the fact that $R_A/v_1 \neq R_B/v_2$.

2. There is a vertex $w \in Unchanged$ such that $v_2$ is in the limited slice $Preserved(w)$. Therefore, $Preserved(w)$ is $R_B/w$. Because $Preserved(w)$ is $R_B/w$, either $R_B/w \neq R_{Base}/w$ or $R_B/w = R_A/w$.

If $R_B/w \neq R_{Base}/w$, $R_B/w \neq R_{Base}/w$, hence $w \in AF_{B, Base}$. Because (1) the HPR algorithm successfully integrates $A, B$, and $Base$ and (2) $w \in AF_{B, Base}$, $R_{old}/w = R_B/w$. Because $v_2$ is a vertex in $R_B/w$, $R_{old}/v_2 = R_B/v_2$. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the fact that $R_A/v_1 \neq R_B/v_2$.

Suppose $R_B/w = R_A/w$. Because $v_2$ is in $R_B/w$, by the definition of equality of limited slices, $v_1$ must be in $R_A/w$ and must correspond to $v_2$. In particular, $v_1$ and $v_2$ must be in the same sequence-congruence class. This contradicts a previous assumption that $v_1$ and $v_2$ are not sequence-congruent.

There is a contradiction in either case. Therefore, there cannot be two vertices $v_1$ and $v_2$ in $R_{new}$ with the same tag. □
**Lemma 6.2.** Suppose $A$, $B$, and $\text{Base}$ can be integrated by the HPR algorithm. Then $R_{\text{new}}$ is a subgraph of $R_{\text{old}}$.

**Proof.** By Lemma 6.1, if $A$, $B$, and $\text{Base}$ can be integrated by the HPR algorithm, there is at most one vertex with a given tag in $R_{\text{new}}$. Thus, tags provide a means for identifying vertices of $R_{\text{new}}$.

Since $R_{\text{new}} = \text{Preserved} \cup \text{ChangedComps}_A \cup \text{ChangedComps}_B$, it suffices to show the following three propositions: (1) $\text{Preserved}$ is a subgraph of $R_{\text{old}}$, (2) $\text{ChangedComps}_A$ is a subgraph of $R_{\text{old}}$, and (3) $\text{ChangedComps}_B$ is a subgraph of $R_{\text{old}}$.

**Proposition 1.** $\text{Preserved}$ is a subgraph of $R_{\text{old}}$.

Since $\text{Preserved} = \bigcup_{u \in \text{Unchanged}} \text{Preserved}(u)$, it suffices to show $\text{Preserved}(u)$ is a subgraph of $R_{\text{old}}$ for each $u \in \text{Unchanged}$. For any vertex $u$ in $\text{Unchanged}$, $u$ is in both $R_A$ and $R_B$. There are four possibilities: (1) $u \in PR_{A,B,\text{Base}}$, (2) $u \in AF_{B,\text{Base}}$ but $u \notin AF_{A,\text{Base}}$, (3) $u \in AF_{A,\text{Base}}$ but $u \notin AF_{B,\text{Base}}$, or (4) $u \in AF_{A,\text{Base}}$ and $u \in AF_{B,\text{Base}}$. We consider each case in turn.

**Case 1.** $u \in PR_{A,B,\text{Base}}$. Because $R_A/u = R_B/u = R_{\text{Base}}/u$, $R_A/u = R_B/u = R_{\text{Base}}/u$. So $\text{Preserved}(u) = R_{\text{Base}}/u$ (or, equivalently, $R_A/u$ or $R_B/u$). Because $R_{\text{Base}}/u$ is a subgraph of $R_{\text{Base}}/u$ and $R_{\text{Base}}/u$ is a subgraph of $R_{\text{Base}}/PR_{A,B,\text{Base}}$, which, in turn, is a subgraph of $R_{\text{old}}$, $\text{Preserved}(u)$ is a subgraph of $R_{\text{old}}$.

**Case 2.** $u \notin AF_{A,\text{Base}}$ but $u \notin AF_{B,\text{Base}}$. Because $R_A/u = R_{\text{Base}}/u$, $R_A/u = R_{\text{Base}}/u$. So $\text{Preserved}(u) = R_B/u$. Because $R_B/u$ is a subgraph of $R_B/u$ and $R_B/u$ is a subgraph of $R_B/AF_{B,\text{Base}}$, which, in turn, is a subgraph of $R_{\text{old}}$, $\text{Preserved}(u)$ is a subgraph of $R_{\text{old}}$.

**Case 3.** $u \in AF_{A,\text{Base}}$ and $u \notin AF_{B,\text{Base}}$. This case is similar to Case 2.

**Case 4.** $u \in AF_{A,\text{Base}}$ and $u \in AF_{B,\text{Base}}$. Since $u \in AF_{A,\text{Base}}$, $R_A/u$ is a subgraph of $R_{A}/AF_{A,\text{Base}}$, which, in turn, is a subgraph of $R_{\text{old}}$. Since $u \in AF_{B,\text{Base}}$, $R_B/u$ is a subgraph of $R_{B}/AF_{B,\text{Base}}$, which, in turn, is a subgraph of $R_{\text{old}}$. Note that $\text{Preserved}(u)$ must be either $R_A/u$ or $R_B/u$, which are subgraphs of $R_A/u$ and $R_B/u$, respectively. Therefore, $\text{Preserved}(u)$ is a subgraph of $R_{\text{old}}$.

In any of the above four cases, $\text{Preserved}(u)$ is a subgraph of $R_{\text{old}}$ for each $u \in \text{Unchanged}$. Therefore, $\text{Preserved}$ is a subgraph of $R_{\text{old}}$.

**Proposition 2.** $\text{ChangedComps}_A$ is a subgraph of $R_{\text{old}}$.

$\text{ChangedComps}_A$ is the union of $R_{A}/w_A$ for all vertices $w_A \in \text{Affected}_A$. It suffices to show that $R_{A}/w_A$ is a subgraph of $R_{\text{old}}$ for each vertex $w_A \in \text{Affected}_A$. Let $w_A$ be a vertex in $\text{Affected}_A$. Because $w_A \in \text{Affected}_A$ and $\text{Modified}_A$ is an empty set, $w_A \in \text{New}_A$. If there is no vertex $w_{\text{Base}}$ in $R_{\text{Base}}$ that has the same tag as $w_A$, by definition, $w_A \in AF_{A,\text{Base}}$. Because $w_A \in \text{New}_A$, if there is a vertex $w_{\text{Base}}$ in $R_{\text{Base}}$ that has the same tag as $w_A$, we have $R_A/w_A \neq R_{\text{Base}}/w_{\text{Base}}$. Therefore, $w_A \in AF_{A,\text{Base}}$.

In either case, $w_A \in AF_{A,\text{Base}}$. Because $R_{A}/w_A$ is a subgraph of $R_{A}/w_A$ and $R_{A}/w_A$ is a subgraph of $R_{A}/AF_{A,\text{Base}}$ and $R_{A}/AF_{A,\text{Base}}$ is a subgraph of $R_{\text{old}}$, $R_{A}/w_A$ is a subgraph of $R_{\text{old}}$. Therefore, $\text{ChangedComps}_A$ is a subgraph of $R_{\text{old}}$.

**Proposition 3.** $\text{ChangedComps}_B$ is a subgraph of $R_{\text{old}}$.

This proposition is similar to Proposition 2.

From the above three propositions, $R_{\text{new}}$ is a subgraph of $R_{\text{old}}$. □

**Lemma 6.3.** Suppose $A$, $B$, and $\text{Base}$ can be integrated by the HPR algorithm. Then $R_{\text{new}}$ is a slice of $R_{\text{old}}$. 

PROOF. Since the HPR algorithm successfully integrates A, B, and Base, \( R_{old} \) is a feasible PRG. Note that every vertex in a feasible PRG has a fixed number of incoming edges of a given type. We prove Lemma 6.3 by considering the incoming edges of each vertex in \( R_{new} \).

By Lemma 6.2, \( R_{new} \) is a subgraph of \( R_{old} \). The proposition that \( R_{new} \) is a slice of \( R_{old} \) is equivalent to the following proposition: if \( v \) is a vertex in \( R_{new} \), and there is a control or flow dependence edge \( u \rightarrow v \) in \( R_{old} \), then the edge \( u \rightarrow v \) is in \( R_{new} \).

Case 1. Suppose one of the following holds: \( v \) is a \( \phi \) vertex, \( v \in \text{Intermediate}_A \), \( v \in \text{Intermediate}_B \), or \( v \in \text{Unchanged} \). Because \( R_{new} = (R_A \cup \text{Affected}_A) \cup (R_B \cup \text{Affected}_B) \cup \bigcup_{u \in \text{Unchanged}} \text{Preserved}(u) \), \( v \) is included in the limited slice \( R_A \cup w_A \) for some \( w_A \in (\text{Affected}_A \cup \text{Unchanged}) \) or the limited slice \( R_B \cup w_B \) for some \( w_B \in (\text{Affected}_B \cup \text{Unchanged}) \). Without loss of generality, assume that \( v \) is in the limited slice \( R_A \cup w_A \) for some \( w_A \in (\text{Affected}_A \cup \text{Unchanged}) \). Note that every vertex in a PRG has a fixed number of incoming edges of a given type. From the definition of limited slices, since \( v \) is included in a limited slice \( R_A \cup w_A \), this limited slice must have included for \( v \) the correct number of incoming edges of each type. Therefore, \( R_{new} \) must have included the correct number of incoming edges for vertex \( v \). Since \( R_{new} \) is a subgraph of \( R_{old} \) (by Lemma 6.2), every incoming edge of \( v \) in \( R_{new} \) is also in \( R_{old} \). If the edge \( u \rightarrow v \) is in \( R_{old} \) but not in \( R_{new} \), then \( v \) has an extra incoming edge in \( R_{old} \), which makes \( R_{old} \) infeasible. This contradicts the observation that \( R_{old} \) is feasible. Therefore, the edge \( u \rightarrow v \) must also be in \( R_{new} \).

Case 2. Suppose \( v \in \text{Affected}_A \). Because \( v \in \text{Affected}_A \) and \( R_{new} = (R_A \cup \text{Affected}_A) \cup (R_B \cup \text{Affected}_B) \cup \bigcup_{u \in \text{Unchanged}} \text{Preserved}(u) \), the limited slice \( R_A \cup v \) is included in \( R_{new} \). Note that every vertex in a PRG has a fixed number of incoming edges of a given type. From the definition of limited slices, the limited slice \( R_A \cup v \) must have included for \( v \) the correct number of incoming edges of each type. Therefore, \( R_{new} \) must have included the correct number of incoming edges for vertex \( v \). Since \( R_{new} \) is a subgraph of \( R_{old} \) (by Lemma 6.2), every incoming edge of \( v \) in \( R_{new} \) is also in \( R_{old} \). If the edge \( u \rightarrow v \) is in \( R_{old} \) but not in \( R_{new} \), then \( v \) has an extra incoming edge in \( R_{old} \), which makes \( R_{old} \) infeasible. This again contradicts the observation that \( R_{old} \) is feasible. Therefore, the edge \( u \rightarrow v \) must also be in \( R_{new} \).

Case 3. Suppose \( v \in \text{Affected}_B \). This case is similar to Case 2.

From the above three cases, we conclude that if \( v \) is a vertex in \( R_{new} \) and there is a control or flow dependence edge \( u \rightarrow v \) in \( R_{old} \), then the edge \( u \rightarrow v \) is in \( R_{new} \). If \( R_{new} \) were not a slice of \( R_{old} \), then there would be some vertex \( v \) in \( R_{new} \) such that at least one incoming edge of \( v \) in \( R_{old} \) was not in \( R_{new} \). However, we just argued that this cannot happen; therefore, \( R_{new} \) is a slice of \( R_{old} \).

THEOREM. (COMPARISON THEOREM). When the HPR algorithm successfully integrates \( A, B, \) and \( Base \), the new algorithm also succeeds and the integrated program produced by the new algorithm is a slice of the integrated program produced by the HPR algorithm.

PROOF. Because the HPR algorithm successfully integrates \( A, B, \) and \( Base, R_{old} \) is a feasible PRG. From Lemma 6.3, we know that \( R_{new} \) is a slice of \( R_{old} \). By the Feasibility Lemma for PRGs, to show that \( R_{new} \) is feasible as well, all we must demonstrate is that \( R_{new} \) is a slice of \( R_{old} \) with respect to a set of non-\( \phi \) vertices. By definition, \( R_{new} = (R_A \cup \text{Affected}_A) \cup (R_B \cup \text{Affected}_B) \cup \bigcup_{u \in \text{Unchanged}} \text{Preserved}(u) \). But \( \text{Affected}_A, \text{Affected}_B, \) and \( \text{Unchanged} \) are sets of non-\( \phi \) vertices, and \( R_{new} = R_{new}/(\text{Affected}_A \cup \text{Affected}_B \cup \text{Unchanged}) \). Therefore, \( R_{new} = R_{old}/(\text{Affected}_A \cup \text{Affected}_B \cup \text{Unchanged}) \). We conclude that the new integration algorithm also produces a feasible merged Program Representation Graph.
Extra components included in the integrated program by the HPR algorithm are the result of that algorithm’s less precise computation of affected components; the fact that the Integration Theorem holds for the new algorithm assures us that the programs produced by the new algorithm are reasonable ones.

It is interesting to consider the kinds of changes that cause the HPR algorithm to report interference, while the new algorithm succeeds in producing an integrated program. Three such classes of changes were illustrated in Figure 1. In those examples, the HPR algorithm reports interference because it incorrectly identifies unchanged program components as having changed execution behaviors. There is another class of integration problems on which the HPR algorithm reports interference while the new algorithm succeeds. These are problems in which both variants change a component’s execution behavior (in different ways). In this case, the HPR algorithm reports interference because its definition of corresponding vertices relies only on tags; there can be only one copy of the changed component in the integrated program, and it cannot simultaneously have both changed behaviors. In contrast, the new algorithm considers a component of a variant to be a new component whenever its execution behavior has been changed. Thus, even if there is a component in the other variant that has the same tag this does not cause any interference since the two components are considered distinct new components by the new integration algorithm. The programs shown below illustrate this situation.

<table>
<thead>
<tr>
<th>Base</th>
<th>Variant A</th>
<th>Variant B</th>
<th>Integrated Program Produced by the New Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;T0&gt; program</td>
<td>&lt;T0&gt; program</td>
<td>&lt;T0&gt; program</td>
<td>&lt;T0&gt; program</td>
</tr>
<tr>
<td>&lt;T1&gt; x := 1</td>
<td>&lt;T1&gt; x := 1</td>
<td>&lt;T1&gt; x := 1</td>
<td>&lt;T1&gt; x := 1</td>
</tr>
<tr>
<td>&lt;T2&gt; x := 2</td>
<td>&lt;T4&gt; y := x + 4</td>
<td>&lt;T2&gt; x := 2</td>
<td>&lt;T4&gt; y := x + 4</td>
</tr>
<tr>
<td>&lt;T3&gt; x := 3</td>
<td>&lt;T2&gt; x := 2</td>
<td>&lt;T4&gt; y := x + 4</td>
<td>&lt;T2&gt; x := 2</td>
</tr>
<tr>
<td>&lt;T4&gt; y := x + 4</td>
<td>&lt;T3&gt; x := 3</td>
<td>&lt;T3&gt; x := 3</td>
<td>&lt;T4&gt; y := x + 4</td>
</tr>
<tr>
<td>&lt;T5&gt; end(x)</td>
<td>&lt;T5&gt; end(x)</td>
<td>&lt;T5&gt; end(x)</td>
<td>&lt;T5&gt; end(x)</td>
</tr>
</tbody>
</table>

Component tags are shown explicitly on the left. The statements tagged T4 in A, B, and Base have different execution behaviors. Since the statements tagged T4 in A, B, and Base are considered to be the same components in the HPR algorithm, there is interference due to conflicting execution behaviors; however, in the new integration algorithm, the two statements tagged T4 in A and B are considered to be distinct new components; they both are included in the integrated program, as shown on the right.

Note that the integrated program produced by the new algorithm includes two components with the same tag. This can cause problems if the integrated program is itself used as an argument in future program-integration problems. The ideal solution to this problem would be to find a mechanism for generating tags (for example, based on the final partition produced by the Sequence-Congruence Algorithm), rather than relying on editor-supplied tags. In this case, the tags generated for one instance of program integration would not be reused by future integrations, so that the integrated program shown above would no longer be problematic. How best to generate tags for use by the program-integration algorithm is currently an open problem.

A final point of comparison with the HPR algorithm is that the algebraic properties of the HPR algorithm have been characterized using Brouwerian algebra [Reps89a]. Unfortunately, the new integration algo-
rithm does not seem to fit this model; thus, the algebraic characterization of the new algorithm is a second open problem.

7. RELATION TO PREVIOUS WORK

This paper presents a new program integration algorithm based on the Sequence-Congruence Algorithm of [Yang89]. There are several advantages of the new algorithm over the HPR integration algorithm [Horwitz88, Horwitz89]. One is concerned with the ability to change the text of a program component. In the HPR integration algorithm, the requirement that two vertices with the same tag must have the same text means that when a programmer changes the text of a program component, the corresponding vertex in the Program Dependence Graph is assigned a new tag. In contrast, the new integration algorithm allows the corresponding vertex in the PRG to retain its old tag.

Because vertices with the same tag can have different text, certain new kinds of interference conditions can occur. To sidestep this problem, we have imposed the requirement that the left-hand sides of corresponding vertices — namely, the variables assigned to in the vertices — must be identical. For instance, consider the following integration example:

<table>
<thead>
<tr>
<th>Base</th>
<th>Variant A</th>
<th>Variant B</th>
<th>$M_1$</th>
<th>$M_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>$&lt;T1&gt; \ x := 1$</td>
<td>$&lt;T1&gt; \ u := 1$</td>
<td>$&lt;T1&gt; \ ??? := 1$</td>
<td>$&lt;T1&gt; \ x := 1$</td>
</tr>
<tr>
<td>end</td>
<td>$&lt;T2&gt; \ y := x + 1$</td>
<td>$&lt;T3&gt; \ z := u + 2$</td>
<td>$&lt;T2&gt; \ y := x + 1$</td>
<td>$&lt;T1&gt; \ u := 1$</td>
</tr>
</tbody>
</table>

If corresponding vertices could have different left-hand sides, the merged program would be as in $M_1$. Note that in $M_1$ there is a conflict in the name of the left-hand-side variable that should be filled in in the statement tagged $T1$. In contrast, since the new integration algorithm requires that corresponding vertices have identical left-hand sides, the merged program produced by the new integration algorithm is as in $M_2$. Because the statements tagged $T1$ in variants $A$ and $B$ are not corresponding vertices (even though they have the same tag and are sequence-congruent), they both are included in the merged program; there is no conflict.

The new integration algorithm also eliminates one of the two integrability tests that were part of the HPR integration algorithm. In the HPR algorithm, we need to test explicitly whether the merged graph preserves the changed computation threads of the variants. By contrast, the new integration algorithm may discover interference in the process of building the merged graph; however, if no such interference is detected, the merged graph is guaranteed to preserve the changed computation threads of the variants.

The basic technique used to identify components with equivalent execution behaviors is the Sequence-Congruence Algorithm of [Yang89]. The Sequence-Congruence Algorithm is based on an idea for finding equivalence classes of program components introduced by Alpern, Wegman, and Zadeck [Alpern88]. Their algorithm first optimistically groups possibly equivalent components in an initial partition and then finds the coarsest partition of the components that is consistent with the initial partition (and the underlying
graph used to represent the program). The Alpern-Wegman-Zadeck algorithm considers only flow dependences in refining the initial partition, and the property that holds for the partition classes produced by that algorithm is that components of a single program that are in the same final partition produce the same values at certain moments during program execution.

In contrast, our Sequence-Congruence algorithm considers control dependences as well as data dependences, and has the following properties: (1) it is able to identify components with equivalent execution behaviors, and (2) it is able to do so even if the components are in different programs.

The problem of identifying different programs that produce an identical sequence of values was also studied by Weiser [Weiser84]. Weiser defined the notion of a slice of program \( P \) with respect to a program point \( i \) and a set of variables \( V \) as a projection of \( P \) that produces the same sequence of values for variables in \( V \) at point \( i \). Although the techniques for computing slices given in [Weiser84, Ottenstein84] have come to be regarded as the definition of slicing, Weiser's definition is actually more general; by his definition, any projection of \( P \) that produces the same sequence of values is a slice. The Sequence-Congruence Algorithm solves a related, but slightly more general problem, that of identifying program projections—in different programs—that produce the same sequence of values.

REFERENCES

Aho74.

Alpern88.

Ball89.

Cytrom89.

Ferrante87.

Hopcroft71.

Horwitz88.

Horwitz88a.
Horwitz, S., J. Prins, and T. Reps, "On the suitability of dependence graphs for representing programs," (Submitted for publication), Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (August 1988).

Horwitz89.

Kuck81.