A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations

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

Given a program *Base* and two variants, *A* and *B*, each created by modifying separate copies of *Base*, the goal of program integration is to determine whether the modifications interfere, and if they do not, to create an integrated program that includes both sets of changes as well as the portions of *Base* preserved in both variants. Text-based integration techniques, such as the one used by the UNIX *diff3* utility, are obviously unsatisfactory because one has no guarantees about how the execution behavior of the integrated program relates to the behaviors of *Base*, *A*, and *B*. The first program-integration algorithm to provide such guarantees was developed by Horwitz, Prins, and Reps. However, a limitation of that algorithm is that it incorporates no notion of semantics-preserving transformations. This limitation causes the algorithm to be overly conservative in its definition of interference. For example, if one variant changes the *way* a computation is performed (without changing the values computed) while the other variant adds code that uses the result of the computation, the algorithm would classify those changes as interfering. This paper describes a new integration algorithm that is able to accommodate semantics-preserving transformations.

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 dependence, data-flow analysis, flow dependence, program dependence graph, program integration, program representation graph, static-single-assignment form

1. INTRODUCTION

Given a program *Base* and two variant programs, each created by modifying a copy of *Base*, the goal of program integration is to determine whether the changes interfere, and if they do not, to create a merged

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602, CCR-8958530, and CCR-9100424, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, Xerox, Eastman Kodak, 3M, HP, and the Cray Research Foundation.

Authors' current addresses: W. Yang, IBM Corporation, Research Triangle Park, NC 27709 S. Horwitz and T. Reps, Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

A preliminary version of this paper appeared in the Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments [26].

program that includes the changes introduced in the variants as well as the portions of *Base* that are preserved in both variants.

The first step of an integration algorithm is to identify the changes that are made to the *Base* program to create the variant programs. There are two approaches to identifying the changes: either we can find the textual changes or we can find the behavioral changes. Although *text*-based tools for three-way merging—such as the UNIX utility *diff3*—have existed for years, when used for merging *programs* such tools are *unsafe* in the sense that they do not protect against unwanted behavioral interactions between the parts of the merged program that are incorporated from different variants. Thus, one has no guarantees about how the execution behavior of the merged program relates to the behaviors of the base program and its variants. To provide such guarantees, one must use information about the language's semantics in the identification of behavioral changes, in the test for interference, and in the method for combining non-interfering variants.

An important class of program modifications is that of semantics-preserving transformations—transformations that change the way computations or stages of computations are performed without changing the values computed. For instance, existing programs are frequently modified in order to improve the performance of the programs. It is thus desirable to have an integration algorithm that can accommodate semantics-preserving transformations. This paper proposes an integration algorithm that is capable of accommodating semantics-preserving transformations.

While our long-term goal is to design a semantics-based program-integration tool for a full-fledged programming language, for now we are using a simplified model of the program-integration problem so as to make it amenable to theoretical study. This model possesses the essential features of the problem, and thus permits us to conduct our studies without being overwhelmed by inessential details. Our integration model has the following characteristics:

(1) We restrict our attention to the integration of programs written in a simple programming language that has only scalar variables, assignment statements, conditional statements, while loops, and output statements. 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.

We assume a standard operational semantics for sequential execution of the corresponding flowchart (control flow graph [2]): at any moment there is a single locus of control together with a global execution state mapping program variables to values; the execution of each assignment statement, output statement, or predicate passes control to a single successor; the execution of each assignment statement changes the global execution state.

(2) When an integration algorithm is applied to base program *Base* and variant programs *A* and *B*, and if integration succeeds—producing program *M*—then for any initial state σ on which *Base*, *A*, and *B* all terminate normally,[1] the following properties concerning the executions of *Base*, *A*, *B*, and *M* on σ must hold:

(i) *M* terminates normally.

(ii) For any program component *c* in variant *A* that produces different sequences of values in *A* and *Base*, component *c* is in *M* and produces the same sequence of values as in *A* (*i.e.*, *M*

[1]There are two ways in which a program may fail to terminate normally on some initial state: (1) the program contains a non-terminating loop, or (2) a fault occurs, such as division by zero.

agrees with *A* at component *c*).

(iii) For any program component *c* in variant *B* that produces different sequences of values in *B* and *Base*, component *c* is in *M* and produces the same sequence of values as in *B* (*i.e.*, *M* agrees with *B* at component *c*).

(iv) For any program component *c* that produces the same sequence of values in *Base*, *A*, and *B*, component *c* is in *M* and produces that same sequence of values (*i.e.*, *M* agrees with *Base*, *A*, and *B* at component *c*).

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

By a "program component" we mean an assignment statement, a predicate, or an output statement. By "the sequence of values produced by a program component," we mean the following: for an assignment statement, the sequence of values assigned to the target variable; for a predicate, the sequence of boolean values to which the predicate evaluates; and for an output statement, the sequence of values that is printed out at that statement.

Properties (1) and (3) are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. A more informal statement of Property (2) is: changes in the behavior of *A* and *B* with respect to *Base* must be incorporated in the integrated program, along with the unchanged behavior of all three. *Any* program *M* that satisfies Properties (1), (2), and (3) integrates *Base*, *A*, and *B*; if no such program exists then *A* and *B interfere* with respect to *Base*. However, Property (2) is not decidable, even under the restrictions given by Properties (1) and (3); consequently, any program-integration algorithm will sometimes report interference—and consequently 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 criteria given above).

The first algorithm that meets the above requirements (*i.e.*, the first algorithm for semantics-based program integration) was given by Horwitz, Prins, and Reps in [12]; that algorithm is referred to hereafter as the HPR algorithm. The HPR algorithm represents a fundamental advance over text-based program-integration algorithms and provides the first step in the creation of a theoretical foundation for building a program-integration tool. However, there is room for improvement. In particular, the HPR algorithm will report interference (and hence will fail to produce an integrated program) when one variant changes the *way* a computation is performed—without changing the values computed—while the other variant adds code that uses the result of the computation.

This situation is illustrated in Figure 1, which shows three example integration problems. The HPR algorithm will report interference in all three cases; however, there is *no* interference according to the integration criteria given above, and an integrated program that satisfies the criteria is shown in each case. In the first example, variant *A* changes the computation of *area* by renaming variable *P* to *PI*, and moving the assignment *rad* := 2 inside the conditional, while variant *B* adds an assignment to variable *vol* that uses the value of *area*. In the second example, variant *A* changes the algorithm for zeroing every other element of array *a*, while variant *B* adds a computation that uses the final value of *a*. In the third example, variants *A* and *B* both perform semantics-preserving transformations: variant *A* changes the way *j* is computed by performing a reduction in strength (replacing a multiplication with an addition), while variant *B* changes the assignment to *k* (which uses the value of *j*) by moving a loop-invariant computation outside the loop.

The limitations of the HPR algorithm illustrated in Figure 1 are due to the way the HPR algorithm identifies which variables are assigned the same values in the base program as in the variants, and to the way program fragments are extracted from *Base*, *A*, and *B*, and combined to form the merged program

Base	Variant *A*	Variant *B*	Integrated Program
program	**program**	**program**	**program**
P := 3.14	$\boxed{PI := 3.14}$	*P* := 3.14	*PI* := 3.14
rad := 2	**if** *DEBUG*	*rad* := 2	**if** *DEBUG*
if *DEBUG*	**then** *rad* := 4	**if** *DEBUG*	**then** *rad* := 4
then *rad* := 4	**else** $\boxed{rad := 2}$	**then** *rad* := 4	**else** *rad* := 2
fi	**fi**	**fi**	**fi**
area := *P* * (*rad***2)	$\boxed{area := PI * (rad{*}{*}2)}$	*area* := *P* * (*rad***2)	*area* := *PI* * (*rad***2)
output(*area*)	**output**(*area*)	$\boxed{height := 10}$	*height* := 10
		$\boxed{vol := height{*}area}$	*vol* := *height***area*
		output(*area*)	**output**(*area*)
		$\boxed{\textbf{output}(vol)}$	**output**(*vol*)
program	**program**	**program**	**program**
i := 1	$\boxed{i := 2}$	*i* := 1	*i* := 2
while *i*≤*N* **do**	**while** *i*≤*N* **do**	**while** *i*≤*N* **do**	**while** *i*≤*N* **do**
if even(*i*) **then** *a*[*i*] := 0 **fi**	$\boxed{a[i] := 0}$	**if** even(*i*) **then** *a*[*i*] := 0 **fi**	*a*[*i*] := 0
i := *i* + 1	$\boxed{i := i + 2}$	*i* := *i* + 1	*i* := *i* + 2
od	**od**	**od**	**od**
output(*a*)	**output**(*a*)	$\boxed{i := 1}$	*i* := 1
		$\boxed{sum := 0}$	*sum* := 0
		$\boxed{\textbf{while } i \le N \textbf{ do}}$	**while** *i*≤*N* **do**
		$\boxed{sum := sum + a[i]}$	*sum* := *sum* + *a*[*i*]
		$\boxed{i := i + 1}$	*i* := *i* + 1
		$\boxed{\textbf{od}}$	**od**
		output(*a*)	**output**(*a*)
		$\boxed{\textbf{output}(sum)}$	**output**(*sum*)
program	**program**	**program**	**program**
k := 0; *i* := 1	*k* := 0; *i* := 1	*k* := 0; *i* := 1	*k* := 0; *i* := 1
while *i*≤100 **do**	$\boxed{twoi := 2}$	**while** *i*≤100 **do**	*twoi* := 2
j := *i* * 2	**while** *i*≤100 **do**	*j* := *i* * 2	**while** *i*≤100 **do**
while *j*<1000 **do**	$\boxed{j := twoi}$	$\boxed{teni := i * 10}$	*j* := *twoi*
k := *k* + *i* * 10 + *j*	**while** *j*<1000 **do**	**while** *j*<1000 **do**	*teni* := *i* * 10
j := *j* + 1	*k* := *k* + *i* * 10 + *j*	$\boxed{k := k + teni + j}$	**while** *j*<1000 **do**
od	*j* := *j* + 1	*j* := *j* + 1	*k* := *k* + *teni* + *j*
i := *i* + 1	**od**	**od**	*j* := *j* + 1
od	$\boxed{twoi := twoi + 2}$	*i* := *i* + 1	**od**
output(*k*)	*i* := *i* + 1	**od**	*twoi* := *twoi* + 2
	od	**output**(*k*)	*i* := *i* + 1
	output(*k*)		**od**
			output(*k*)

Figure 1. Three example integration problems that illustrate the limitations of the HPR algorithm with regard to semantics-preserving transformations. Modifications in variants *A* and *B* are enclosed in boxes. In all three cases, the HPR algorithm would report interference even though there is no interference according to the integration criteria.

(these concepts are explained in more detail in Section 2). For example, consider the first integration problem of Figure 1. The change made to *Base* to create variant *B* was the addition of the computation of *vol*. This new code must be included in the merged program; however, the HPR algorithm would extract from variant *B* the *entire* program fragment needed to compute the value of *vol*. This fragment includes the statement “*area* := *P* * (*rad* ** 2)”, which is undesirable since the way *area* is computed (though not its value) has been changed in variant *A* (by renaming *P* to *PI* and by moving one assignment to *rad* inside the conditional). It would be preferable to extract from variant *B* only the assignments to *height* and *vol*, combining this fragment with the changed fragment from *A*. However, to do this requires being able to recognize that the value of *area* is the same in variant *A* as in *Base*, which the HPR algorithm is unable to do.

To address these limitations of the HPR algorithm, we have designed a new algorithm that accommodates semantics-preserving transformations. The algorithm uses a new operation, called *limited slicing*, to extract program fragments from *Base*, *A*, and *B* that are smaller than the fragments extracted by the HPR algorithm. To identify variables that are assigned the same values in the base program and a variant, the new algorithm uses an auxiliary algorithm that determines, for each component of *Base*, *A*, and *B*, which other components are *congruent*. A precise definition of congruence is given in Section 4; roughly, two components are congruent only if they compute the same sequences of values when their respective programs are executed on the same initial state. Because the new integration algorithm can be used with *any* safe congruence-testing algorithm, this paper actually describes a *class* of integration algorithms that accommodate semantics-preserving transformations. One congruence-testing algorithm is given in Section 6; the correctness of this algorithm is proved fully in [24].

The remainder of the paper is organized into six sections, as follows. Section 2 summarizes and contrasts the techniques used by the HPR algorithm and the integration algorithm from this paper. Section 3 defines the graph representation of programs used by the new integration algorithm. Section 4 concerns congruence of program components and presents the integration algorithm itself. Section 5 discusses two important properties of the new integration algorithm. Section 6 gives a congruence-testing algorithm, which can be used in the new integration algorithm. Section 7 discusses related work. The Appendix contains a proof showing that the integration algorithm presented in this paper satisfies the integration criteria given above.

2. IDENTIFYING CHANGED COMPONENTS AND EXTRACTING PROGRAM FRAGMENTS

The fundamental problems in program integration are (1) to determine, for all possible initial states, which components of a variant will produce different values than the corresponding components of the base program (we call such components *changed components*), and (2) to extract fragments from the base and the variant programs to form a merged program. This section summarizes the techniques used by the HPR algorithm and contrasts them against the ones used by the new integration algorithm.

A component c of a variant is a *changed component* if its execution behavior is not equivalent to the execution behavior of the corresponding component of *Base*. The execution behavior of a component is the sequence of values produced at the component. In order to account for non-terminating programs, we define equivalence of execution behaviors as follows: Two program components c_1 and c_2 of programs P_1 and P_2 have equivalent execution behaviors if and only if all of the following hold:

(1) For all initial states σ such that both P_1 and P_2 terminate normally when executed on σ, the sequence of values produced at c_1 when P_1 is executed on σ is identical to the sequence of values produced at c_2 when P_2 is executed on σ.

(2) For all initial states σ such that neither P_1 nor P_2 terminates normally on σ, either (1) the sequences of values produced at c_1 and c_2 are identical infinite sequences, or (2) the sequence of

values produced at c_1 is a prefix of the sequence of values produced at c_2, or *vice versa*.

(3) For all initial states σ such that P_1 terminates normally on σ but P_2 fails to terminate normally on σ, the sequence of values produced at c_2 is a prefix of the sequence of values produced at c_1.

(4) For all initial states σ such that P_2 terminates normally on σ but P_1 fails to terminate normally on σ, the sequence of values produced at c_1 is a prefix of the sequence of values produced at c_2.

In the HPR algorithm, changed components are identified by comparing *program slices* [2318]. The slice of a program with respect to a component *c* is a projection of the program that includes all program components that might affect (either directly or transitively) the sequence of values produced at *c*. For example, there are six different slices of program *Base* from the first integration problem of Figure 1. These slices are shown below; in each case the component with respect to which the slice is taken is enclosed in a box.

```
program            program            program
[P := 3.14]        [rad := 2]         if [DEBUG]
                                      fi

program            program            program
if DEBUG           P := 3.14          P := 3.14
  then [rad := 4]  rad := 2           rad := 2
fi                 if DEBUG           if DEBUG
                     then rad := 4      then rad := 4
                   fi                 fi
                   [area := P * (rad**2)]  area := P * (rad**2)
                                      [output(area)]
```

The HPR algorithm uses the following criterion for finding changed components: a variant's changed components are all those whose slices differ from the corresponding slices of *Base*. The intuition behind this criterion is the following: If a component *c*'s slice in a variant differs from its slice in the base program, then the way *c*'s values are computed differs in the variant and the base program, and thus the values themselves might differ.

While this criterion for finding changed components is safe, it is overly pessimistic since it is possible that components with different slices still compute the same values. For example, consider again the first integration problem of Figure 1. The slices of *Base*, *A*, and *B* with respect to their assignments to variable *area* are shown below. (In this example, the slice of *B* is identical to that of *Base*.)

```
Slice of Base and B        Slice of A
program                    program
P := 3.14                  PI := 3.14
rad := 2                   if DEBUG
if DEBUG                     then rad := 4
  then rad := 4              else rad := 2
fi                         fi
[area := P * (rad**2)]     [area := PI * (rad**2)]
```

The slice of variant *A* differs from the corresponding slice of *Base*; thus, the assignment to *area* in variant *A* would be classified as a changed component of *A* by the HPR algorithm although in fact the value assigned to *area* is the same in *A* as in *Base* (and *B*). (It is this (mis)classification that causes the HPR algorithm to report interference for this example.)

Although the problem of identifying changed components *exactly* is, in general, undecidable, the slice-comparison criterion that is used in the HPR algorithm is not the only way to identify changed components *safely*; for example, a much different algorithm is given in Section 6. In this paper, we assume that such an algorithm is at our disposal; in particular, we assume that we have an algorithm for testing the congruence condition spelled out in Section 4.1. The integration algorithm that we give is parameterized by this auxiliary congruence algorithm; more precise congruence algorithms will allow the integration algorithm to accommodate more semantics-preserving transformations.

In addition to using program slicing to find changed components, the HPR algorithm also makes use of program slicing to extract fragments from *Base*, *A*, and *B*, which are then combined to form the merged program. In particular, the merged program is formed by taking the union of three slices: (1) variant *A* sliced with respect to the changed components of *A*, (2) variant *B* sliced with respect to the changed components of *B*, and (3) *Base* sliced with respect to the unchanged components of *Base*, *A*, and *B* (in the HPR algorithm, a component is classified as unchanged if its slice is the same in *Base*, *A*, and *B*). The problem with this approach to creating the merged program is that a slice corresponds to a *complete* fragment, and it is not desirable to extract a complete fragment when part of that fragment has been changed by the other variant.

For example, consider variant *B* in the second integration problem of Figure 1. Variant *B* differs from *Base* because a new computation to sum the elements of array *a* has been added. New components are always classified as changed components; thus, the HPR algorithm would include in the integrated program all slices of variant *B* with respect to the new components of *B*. Those slices would include the first loop of *B* (which zeros every other element of array *a*). This is undesirable since that loop has been changed by variant *A*. Since the value of array *a* after the first loop terminates (which is used by the new computation of *B*) is the same in *A* as in *B*, it would be preferable to include only the *new* computation from *B* rather than including the entire slice with respect to the new components.

By contrast, the algorithm given in this paper uses a different operation—called *limited slicing*—to extract *partial* fragments, rather than full slices, from *Base*, *A*, and *B*, which are then used to form the merged program. Limited slicing is defined and discussed in Section 4.3.

3. PROGRAM REPRESENTATION GRAPHS

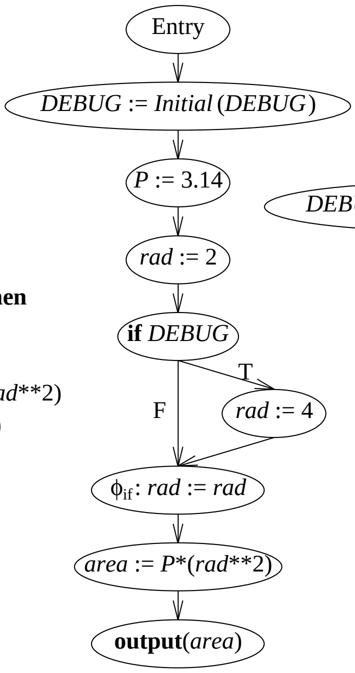
The new program-integration algorithm uses a graph representation of programs called a *Program Representation Graph*. Program Representation Graphs (PRGs) combine features of program dependence graphs [149] and static-single-assignment forms [4^{22}6]. A program's PRG is defined in terms of an augmented version of the program's control-flow graph. The standard control-flow graph includes a special *Entry* vertex, a vertex for each *if* or *while* predicate, and a vertex for each assignment or output statement in the program. The control-flow graph is augmented as follows. A vertex labeled "$x := Initial(x)$" is added at the beginning of the control-flow graph for each variable x that may be used before being defined. As in static-single-assignment forms, the control-flow graph is further augmented by adding special "ϕ vertices" so that each use of a variable in an assignment statement, a predicate, or an output statement is reached by exactly one definition.

A vertex labeled "ϕ_{if}: $x := x$" is added at the end of each *if* statement for each variable x that is defined within either (or both) branches of the *if* and is live at the end of the *if*; a vertex labeled "ϕ_{enter}: $x := x$" is added inside each *while* loop immediately before the loop predicate for each variable x that is defined within the *while* loop, and is live immediately before the loop predicate (*i.e.*, may be used either inside the loop, after the loop, or by the loop predicate before being redefined); a vertex labeled "ϕ_{exit}: $x := x$" is added immediately after the loop for each variable x that is defined within the loop and is live after the loop. Figures 2(a) and 2(b) show a program and its augmented control-flow graph.

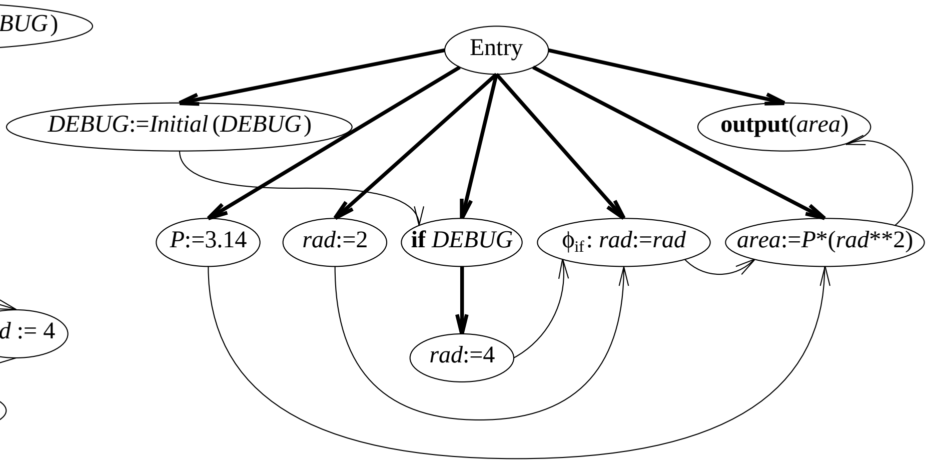
The vertices of a program's Program Representation Graph (PRG) are the same as the vertices in the augmented control-flow graph (an *Entry* vertex, a vertex for each predicate, assignment, and output statement, and for each *Initial*, ϕ_{if}, ϕ_{enter}, and ϕ_{exit} vertex). The edges of the PRG represent *control* and *flow* dependences. The source of a control dependence edge is always either the *Entry* vertex or a predicate vertex; control dependence edges are labeled either **true** or **false**. The intuitive meaning of a control dependence edge from vertex v to vertex w is the following: if the program component represented by vertex v is evaluated during program execution and its value matches the label on the edge, then (assuming all constructs terminate normally) the component represented by w will eventually execute; however, if the value does not match the label on the edge, then the component represented by w may never execute. (By definition, the *Entry* vertex always evaluates to **true**.)

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

(a)



(b)



(c)

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.

Algorithms for computing control dependences in languages with unrestricted control flow are given in [96]. For the restricted language under consideration here, control dependences can be computed in a more straightforward fashion. Most control dependence edges reflect the nesting structure of the program: there is an edge labeled **true** from the vertex that represents a *while* predicate to all vertices that represent statements nested immediately within the loop; there is an edge labeled **true** from the vertex that represents an *if* predicate to all vertices that represent statements nested immediately within the true branch of the *if*, and an edge labeled **false** to all vertices that represent statements nested immediately within the false branch; there is an edge labeled **true** from the *Entry* vertex to all vertices that represent statements that are not inside any *while* loop or *if* statement. In addition, there is a control dependence edge labeled **true** from every vertex that represents a *while* predicate to itself, and every ϕ_{enter} vertex has *two* incoming control dependence edges: one from its enclosing loop's predicate vertex *p*, and the other from *p*'s control predecessor.

Flow dependence edges represent possible flow of values, *i.e.*, there is a flow dependence edge from vertex *v* to vertex *w* if vertex *v* represents a program component that assigns a value to some variable *x*, vertex *w* represents a 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.

Figure 2(c) shows the Program Representation Graph of the program of Figure 2(a). Control dependence edges are shown using bold arrows and are unlabeled (in this example, all control dependence edges would be labeled **true**); data dependence edges are shown using arcs.

4. THE NEW INTEGRATION ALGORITHM

Given a base program *Base* and variant programs *A* and *B*, the new integration algorithm performs the following steps:

(1) Apply a congruence-testing algorithm to the Program Representation Graphs of the three programs.

(2) Use the results of 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) For all ϕ vertices *v* from which there is no path to a non-ϕ vertex in the merged graph, remove *v* and all of its incoming and outgoing edges.

(6) 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 (6).

4.1. Vertex Congruence

The new integration algorithm uses an auxiliary algorithm that identifies *congruent vertices* of the Program Representation Graphs of *Base*, *A*, and *B*. By definition, vertices *u* and *v* are congruent if and only if all of the following hold:

(1) Vertices *u* and *v* have equivalent execution behavior (as defined in Section 2).

(2) The **true**/**false** labels on *u*'s incoming control dependence edges match the **true**/**false** labels on *v*'s

corresponding incoming control dependence edges.[2]

(3) The sources of *u*'s incoming control dependence edges are congruent to the corresponding sources of *v*'s incoming control dependence edges.

Example. To illustrate congruence of vertices, we can look at the first example in Figure 1. The assignments $P := 3.14$ in *Base* and *B* and the assignment $PI := 3.14$ in *A* are congruent since the same values are produced at these components, their incoming control dependence edges have the same labels, and the sources of their incoming control dependence edges, the *Entry* vertices, are congruent. Similarly, the assignments to *area* in all three programs are congruent. However, the assignments $rad := 2$ in *Base* and *A* are not congruent because they have inequivalent behavior (if variable *DEBUG* is *true*, then $rad := 2$ in *A* will not be executed; however, the assignment $rad := 2$ is always executed in *Base*).

It is, in general, undecidable to identify congruent vertices exactly; the new integration algorithm can use any *safe* congruence-testing algorithm (*i.e.*, one that identifies a subset of the exact set of congruent pairs of vertices). For example, comparing program slices is a safe congruence-testing algorithm [20]. Our investigation of appropriate congruence-testing algorithms led to the development of the Sequence-Congruence Algorithm, which will be described in Section 6.

One advantage of the new integration algorithm is that it can easily exploit additional facts about program semantics. Many techniques used in compiler optimization [3,16,2], such as constant propagation, movement of invariant code, and common subexpression elimination, can be combined with the Sequence-Congruence Algorithm to detect larger classes of congruent components. An alternative approach would be to make use of knowledge of semantics-preserving program transformations that have been applied to a program or certain parts of the program. This information could be supplied by the editor front-end of a program-transformation system.

In the following discussion of the new integration algorithm, we do not make any assumption about how the information about which components are congruent was acquired. All we assume is that this information is at our disposal.

4.2. 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 integration algorithm attempts to preserve both kinds of changes in the integrated program. The vertices in each of the three programs (*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 identification of congruent vertices performed by Step (1) of the integration algorithm cannot always provide an answer, since a vertex in one program may be congruent to several vertices in another program (*i.e.*, congruence may not define a one-to-one correspondence). Therefore, it is assumed that vertices are tagged such that no two vertices in a program have the same tag (a vertex in one program may, of course, have the same tag as a vertex in another program). Tags may be provided by the editor used to create *A* and *B* from *Base*, or may be supplied by some other mechanism—the source of the tags is not relevant to the algorithm itself.

[2]Recall that ϕ_{enter} and *while* predicate vertices have two incoming control dependence edges; the *Entry* vertex has no incoming control dependence edges; all other vertices have exactly one incoming control dependence edge.

Given this assumption, the correspondence between vertices of the three programs is established as follows: Two vertices v_1 and v_2 *correspond* if and only if all of the following hold:

(1) v_1 and v_2 are congruent;

(2) v_1 and v_2 have the same tag;

(3) if v_1 and v_2 are assignment statements (including ϕ assignments), they assign to the same variable.[3]

Using this definition of corresponding vertices, each vertex of *Base*, *A*, and *B* is classified as defined below.

Every vertex in *A* is in one of five sets: New_A, $Modified_A$, $Modified_B$, $Intermediate_A$, or *Unchanged*.

(1) A vertex is in New_A if there is no corresponding vertex in *Base*. Vertices in New_A represent program components that have been added to *Base* to create *A*, or have been moved to contexts that have changed their execution behaviors. (The vertices in New_A may or may not have corresponding vertices in *B*.)

(2) A vertex is in $Modified_A$ 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 $Modified_A$ represent components of *A* whose texts have been changed but whose execution behaviors remain the same. (The vertices in $Modified_A$ may or may not have corresponding vertices in *B*.)

(3) A vertex is in $Modified_B$ 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*. (These vertices will also be classified as $Modified_B$ in *B* and *Base*.)

(4) A vertex is in $Intermediate_A$ 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 in *B*, or because the vertex's left-hand side variable was changed in *B*).

(5) A vertex is in *Unchanged* if there are corresponding vertices in both *Base* and *B*, and all three vertices have the same text. Vertices in *Unchanged* represent components that are textually and behaviorally identical in all three programs.

Vertices in *B* are similarly classified into the sets New_B, $Modified_B$, $Modified_A$, $Intermediate_B$, or *Unchanged*. Vertices in *Base* are classified into the sets $Modified_A$, $Modified_B$, $Intermediate_A$, $Intermediate_B$, *Unchanged*, and *Deleted*. A vertex in *Base* is in *Deleted* if neither *A* nor *B* contains a corresponding vertex. Vertices in *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*.

Because the text in corresponding vertices from *A* and *B* can be different, there are two cases when the classification process discovers that the changes made in *A* and *B* interfere: (1) when corresponding vertices with different text are classified New_A and New_B, respectively; (2) when corresponding vertices with different text are classified $Modified_A$ and $Modified_B$, respectively. Since a vertex in the merged PRG can have only one version of the text, it is not possible to preserve the changed text of this vertex from both *A* and *B*.

[3]This requirement will be explained in Section 4.8.

4.3. Identifying Changed and Preserved Computations

The merged graph must include all of the changed components of the variants, and must include enough of the “neighborhoods” of those components to ensure that they retain their execution behaviors. Limited slices provide the mechanism for extracting these neighborhoods.

Definition. Let R be the Program Representation Graph of *Base*, *A*, or *B*, and let S be a set of (ϕ and non-ϕ) vertices in R. The *limited slice* of R with respect to S, denoted by $R//S$, 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 vertices along this path, excluding the two endpoints, belong to either $Intermediate_A$ or $Intermediate_B$, then all vertices and edges on this path are included in $R//S$.

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.

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.)

$Affected_A = New_A \cup Modified_A$

$ChangedComps_A = R_A \,//\, Affected_A$

$Affected_B$ 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 equal;[4] 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:

Relationship of limited slices	$Preserved(u)$
$R_A//u = R_B//u$	$R_A//u$
$(R_A//u = R_{Base}//u)$ and $(R_A//u \neq R_B//u)$	$R_B//u$
$(R_B//u = R_{Base}//u)$ and $(R_B//u \neq R_A//u)$	$R_A//u$

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

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

[4]Two limited slices are *equal* if the vertex correspondence relation induces an isomorphism between them.

4.4. 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*:

$$R_M = ChangedComps_A \cup ChangedComps_B \cup Preserved$$

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 $ChangedComps_A$ and $ChangedComps_B$ will include corresponding vertices with different text. This can *only* happen, however, if the two vertices are both classified $Modified_A$ or both classified $Modified_B$. 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*. In all other cases, corresponding vertices from *A* and *B* have the same text (otherwise interference would have been detected during vertex classification).

4.5. Removing Extra φ Vertices

When the PRG of a program is created, there is a path from every φ vertex to a non-φ vertex. "Extra" φ vertices—ones without such paths—can sometimes occur in the merged graph created by the previous step of the integration algorithm. One way this situation arises is when a φ vertex is classified as *Unchanged*, but—because of deletions or rearrangements of the code—some of the vertex's successors in *Base* are no longer successors in *A* and the rest are no longer successors in *B*. If the "extra" φ vertices were left in the merged graph, then the graph would not be the PRG of any program, and the next step of the integration algorithm, which tests whether the merged graph corresponds to some program, would fail. Thus, all such "extra" φ vertices and their incident edges are removed from the merged graph.

4.6. 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 [11]; a similar result can be shown for Program Representation Graphs. The crux of the problem is finding an order for each predicate's control children. However, we expect that, for graphs created by merging the Program Representation Graphs of actual programs, problematic cases will rarely arise. We have explored ways of reducing the search space, in the belief that a backtracking method for solving the remaining step will behave satisfactorily [12].

4.7. An Example Integration

Figure 3 illustrates the new integration algorithm using the first set of example programs from Figure 1. Figure 3 shows: (1) the sets of vertices that would be classified as $Affected_A$, $Affected_B$, and *Unchanged* if the Sequence-Congruence Algorithm (to be discussed in Section 6) were used for Step (1) of the new integration algorithm; (2) the graph fragments $ChangedComps_A$, $ChangedComps_B$, and *Preserved*; and (3) the merged graph. This merged graph is feasible, and corresponds to the program shown in Figure 1 as the result of integrating the first set of programs.

4.8. Discussion of Classification of Vertices

In Section 4.2, we require that corresponding assignment statements assign to the same variables otherwise they are not corresponding components. This is because vertices with the same tag can have different

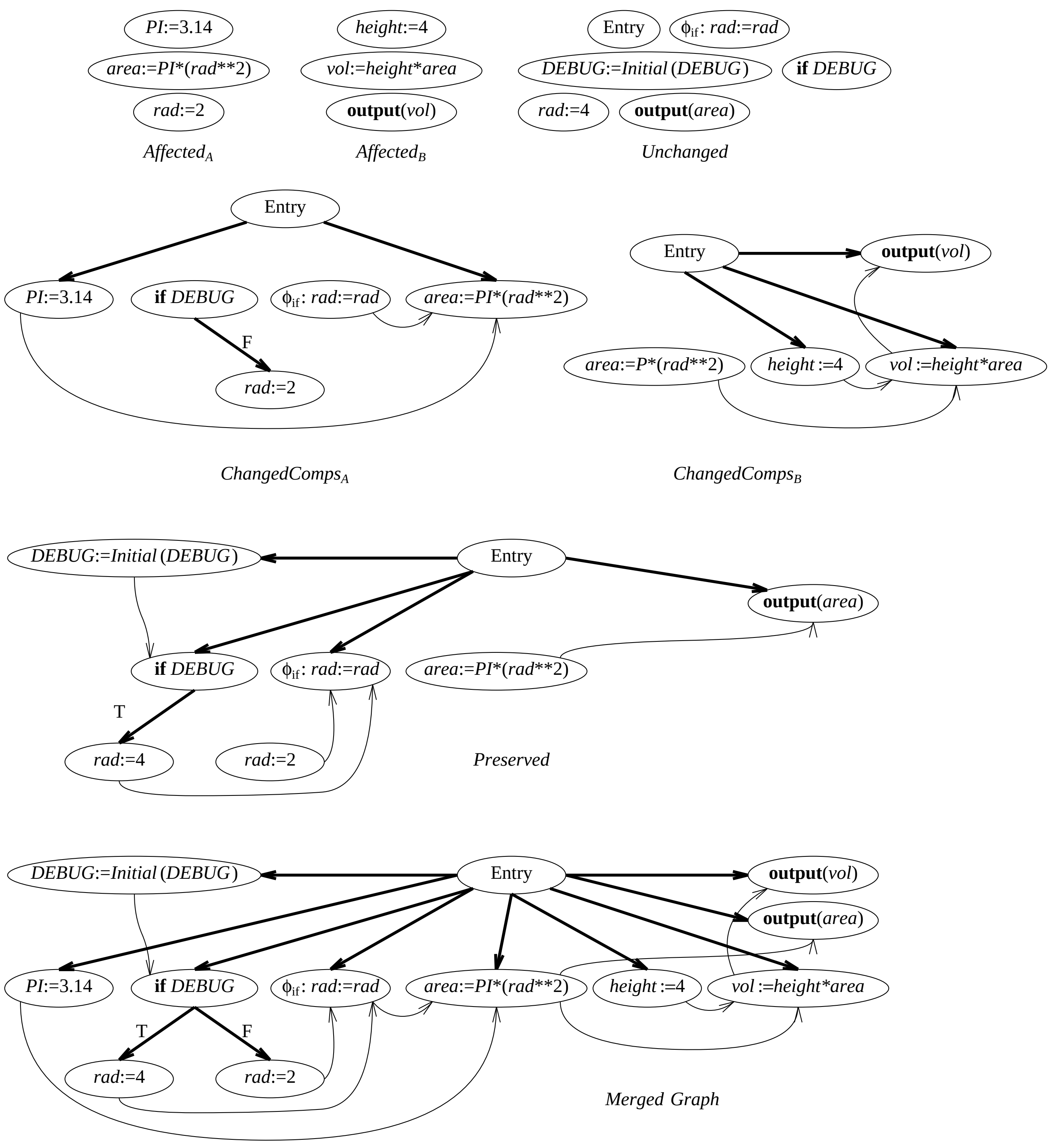


Figure 3. The new integration algorithm is illustrated using the first set of example programs from Figure 1. Note that some vertices in *ChangedComps*$_A$, *ChangedComps*$_B$, and *Preserved* have no incoming control dependence edges.

texts. Without such a restriction, certain new kinds of interference conditions can occur. To sidestep this problem, we have imposed the requirement. For instance, consider the following integration example:

```
Program Base          Variant A              Variant B              Program M1             Program M2

      program                program                program                program                program
<T1>  x := 1          <T1>   x := 1          <T1>   u := 1          <T1>   ??? := 1        <T1>   x := 1
                      <T2>   y := x + 1      <T3>   z := u + 2      <T2>   y := x + 1      <T2>   y := x + 1
                                                                    <T3>   z := u + 2      <T1>   u := 1
                                                                                           <T3>   z := u + 2
```

If corresponding assignment statements could assign to different variables, the merged program would be *M1*. Note that in *M1* there is a conflict in the name of the variable that should be used in the statement tagged T1. Since the new integration algorithm requires that corresponding assignment statements assign to the same variable, the merged program produced by the new integration algorithm is *M2*. Note that the statements tagged T1 in variants *A* and *B* are not corresponding vertices (even though they satisfy all other requirements of correspondence). Hence they both are included in the merged program; there is no interference.

5. PRESERVATION OF TEXTUAL CHANGES AND BEHAVIORAL CHANGES

There are two kinds of changes that can be made in the variants: the text of a component may be changed or the behavior of the component may be changed. In this section, we show that both kinds of changes will be preserved in the merged program in a successful integration. In particular, we show that the new integration algorithm satisfies the conditions of the integration model defined in Section 1.

We need a new term to state the properties precisely: Two program components are *analogous* if they have the same tag. Note that corresponding components must be analogous, but not *vice versa*.

5.1. Preservation of Textual Changes

The preservation of textual changes in the merged program is shown by considering the construction of the merged graph. In the new integration algorithm, textual changes are captured by the sets $Affected_A$ and $Affected_B$, which include, among other components, those components whose text has been changed. The limited slices with respect to components in $Affected_A$ and $Affected_B$ are always included in the merged graph. Thus, textual changes made in *A* and *B* are preserved in the merged program in a successful integration. The preservation of textual changes is summarized in the following Theorem.

THEOREM. *Suppose the new integration algorithm successfully integrates two variants A and B with respect to the base program Base and produces a merged program M. Then*

(1) *For any program component v_A in A, if v_A's text differs from that of the analogous component in Base, then there is a component v in M that has the same text as v_A.*

(2) *For any program component v_B in B, if v_B's text differs from that of the analogous component in Base, then there is a component v in M that has the same text as v_B.*

(3) *For any program component v_{Base} in Base, if v_{Base} has the same text as the analogous components in both A and B, then there is a component v in M that has the same text as v_{Base}.*

PROOF. Suppose v_A is a component of *A* whose text differs from that of the analogous component in *Base*. Then $v_A \in Modified_A$ or $v_A \in New_A$ depending on whether v_A is put in the same congruence class as the analogous component in *Base* in the first step of the new integration algorithm. In either case, since the limited slices with respect to components in $\mathit{Modified}_A$ and New_A are included in the merged graph, there is a component v in M that has the same text as v_A. This proves the first clause. The second clause is proved similarly.

Suppose v_{Base} is a component of *Base* whose text is the same as that of the analogous components, v_A and v_B, in A and B, respectively. Then either $v_{Base} \in \mathit{Unchanged}$ or $v_A \in \mathit{New}_A$ or $v_B \in \mathit{New}_B$ depending on whether v_{Base}, v_A, and v_B are put in the same congruence classes in the first step of the new integration algorithm. If $v_{Base} \in \mathit{Unchanged}$, there is a component v in M that has the same text as v_{Base}. If $v_A \in \mathit{New}_A$, there is a component v in M that has the same text as v_A. If $v_B \in \mathit{New}_B$, there is a component v in M that has the same text as v_B. In any case, there is a component v in M that has the same text as v_{Base}. This proves the third clause. □

5.2. Preservation of Behavioral Changes

In addition to textual changes, we can show that the new integration algorithm preserves the behavioral changes made in the variants. The preservation of behavioral changes implies that the new integration algorithm satisfies the conditions of the integration model defined in Section 1.

THEOREM. (Integration Theorem). *Suppose the new integration algorithm successfully integrates two variants A and B with respect to the base program Base and produces a merged program M. Then for any initial state σ on which A, B, and Base all terminate normally:*

(1) *M terminates normally on σ.*

(2) *For any program component v_A in A, if v_A produces a different sequence of values than the analogous component in Base, then there is a component v in M that produces the same sequence of values as v_A.*

(3) *For any program component v_B in B, if v_B produces a different sequence of values than the analogous component in Base, then there is a component v in M that produces the same sequence of values as v_B.*

(4) *For any program component v_{Base} in Base, if v_{Base} produces the same sequence of values as the analogous components in both A and B, then there is a component v in M that produces the same sequence of values as v_{Base}.*

The proof of this theorem is included in the Appendix.

6. THE SEQUENCE-CONGRUENCE ALGORITHM

The first step in the new program-integration algorithm is to determine the set of congruent pairs of program components. To determine the congruent pairs of components is essentially to determine program components that have equivalent behaviors. Any technique that can detect program components with equivalent behaviors can be used in the new program-integration algorithm. This section describes one such algorithm that we developed, which is called the Sequence-Congruence Algorithm.

The Sequence-Congruence Algorithm was inspired by an idea of Alpern, Wegman, and Zadeck for extending value numbering to work in the presence of conditional statements and loops [4]; however, our results are quite different. The Alpern-Wegman-Zadeck algorithm finds what they call *congruent* program components by first optimistically grouping possibly congruent components in an initial partition and then finding the coarsest partition consistent with the initial partition. They show that if two components are in the same final partition, and the components both dominate some point p in the program's control-flow graph, then whenever p is executed, the values most recently produced at the two components will be the same.

The Sequence-Congruence Algorithm also uses the idea of partitioning an initial optimistic grouping of possibly equivalent components. However, it is applied to a different graph and uses two partitioning passes rather than one;[5] consequently, the final partitions produced by the Sequence-Congruence Algorithm are different from the final partitions produced by the Alpern-Wegman-Zadeck algorithm. Components that are in the same final partition produced by the Sequence-Congruence Algorithm have a different property than components in the same final partition produced by the Alpern-Wegman-Zadeck algorithm; while the latter have the same values at *certain moments* during program execution, the former—as shown in [25]—have *equivalent execution behaviors* (*i.e.*, the *sequences of values* produced at the components are guaranteed to be identical).

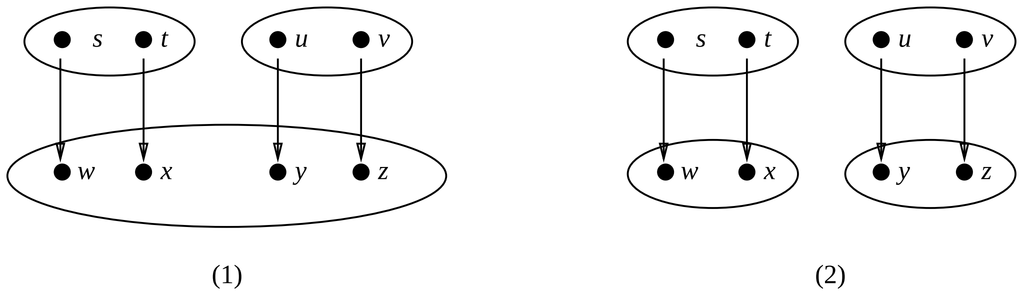
A further point of contrast between our work and that of [4] concerns the idea of applying partitioning to more than one program simultaneously. This idea makes no sense in the context of the Alpern-Wegman-Zadeck algorithm since a point in one program's control-flow graph cannot be dominated by a component of another program; however, the Sequence-Congruence Algorithm can be applied to any number of programs to find components with equivalent execution behaviors.

Section 6.1 presents the Sequence-Congruence Algorithm; Section 6.2 gives a rough outline of a proof that the Sequence-Congruence Algorithm correctly identifies program components with equivalent behaviors; Section 6.3 describes some enhancements to the Sequence-Congruence Algorithm that allow more precise determination of which components have equivalent behaviors.

6.1. The Sequence-Congruence Algorithm

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 safe to assume that components with different operators, different operands, or different controlling predicates will have different 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 and the idea of finding the coarsest stable partition of a graph. The following figure illustrates what we mean by a stable partition:



In Part (1) of the above figure, vertices *s* and *t* are in the same class; *u* and *v* are in the same class; *w*, *x*, *y*, and *z* are in the same class. Because the predecessors of *w* and *x* and the predecessors of *y* and *z* are in

[5]Combining the two partitioning passes into a single pass would not invalidate our results, but would lead to a weaker sequence-congruence algorithm; *i.e.*, program components in the same final partition would still be guaranteed to have equivalent behaviors, but fewer components would be placed in the same final partition. This point is discussed further in connection with the example given at the end of Section 6.1.

different classes, the partition is unstable; therefore, the class containing *w*, *x*, *y*, and *z* must be split into two classes, one for *w* and *x* and the other for *y* and *z* as shown in Part (2) of the figure. The graph shown in Part (2) is stable, and is the coarsest stable partition of the graph of Part (1). Given an initial partition of a graph's vertices, there exists a coarsest refinement of the initial partition that is stable, which can be computed by a variation of an algorithm due to Hopcroft [10].

The Sequence-Congruence Algorithm uses this technique to partition components of one or more *PRG*s. The algorithm consists of two partitioning passes. Vertices that have different operators are put into different initial partitions. 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. Both passes use the same basic partitioning algorithm to refine the partition of the graph's vertices; only the starting partition and the edges considered in the two passes are different.

The initial partition is based on the operator in a vertex. The operator in a statement or a predicate vertex is determined from the expression part of the vertex. For instance, statement "$x := a + b * c$" has the same operator as statement "$y := d + e * f$" but a different operator than statement "$z := g * h$"; that is, the structure of the expression in the vertex defines the operator. The expression "$a + b * c$" uses the operator that takes three arguments *a*, *b*, and *c*, and returns the value of "$a + b * c$". (Note that the assignment sign := is *not* considered to be an operator in the Sequence-Congruence Algorithm since it does not compute a value. It is the expression on the right-hand side of the assignment sign that computes a value.)

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; an output statement is *simple* if it prints out the value of a variable. Vertices that represent simple predicates, simple assignments, or simple output statements are called *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 "**output**(*i*)."

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 the constant. There is a different operator for each different constant in the program.

In *PRG*s, two vertices that are the same kind of ϕ vertex (*i.e.*, ϕ_{enter}, ϕ_{exit}, or ϕ_{if}) or that have the same operators must have the same number of incoming control and flow dependence edges. The *corresponding 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 $u_1 \longrightarrow v_1$ and $u_2 \longrightarrow v_2$ have the same *type*. Edge types are defined as follows.

Due to the presence of ϕ vertices in *PRG*s, each use of a variable in a non-ϕ vertex is reached by exactly one (ϕ or non-ϕ) assignment to that variable. Therefore, if the operator in a non-ϕ vertex is an *n*-ary operator, there are exactly *n* incoming flow dependence edges for this vertex. These flow dependence edges are assigned types $flow_1$, $flow_2$, . . ., $flow_n$, one for each operand.

A vertex *u* labeled "ϕ_{if}: $x := x$" has two incoming flow dependence edges: one represents the value that reaches *u* from or via the *true* branch of the associated *if* statement; the other represents the value that reaches *u* from or via the *false* branch. The flow dependence edges incident on a ϕ_{if} vertex are assigned types $flow_{true}$ and $flow_{false}$, respectively. For instance, consider the following program fragment:

```
<T1>    x := 1
        if P then
<T2>        x := 2
        fi
<T3>    φ_if: x := x
```

The assignment at T1 reaches T3 via the *false* branch of the *if* statement; so the flow dependence edge from T1 to T3 has type $flow_{false}$. The assignment at T2 reaches T3 from the *true* branch; so the flow dependence edge from T2 to T3 has type $flow_{true}$.

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

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

The basic partitioning algorithm:

```
The initial partition is B[1], B[2], ..., B[p]
WAITING := { 1, 2,..., p }
q := p
while WAITING ≠ ∅ do
    select and delete an integer i from WAITING
    for each edge type m do
        FOLLOWER := ∅
        for each vertex u in B[i] do FOLLOWER := FOLLOWER ∪ m-successor(u) od
        for each j such that B[j] ∩ FOLLOWER ≠ ∅ and B[j] ⊄ FOLLOWER do
            q := q + 1
            create a new class B[q]
            B[q] := B[j] ∩ FOLLOWER
            B[j] := B[j] − B[q]
            if j ∈ WAITING then add q to WAITING
            else if size(B[j]) ≤ size(B[q]) then add j to WAITING
                 else  add q to WAITING
                 fi
            fi
        od
    od
od
```

Figure 4. The basic partitioning algorithm. This algorithm finds the coarsest partition of a graph's vertices that is compatible with a given initial partition and the edges in the graph.

Control dependence edges are assigned types as follows: All vertices except ϕ_{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 $control_{loop}$. The incoming control dependence edge of a ϕ_{enter} vertex u whose source is *not* the associated *while* predicate for u is assigned type $control_{enter-true}$ or $control_{enter-false}$ depending on whether the label on the control dependence edge is *true* or *false*. All other control dependence edges are assigned type $control_{true}$ or $control_{false}$ depending on whether the label on the control dependence edge is *true* or *false*.

The two partitioning passes of the Sequence-Congruence Algorithm both use the basic partitioning algorithm shown in Figure 4. This algorithm is adapted from [4,1], which in turn is based on an algorithm of [10] for minimizing a finite state machine. This algorithm finds the coarsest stable partition of a graph's vertices (*i.e.*, a partition that is compatible with a given initial partition and the edges in the graph). 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 corresponding predecessor u' of v' such that u and u' are in the same class after partitioning, and *vice versa*. The m-successors of a vertex u are the vertices v such that there is an edge $u \longrightarrow v$ of type m. The *size* function returns the number of elements in a class.

Figure 5 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 *PRG*s are treated as one graph; thus, when we refer below to "the graph," we mean the collection of *PRG*s.

Pass 1:

For the first pass, some additional edges are added to the graph: an edge from every *if* predicate to each associated ϕ_{if} vertex and an edge from every *while* predicate to each associated ϕ_{exit} vertex are added to the *PRG*s. These added edges are assigned types $flow_{if}$ and $flow_{while}$, respectively.

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* vertices for x; there is a class for all non-ϕ vertices that have the same operators; for each nesting level of *while* loops, there is a class for all the ϕ_{enter}

The Sequence-Congruence Algorithm:

```
Pass 1:  Add a flow-if edge from every if predicate to each associated $\phi_{if}$ vertex.
         Add a flow-while edge from every while predicate to each associated $\phi_{exit}$ vertex.
         Create an initial partition using the operators in the vertices.
         Apply the basic partitioning algorithm to refine the initial partition, ignoring all control dependence edges.
         Discard all flow-if and flow-while edges.
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 5. The Sequence-Congruence Algorithm. The Sequence-Congruence Algorithm consists of two passes. Both passes use the basic partitioning algorithm shown in Figure 4; only the starting partition and the edges considered in the two passes are different.

vertices at that nesting level; there is a class for all the ϕ_{exit} vertices; there is a class for all the ϕ_{if} vertices; there is a class for all the **output** vertices.

This initial partition is refined by the basic partitioning algorithm; however, all control dependence edges are ignored in the first pass. (The edges added in the beginning of the first pass—those of types $flow_{if}$ and $flow_{while}$—are discarded at the end of the first pass.)

*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.

In the worst case, the Sequence-Congruence Algorithm requires $O(E_1 \log E_1 + E_2 \log E_2)$ time where E_1 is the number of flow dependence edges plus the number of ϕ_{if} and ϕ_{exit} vertices, and E_2 is the number of control dependence edges in the graph.

Example. Figure 6 shows an example of partitioning (not all partition classes are shown in Figure 6; only those that stem from the three classes listed under "Initial Partition"). Note that vertex "$i := i + 2$" is separated from vertices "$j := i + 2$" and "$k := i + 2$" during Pass 1 because the flow predecessor of vertex "$i := i + 2$" is a ϕ_{exit} vertex, while the other two vertices have a ϕ_{enter} vertex as their flow predecessor. This separation causes the vertex "**output**(i)" to be separated from the vertices "**output**(j)" and "**output**(k)" during Pass 1 as well. Note also that vertex "$k := i + 2$" is separated from vertex "$j := i + 2$" in the final partition, because these two vertices have inequivalent control predecessors. If the Sequence-Congruence Algorithm used a *single* partitioning pass that considered all edge types simultaneously, then the final partitions would be refinements of the partitions created using two passes (*i.e.*, the algorithm would still be safe, but it would be more conservative). For example, if a single partitioning pass were used the vertices "**output**(j)" and "**output**(k)" would be placed in different final partitions because their flow predecessors, "$j := i + 2$" and "$k := i + 2$", respectively, are placed in different partitions.

6.2. The Sequence-Congruence Theorem

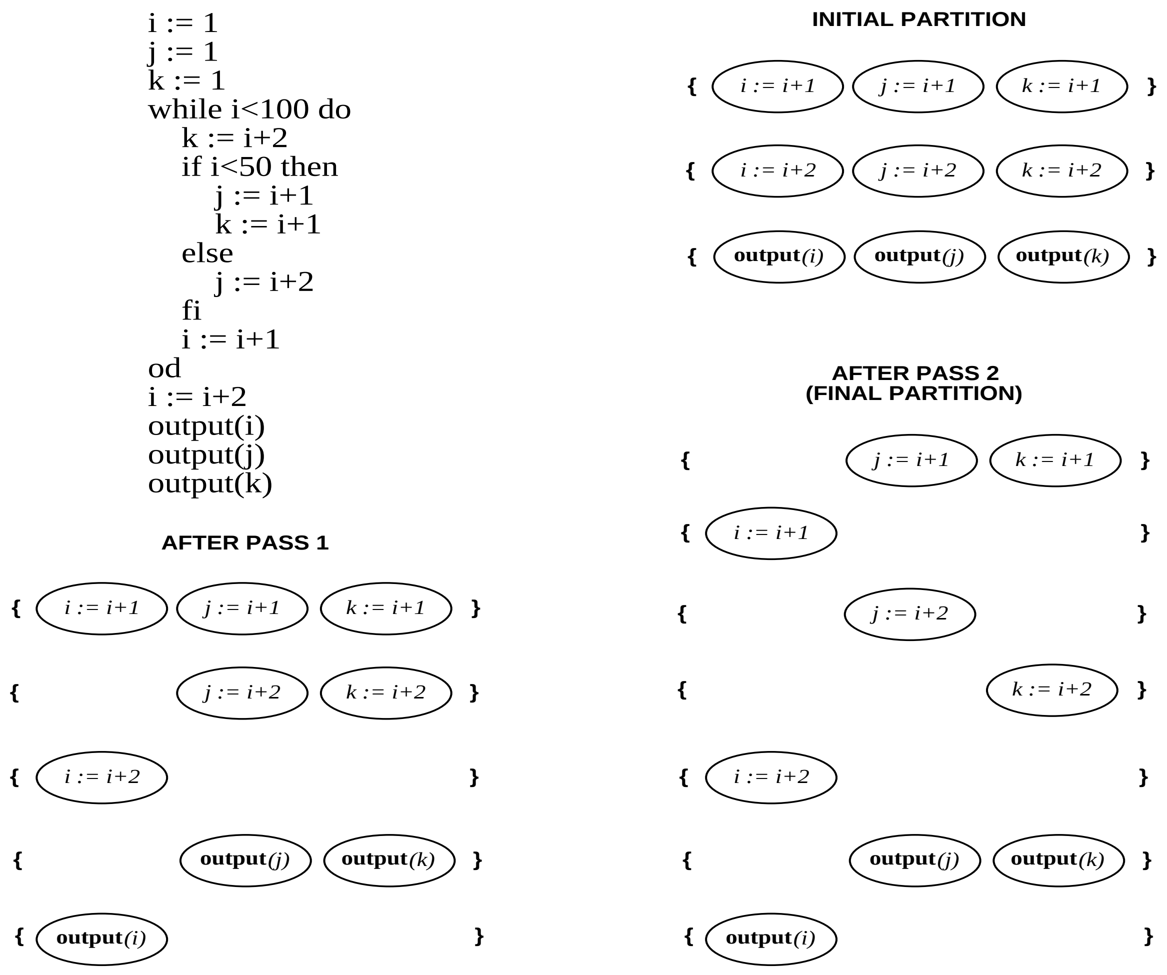
It is possible to show that the Sequence-Congruence Algorithm is a safe algorithm for identifying program components that have equivalent behaviors. This is summarized in the following theorem.

THEOREM. (Sequence-Congruence Theorem). *Program components that are in the same final partition determined by the Sequence-Congruence Algorithm have equivalent execution behaviors.*

For the sake of brevity, the proof of the theorem is not included in this paper; full details can be found in [25]. (A rough outline of the proof is as follows: Suppose the theorem is not correct. Then there are program components that are in the same final partition but have inequivalent execution behaviors. We can find the "earliest" time t when the theorem fails; *i.e.*, when two components that are in the same final partition produce different values. We then show that for the theorem to fail at time t, the theorem must have already failed before time t, which contradicts the choice of t.)

6.3. Enhancements

In this section we consider several enhancements to the Sequence-Congruence Algorithm. The first is concerned with simple assignment statements, simple predicates, and simple output statements. Due to the property of the *identity* operator in a simple vertex, we can merge a simple vertex v with its flow predecessor u before performing the first pass of partitioning. By "merging a vertex v with another vertex u" we mean "replace every edge $v \longrightarrow x$ with an edge $u \longrightarrow x$, remove edge $u \longrightarrow v$, and remove vertex v." This merge operation is undone before the second pass, but vertices u and v are left in the same partition.



```
i := 1
j := 1
k := 1
while i<100 do
   k := i+2
   if i<50 then
      j := i+1
      k := i+1
   else
      j := i+2
   fi
   i := i+1
od
i := i+2
output(i)
output(j)
output(k)
```

Figure 6. An example illustrating how the Sequence-Congruence Algorithm partitions the vertices of a (single) program into equivalence classes. Nine vertices of the example program are shown in their classes in the initial partition, after Pass 1, and after Pass 2.

Vertices *u* and *v* may or may not be put in different partitions during the second pass. For instance, consider the following example:

<T1>	$a := 1$	<T3>	$c := 1$
<T2>	$b := a + 2$	<T4>	$d := c$
		<T5>	$e := d + 2$

If we merge the simple assignment statement T4 with its flow predecessor T3 before performing the first pass of partitioning, we can discover that T2 and T5 are have equivalent behaviors.

In the Sequence-Congruence Algorithm, we assume that statement "$x := a + b * c$" has the same operator as statement "$y := d + e * f$" but a different operator than statement "$z := g * h$"; that is, the structure of the right-hand-side expression defines the operator. The expression "$a + b * c$" uses the operator that takes three arguments a, b, and c, and returns the value of "$a + b * c$". Thus, in the following program fragment, T1 and T2 are not in the same equivalence class because they have different operators.

```
<T1>    x := a + b * c
        z := b * c
<T2>    y := a + z
```

We can detect larger equivalence classes if the program is transformed to three-address code before partitioning. For the above example, the assignment to x is replaced by two statements when the program fragment is transformed to three-address code; consequently, T3 and T4 are placed in the same equivalence class by the Sequence-Congruence Algorithm.

```
        temp := b * c
<T3>    x := a + temp
        z := b * c
<T4>    y := a + z
```

Similarly, a constant inside an expression is tightly coupled with the operator. The expression "$a + 1$" uses the unary operator that takes an argument a and returns the value of "$a + 1$". Therefore, in the following program fragment, T5 and T6 are not in the same equivalence class because they have different operators (and different numbers of incoming flow dependence edges).

```
<T5>    x := a + 1
        z := 1
<T6>    y := a + z
```

A simple transformation can improve the result of partitioning: for each constant c that appears in the program, (1) a new variable *Const_c* is created, (2) an assignment statement "*Const_c* := c" is added at the very beginning of the program, and (3) all references to c in the program are changed to references to *Const_c*. This transformation does not change the execution behavior of a program; however, larger equivalence classes of components that has equivalent behaviors will result from partitioning.

As presented in Section 6.1, the Sequence-Congruence Algorithm uses the same basic partitioning algorithm—the one given in Figure 4—for both Pass 1 and Pass 2; however, this is not strictly necessary. In principle, it is possible to use variants of the basic partitioning algorithm tailored specially for the two different passes. For example, one kind of enhancement that may be worthwhile incorporating into Pass 1 is one that takes into account the mathematical properties of an expression's operator. For instance, consider the following example:

```
<T1>    a := 1                    <T5>    c := 2
<T2>    b := 2                    <T6>    d := 1
<T3>    x := a + b                <T7>    u := c + d
<T4>    y := x * 3                <T8>    v := u * 3
```

With the present algorithm for Pass 1, T3 and T7 are eventually placed in separate classes, and hence T4 and T8 are also placed in separate classes in the first pass. However, because addition is commutative, T3 and T7 could be placed in a single equivalence class, which then also makes it possible for T4 and T8 to be members of a single equivalence class. A simple enhancement to the basic partitioning algorithm extends it to handle commutative operators [4].

The benefits of finding larger equivalence classes during Pass 1 carry over to Pass 2; in this example, T3 and T7 would be members of the same equivalence class, and T4 and T8 would be members of another.

It is obvious that the Sequence-Congruence Algorithm could also benefit from additional information supplied by the programmers. Such information could be furnished through pragmas or could be obtained through interaction with the programmers. For instance, if it is asserted that two components have equivalent behavior, the technique of *congruence closure* [7,17] can merge the equivalence classes to which the two components belong and propagate the effects to combine other equivalence classes.

7. Comparison With Related Work

In addition to the HPR algorithm [12], there has been previous work on integrating functions [5], logic programs [15], and specifications [8]. Different models of integration have been used in each case. In Berzins's work on integrating functions, variants *A* and *B* are merged without regard to *Base*. The function that results from the merge preserves the (entire) behavior of *both*; thus, *A* and *B* cannot be merged if they conflict at any point where both are defined. Similarly, Lakhotia and Sterling's 1−1 join operation is a two-way merge. However, in their work there is no notion of interference, and the characterization of the semantic properties of the merged program was left as an open question in [15]. Feather's work on integrating specifications does take *Base* into account, but although the integration algorithm preserves syntactic modifications, it does not guarantee any semantic properties of the integrated specification. Both the HPR algorithm and the algorithm described in this paper are three-way integration operations that satisfy the semantic criterion stated in Section 1.

The program-integration algorithm presented in this paper addresses an important limitation of the HPR algorithm, namely, its inability to integrate program variants when a change made in one of the variants is a semantics-preserving transformation of some part of the base program, and a change made in the other variant uses the result computed by that part of the program. The key idea of the new algorithm is the use of limited slices—rather than full slices—to extract what is changed in each of the variants.

Because the new integration algorithm is parameterized by the auxiliary algorithm used for identifying congruent program components, we have actually defined a *class* of integration algorithms that accommodate semantics-preserving transformations. Any technique for identifying congruent vertices that meets the conditions enumerated in Section 4.1 can be used in conjunction with the integration algorithm described in this paper. Thus, applying the Sequence-Congruence Algorithm, which was discussed in Section 6, is just one of the ways in which the first step of the new integration algorithm could be implemented.

The Sequence-Congruence Algorithm is not the first to address the problem of equivalent execution behaviors; for example, it is shown in [19] that two components with isomorphic slices have equivalent execution behaviors (the slice of a program with respect to a program component *c* is, roughly, all the statements and predicates in the program that can potentially affect the values produced at *c* during program execution). Further, this result holds even if the two components happen to be in *two different programs*, provided the two programs are run on identical—or actually just sufficiently similar—initial states. However, the Sequence-Congruence Algorithm is both more powerful and more efficient than the method based on comparing slices. All components (in the same or in different programs) with isomorphic slices are identified as equivalent by the Sequence-Congruence Algorithm, but not all components identified as equivalent by the Sequence-Congruence Algorithm have isomorphic slices. Partitioning the components of one or more programs into equivalence classes can be done using the Sequence-Congruence Algorithm in worst-case time $O(N \log N)$ (where N is the sum of the programs' sizes); such a partitioning requires worst-case time $O(N^2)$ using the program slice comparison technique of [13].

We have shown that when the Sequence-Congruence Algorithm is used for congruence testing, the new integration algorithm is strictly better than the HPR algorithm in the following sense [25]:

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

(2) The integrated program produced by the new algorithm satisfies the integration criteria stated in Section 1.

(3) There are integration problems on which the new algorithm succeeds while the HPR algorithm reports interference.

For instance, while the HPR algorithm reports interference for all the examples of Figure 1, the new integration algorithm, used in conjunction with the Sequence-Congruence Algorithm, succeeds on the first example. (The second and third examples of Figure 1 illustrate limitations of the Sequence-Congruence Algorithm, which fails to identify all the unchanged components in those two examples.)

The development of more powerful congruence-testing algorithms (and hence more powerful program-integration algorithms) is a subject of on-going research. Relevant work on program-optimization and transformation techniques for graph representations similar to PRGs includes [22], [9], and [21].

APPENDIX. PROOF OF THE INTEGRATION THEOREM

This appendix gives the detailed proof of the integration theorem stated in Section 5. 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.

By the construction of R_M, every 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.

A vertex v in R_M may have a different text from one of its originating vertices, but the text of v must match at least *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 have equivalent behavior.

Every edge $u \longrightarrow v$ of R_M is taken from either R_A or R_B or both. (It is possible that the edge $u \longrightarrow v$ appears in R_{Base} as well.) Since each control or flow dependence edge is identified by its two end-points, if an edge $u \longrightarrow v$ of R_M is taken from R_A (or R_B), then there are originating vertices u' and v' of u and v, respectively, and an identical control or flow dependence edge $u' \longrightarrow v'$ in R_A (or R_B, respectively). In addition, it can be shown (by case analysis on the classification of v') that v' and v have the same text.

We first prove an auxiliary lemma that will be used in the proofs of the remaining lemmas in this section.

LEMMA 1. *Suppose s_1 and s_2 are two sequences of boolean values. If (1) either s_1 is a prefix of s_2 or* vice versa, *(2) the last values of s_1 and s_2 are* true, *and (3) s_1 and s_2 have the same number of* true *values, then s_1 and s_2 are identical sequences.*

PROOF. Suppose s_1 and s_2 are not identical. Without loss of generality, we may assume s_1 is a proper prefix of s_2. If s_2 has k *true* values, then s_1 can have at most $k - 1$ *true* values since the last value of s_2, which is *true*, does not appear in s_1. But this contradicts the assumption that s_1 and s_2 have the same number of *true* values. Therefore, s_1 and s_2 are identical sequences. □

The proof of the Integration Theorem proceeds by first showing that every vertex of R_M produces the same sequence of values as its originating vertices when the merged program M is run on an initial state σ on which A, B, and $Base$ all terminate normally. Then we show that the merged program M must also terminate normally on an initial state σ when A, B, and *Base* all terminate normally on σ. Finally, we prove the existence of vertices with the changed and preserved behaviors in M.

LEMMA 2. *Suppose A and B are two variants of Base for which the new integration algorithm succeeds and produces a merged program M. When M is run on an initial state σ on which A, B, and Base all terminate normally, every program component of M produces the same sequence of values as its originating component in A or B.*

PROOF. We prove this lemma by contradiction. Suppose it is not the case that every vertex of R_M produces the same sequence of values as its originating vertex. We choose a vertex u_M in M such that u_M produces a different sequence of values than its originating vertex and each of u_M's control predecessors, v_M, produces the same sequence of values as v_M's originating vertex. Note that it is always possible to choose such a u_M because there is a control dependence path from the *Entry* vertex to every other vertex in the *PRG* and all *Entry* vertices always produce the same (sequence of) values.

There are two ways in which u_M and its originating vertex could produce different sequences of values: (1) there is a constant k such that the k^{th} values produced at u_M and its originating vertex differ, and (2) the sequence of values produced at u_M is a proper prefix of that produced at its originating vertex. In the latter case, because u_M's control predecessor, v_M, produces the same sequence of values as v_M's originating vertex, there must be a non-terminating or faulting computation w_M that is executed between v_M and u_M. If this is the case, we choose w_M instead of u_M. Therefore, we can always find a vertex u_M in M and a constant k such that the k^{th} value produced at u_M differs from the k^{th} value produced at its originating vertex.

Let t_M be the moment just after u_M executes for the k^{th} time. If there are many u_M in M and k such that the k^{th} value produced at u_M differs from that produced at its originating vertex, choose the ones with the earliest t_M.

If u_M has no flow predecessor, that is, u_M is a constant vertex,[6] then at least one of its originating vertices, say u_A in A, must be an identical constant vertex. Because u_M and u_A are identical constant vertices, they cannot produce different values. Thus, u_M cannot be a constant vertex.

Let v_M be a flow predecessor of u_M. Without loss of generality, assume the flow edge $v_M \longrightarrow_f u_M$ is taken from A; that is, there is a corresponding flow edge $v_A \longrightarrow_f u_A$ in A and v_A and u_A are originating vertices of v_M and u_M, respectively. Let t_A be the moment just after u_A executes for the k^{th} time.

Since u_M and u_A produce different values at t_M and t_A, respectively, we may assume the values of v_M and v_A at t_M and t_A,[7] respectively, are different (for if all corresponding flow predecessors of u_M and u_A have the same value at t_M and t_A, respectively, then the values of u_M and u_A at t_M and t_A must be the same). However, since t_M is the earliest time when the lemma fails and v_A is an originating vertex of v_M, the only way that the values of v_M and v_A at t_M and t_A could be different is that v_M and v_A have executed a different number of times by the times t_M and t_A. In what follows, we will show that this cannot happen.

Let w_M be the least common control ancestor of u_M and v_M in M; *i.e.*, w_M is a common control ancestor of u_M and v_M in M, and all other common control ancestors of u_M and v_M are control ancestors of w_M (*while* predicates are not considered to be control ancestors of themselves). Let w_A be the least common

[6]An assignment vertex is a constant vertex if the right-hand-side expression consists of a single constant; a predicate vertex is a constant vertex if the expression consists of a constant boolean value. An output statement is a constant vertex if it prints out the value of a constant.

[7]The value of a vertex v at time t is the most recent value produced at v by the time t.

control ancestor of u_A and v_A in A. Note that w_A might not be the originating vertex of w_M. Depending on whether the whole control dependence path $w_M \longrightarrow_c^* u_M$ is taken from A we have two cases.

Case 1. The whole control dependence path $w_M \longrightarrow_c^* u_M$ is taken from A.

We have the situation as shown in Figure A1. Note that at times t_M and t_A, both u_M and u_A have produced k values; all but the k^{th} values produced are pairwise identical. We want to show that w_M and w_A have produced the same sequence of values by the times t_M and t_A, respectively.

- First assume that u_M and u_A are neither ϕ_{enter} vertices nor *while* predicate vertices; thus, u_M and u_A have exactly one incoming control edge. Let x_M and x_A be the control predecessors of u_M and u_A, respectively, and assume that the control edges $x_M \longrightarrow_c u_M$ and $x_A \longrightarrow_c u_A$ are both labeled *true*. Since x_A is an originating vertex of x_M, and the lemma does not fail until time t_M, either the sequence of values produced by x_M by the time t_M is a prefix of the sequence of values produced by x_A by the time t_A, or *vice versa*.

  At time t_M, u_M has executed k times and the control edge $x_M \longrightarrow_c u_M$ is labeled *true*; thus, by the time t_M, x_M has produced a sequence of (boolean) values, k of which are *true* and the last value in the sequence is *true*. Similarly, by the time t_A, x_A has produced a sequence of (boolean) values, k of which are *true* and the last value in the sequence is *true*. By Lemma 1, we conclude that by the times t_M and t_A, x_M and x_A have produced the same sequence of values.

- Next assume that u_M and u_A are *while* predicate vertices; thus, u_M and u_A have one incoming control edge in addition to the self-loops . Let x_M and x_A be the control predecessors of u_M and u_A, respectively, along the control edges that are not the self-loops and assume the control dependence edges $x_M \longrightarrow_c u_M$ and $x_A \longrightarrow_c u_A$ are both labeled *true*. Since x_A is an originating vertex of x_M, and the lemma does not fail until time t_M, either the sequence of values produced by x_M by the time t_M is a prefix of the sequence of values produced by x_A by the time t_A, or *vice versa*.

  By the times t_M and t_A, u_M and u_A have produced k values, of which all but the last ones are pairwise identical. Thus, the loops of u_M and u_A must have performed the same number of times by the times t_M and t_A. Therefore, x_M and x_A must have produced the same number of *true* values, and the most recent values produced at both x_M and x_A are *true*. By Lemma 1, x_M and x_A must have produced identical sequence of values by the times t_M and t_A, respectively.

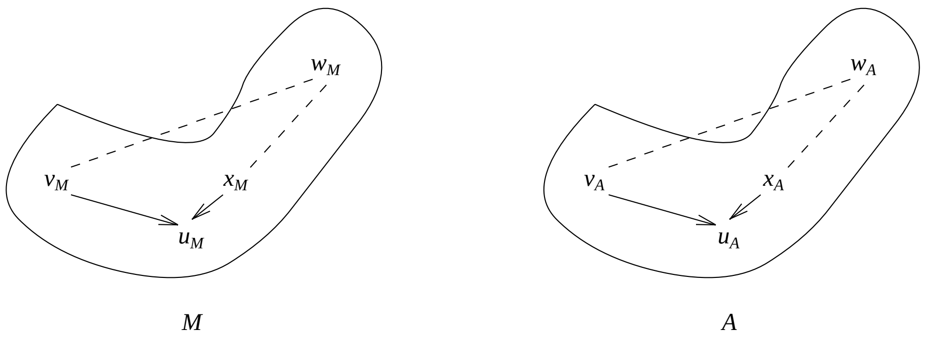


Figure A1.

- Next assume that u_M and u_A are ϕ_{enter} vertices and the flow dependence edge $v_M \longrightarrow_f u_M$ is from outside the loop; thus, u_M and u_A have two incoming control dependence edges. Let x_M and x_A be the associated *while* predicates of u_M and u_A, respectively. Let y_M and y_A be the other control predecessors of u_M and u_A, respectively, and assume that the control edges $y_M \longrightarrow_c u_M$ and $y_A \longrightarrow_c u_A$ are both labeled *true*. We have the situation as shown in Figure A2.

  Note that x_A and y_A are originating vertices of x_M and y_M, respectively. Since the lemma does not fail until the time t_M, either the sequence of values produced by y_M by the time t_M is a prefix of that produced by y_A by the time t_A, or *vice versa*. Also note that, by the times t_M and t_A, the last values produced at y_M and y_A are *true* since the control edges $y_M \longrightarrow_c u_M$ and $y_A \longrightarrow_c u_A$ are labeled *true*.

  Because x_A is an originating vertex of x_M, and because the lemma does not fail until the time t_M, either the sequence of values produced by x_M by the time t_M is a prefix of that produced by x_A by the time t_A, or *vice versa*. Note that y_M and y_A are also control predecessors of x_M and x_A.

  Since u_M is a ϕ_{enter} vertex, the number of times u_M has executed equals the number of *true* values produced at y_M plus the number of *true* values produced at x_M. Similarly, the number of times u_A has executed equals the number of *true* values produced at y_A plus the number of *true* values produced at x_A.

  Suppose the sequences of values produced at y_M and y_A by the times t_M and t_A are not identical. The sequence of values produced at y_M is a *proper* prefix of that produced at y_A, or *vice versa*. If the sequence of values produced at y_M is a proper prefix of that produced at y_A, then the sequence of values produced at x_M is also a proper prefix of that produced at x_A. On the other hand, if the sequence of values produced at y_A is a proper prefix of that produced at y_M, then the sequence of values produced at x_A is also a proper prefix of that produced at x_M. In either case, since the last values produced at y_M and y_A are *true*, the total number of *true* values produced at y_M and x_M cannot be equal to that produced at y_A and x_A. Therefore, u_M and u_A could not have executed the same number of times by t_M and t_A. However, this contradicts the assumption that, by the times t_M and t_A,

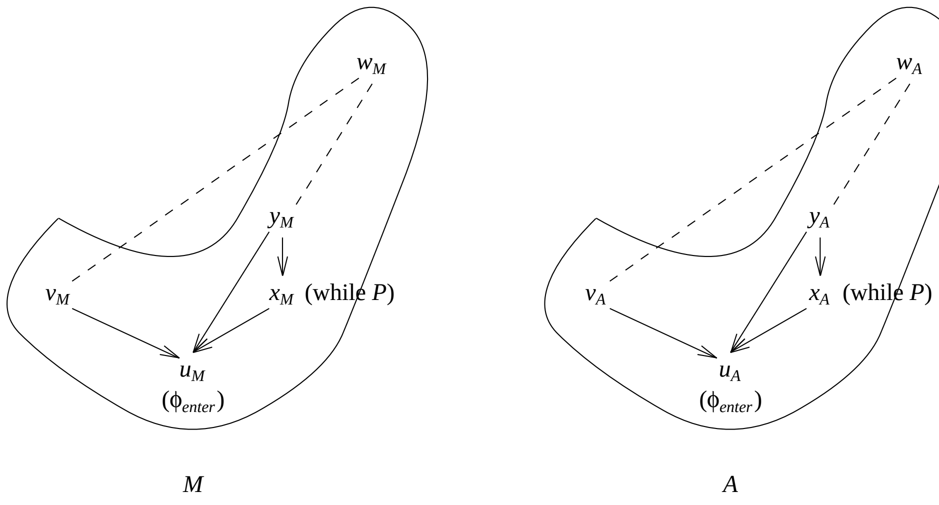


Figure A2.

both u_M and u_A have executed k times. We conclude that y_M and y_A must have produced identical sequence of values by the times t_M and t_A.

Because y_M and y_A have produced the same sequence of values by the times t_M and t_A, and because u_M and u_A have executed the same number of times, x_M and x_A must have also produced identical sequence of values by the times t_M and t_A.

- Lastly, assume that u_M and u_A are ϕ_{enter} vertices and the flow dependence edge $v_M \longrightarrow_f u_M$ is from inside the loop. Let x_M and x_A are the associated *while* predicates. Note that in this case, w_M and x_M are the same vertex; w_A and x_A are the same vertex. We have the situation as shown in Figure A3.

  Note that t_M is the moment immediately after u_M executes for the k^{th} time. Thus, by the time t_M, x_M has executed $k - 1$ times. Similarly, t_A is the moment immediately after u_A executes for the k^{th} time. By the time t_A, x_A has executed $k - 1$ times. Therefore, x_M and x_A have executed the same number of times.

  Since x_A is an originating vertex of x_M and the lemma does not fail until the time t_M, x_M and x_A must have produced the same sequence of values by the times t_M and t_A, respectively.

We conclude that by the times t_M and t_A, x_M and x_A, which are control predecessors of u_M and u_A, have produced the same sequence of values.

By repeating the above arguments for each pair of corresponding control ancestors of u_M and u_A, we know that w_M and w_A have produced the same sequence of values by the times t_M and t_A.

Next we will do a case analysis on u_M and u_A. In each subcase, we will show that v_M and v_A have produced the same sequence of values by the times t_M and t_A (because w_M and w_A have produced the same sequence of values).

Subcase 1. u_M and u_A are non-ϕ vertices. In this case, there are control edges $w_M \longrightarrow_c v_M$ and $w_A \longrightarrow_c v_A$. We have the situation as shown in Figure A4.

Since w_M and w_A have produced the same sequence of values by the times t_M and t_A, v_M and v_A must have executed the same number of times. Since v_A is an originating vertex of v_M and the lemma does not fail until the time t_M, it must be that, by the times t_M and t_A, v_M and v_A have produced the same sequence of values.

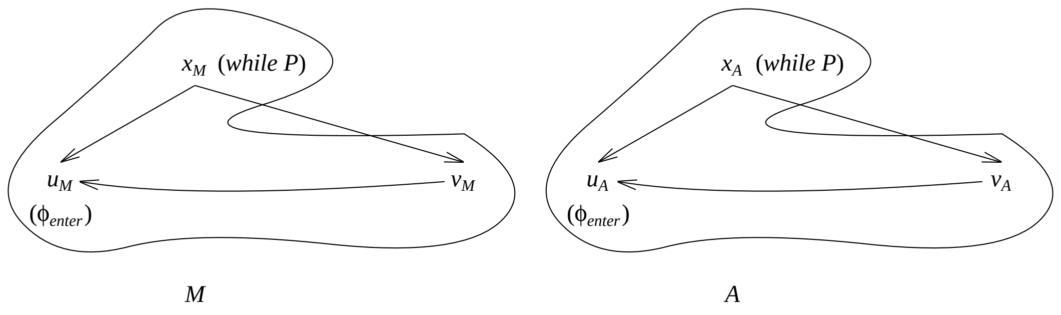


Figure A3.

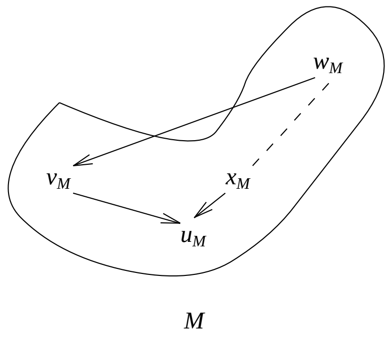


Figure A4.

Subcase 2. u_M and u_A are ϕ_{if} vertices and there is a control edge $w_M \longrightarrow_c v_M$. Since v_A is an originating vertex of v_M, there is a control edge $w_A \longrightarrow_c v_A$. We have the situation as shown in Figure A4. By the same argument as in *Subcase 1* above, we know that, by the times t_M and t_A, v_M and v_A have produced the same sequence of values.

Subcase 3. u_M and u_A are ϕ_{if} vertices and there is an *if* predicate P_M on the control dependence path such that $w_M \longrightarrow_c P_M \longrightarrow_c v_M$. Since v_A is an originating vertex of v_M, there must be an *if* predicate P_A on the control dependence path such that $w_A \longrightarrow_c P_A \longrightarrow_c v_A$. We have the situation as shown in Figure A5.

There are two possibilities depending on whether P_A is an originating vertex of P_M.

- Suppose P_A is an originating vertex of P_M. Since w_M and w_A have produced the same sequence of values by the times t_M and t_A, P_M and P_A must have executed the same number of times. Furthermore, since P_A is an originating vertex of P_M and the lemma does not fail until the time t_M, P_M and P_A must have produced the same sequence of values by the times t_M and t_A. Similarly, v_M and v_A must have produced the same sequence of values by the times t_M and t_A.
- Suppose P_A is not an originating vertex of P_M. Then there must be originating vertices w_B, P_B, and v_B in B for w_M, P_M, and v_M, respectively. Because both v_A and v_B are originating vertices of v_M, v_A and v_B are corresponding vertices. Therefore, since P_A and P_B are control predecessors of

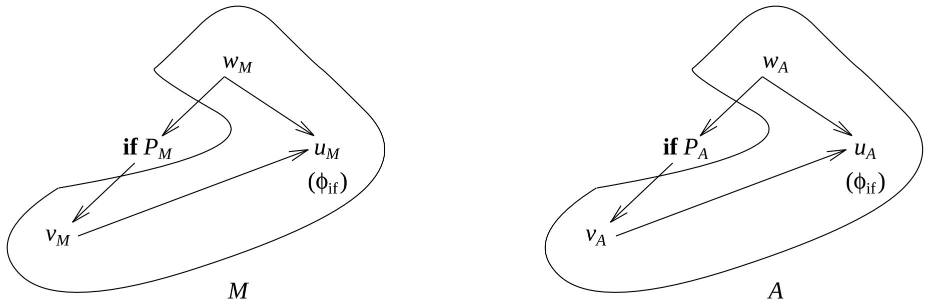


Figure A5.

corresponding vertices, P_A and P_B are congruent vertices (see the definition in Section 4.1). Because both w_A and w_B are originating vertices of w_M, w_A and w_B are corresponding vertices. We have the situation as shown in Figure A6.

Let t_B be the point of time during the execution of B such that v_B has executed the same number of times as v_M at t_M. (We are able to choose such a t_B for otherwise either (1) P_M and P_B would have produced different values earlier than t_M and t_B or (2) w_M and w_B would have produced different values earlier than t_M and t_B. In either case, the lemma would have failed earlier than t_M.) Because v_B is an originating vertex of v_M, v_M and v_B have produced the same sequence of values by the times t_M and t_B; otherwise the lemma would have failed earlier than t_M.

By the times t_M and t_B, v_M and v_B have executed the same number of times; Thus, by the same argument as in *Subcase 1* (where it is shown that when u_M and u_A have executed the same number of times, their control ancestors, w_M and w_A, also have produced the same sequence of values), P_M and P_B have produced the same sequence of values and w_M and w_B have produced the same sequence of values.

Because w_M and w_A have produced the same sequence of values by the times t_M and t_A and w_M and w_B have produced the same sequence of values by the times t_M and t_B, w_A and w_B have produced the same sequence of values by the times t_A and t_B. Because w_A and w_B have produced the same sequence of values by the times t_A and t_B, the two *if* predicates P_A and P_B must have executed the same number of times by the times t_A and t_B. Because P_A and P_B are congruent vertices and they have executed the same number of times, P_A and P_B must have produced the same sequence of values by the times t_A and t_B.

Because P_A and P_B have produced the same sequence of values by the times t_A and t_B, v_A and v_B must have executed the same number of times by the times t_A and t_B. Because v_A and v_B are corresponding vertices and they have executed the same number of times, v_A and v_B must have produced the same sequence of values by the times t_A and t_B. Because v_M and v_B have produced the same sequence of values by the times t_M and t_B, and because v_A and v_B have produced the same sequence of values by the times t_A and t_B, v_M and v_A have produced the same sequence of values by the times t_M and t_A.

Subcase 4. u_M and u_A are ϕ_{enter} vertices and v_M and v_A are flow predecessors of u_M and u_A from outside the loops, respectively. This case is similar to *Subcase 1* above.

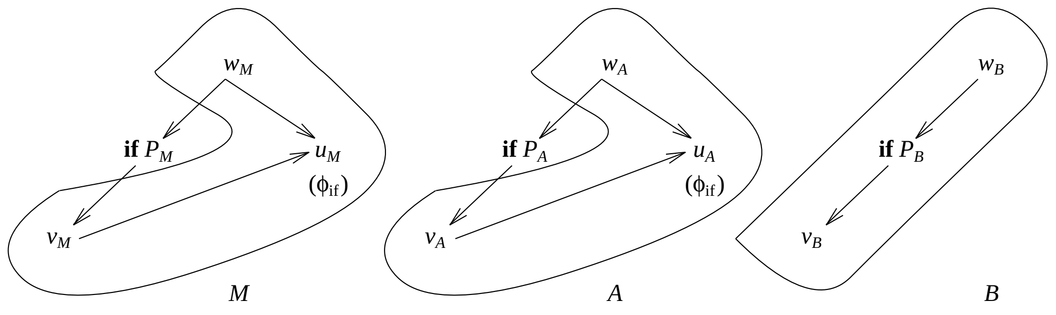


Figure A6.

Subcase 5. u_M and u_A are ϕ_{enter} vertices and v_M and v_A are flow predecessors of u_M and u_A from inside the loops, respectively. We have the situation as shown in Figure A7.

Because by the times t_M and t_A, w_M and w_A have produced the same sequence of values, v_M and v_A must have executed the same number of times. Furthermore, since v_A is an originating vertex of v_M, v_M and v_A must have produced the same sequence of values by the times t_M and t_A for otherwise the lemma would have failed earlier than t_M.

Subcase 6. u_M and u_A are ϕ_{exit} vertices. In this case, v_M and v_A are ϕ_{enter} vertices. The situation is shown in Figure A8.

This case is similar to *Subcase 3* above. There are two cases depending on whether P_A is an originating vertex of P_M. By the same argument as in *Subcase 3* above, v_M and v_A have produced the same sequence of values by the times t_M and t_A.

In each of the above six subcases, v_M and v_A have produced the same sequence of values by the times t_M and t_A. Thus, at times t_M and t_A, u_M and u_A must produce the same values, which contradicts the previous assumption that u_M and u_A produce different values at times t_M and t_A.

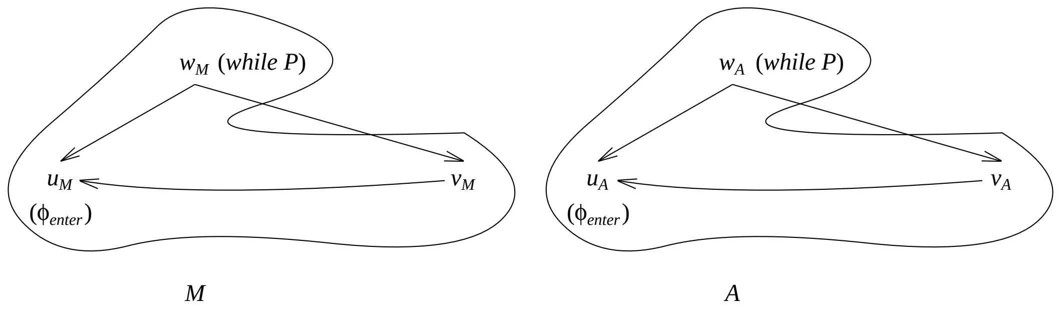


Figure A7.

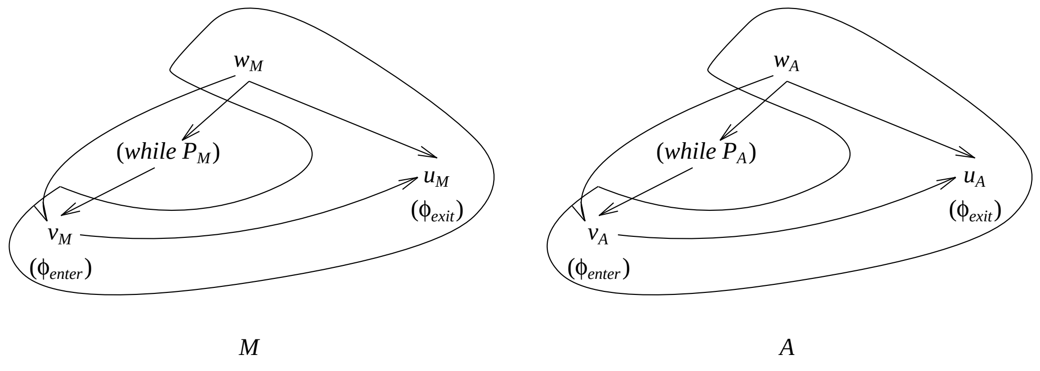


Figure A8.

Case 2. Some of the edges on the control dependence path $w_M \longrightarrow_c^* u_M$ are taken from A; others are taken from B.

We can decompose the control dependence path $w_M \longrightarrow_c^* u_M$ into fragments such that fragments are taken from A and B alternately. The situation is shown in Figure A9.

In Figure A9, the control dependence path $y_M \longrightarrow_c^* u_M$ is taken from A; the control dependence path $z_M \longrightarrow_c^* y_M$ is taken from B, *etc*. Note that these fragments overlap at common predicate vertices and the predicates at which fragments overlap have originating vertices in both A and B. In Figure A9, y_M and z_M are predicates at which fragments overlap and they have originating vertices y_A, y_B, z_A, and z_B, respectively.

Because at times t_M and t_A, u_M and u_A have executed the same number of times, and because the control dependence path $y_M \longrightarrow_c^* u_M$ is taken from A, by the same arguments as in Case 1 above, by the times t_M and t_A, y_M and y_A have produced the same sequence of values.

Let t_B be a point of time during the execution of B such that by the times t_A and t_B, y_A and y_B have produced the same sequence of values. (It is always possible to find such a time t_B since y_A and y_B are corresponding vertices and both A and B terminate normally on the initial state σ.) Thus, at times t_M, t_A, and t_B, y_M, y_A, and y_B have produced the same sequence of values.

Since y_A and y_B are corresponding vertices, each pair of corresponding control ancestors of y_A and y_B must have produced the same sequence of values by the times t_A and t_B. In particular, z_A and z_B have produced the same sequence of values and w_A and w_B have produced the same sequence of values by the times t_A and t_B.

On the other hand, at the times t_M and t_B, y_M and y_B have produced the same sequence of values. Since the control dependence path $z_M \longrightarrow_c^* y_M$ is taken from B, by the same arguments as in Case 1, corresponding control ancestors of y_M and y_B on the paths $z_M \longrightarrow_c^* y_M$ and $z_B \longrightarrow_c^* y_B$ must have produced the same sequence of values. In particular, z_M and z_B have produced the same sequence of values by the times t_M

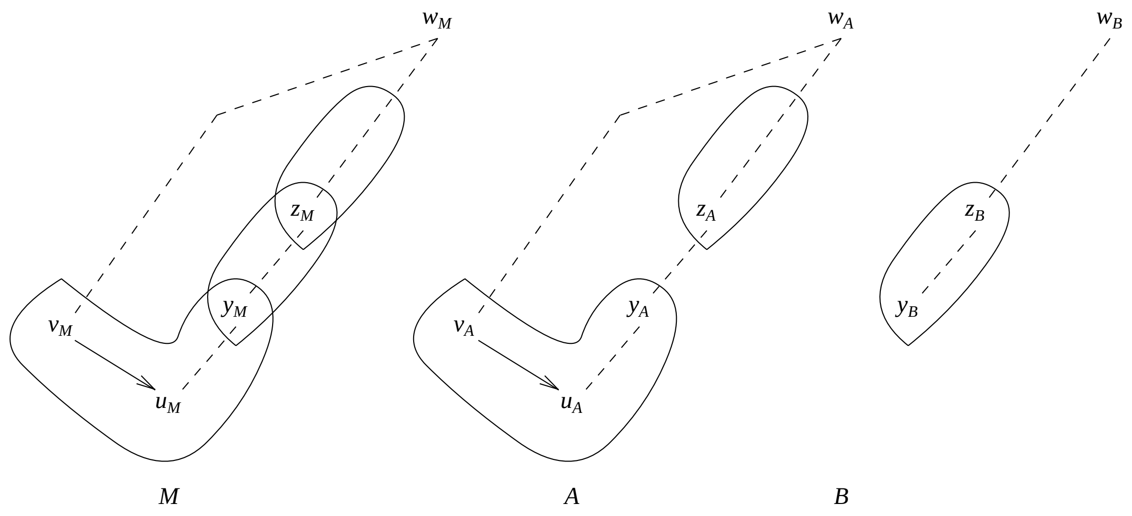


Figure A9.

and t_B. Thus, by the times t_M, t_A, and t_B, z_M, z_A, and $z_B$have produced the same sequence of values.

By repeating the argument in the previous paragraph for each fragment on the control dependence path $w_M \longrightarrow_c^* u_M$, we know that if w_M is in a fragment that is taken from *A*, then w_M and w_A have produced the same sequence of values by the times t_M and t_A; if w_M is in a fragment that is taken from *B*, then w_M and w_B have produced the same sequence of values by the times t_M and t_B. In the latter case, since w_A and w_B have produced the same sequence of values by the times t_A and t_B, w_M and w_A have produced the same sequence of values by the times t_M and t_A. Thus, regardless of whether w_M is taken from *A* or *B*, w_M and w_A must have produced the same sequence of values by the times t_M and t_A.

Since w_M and w_A have produced the same sequence of values by the times t_M and t_A, by the same argument as in Case 1, we know that v_M and v_A have produced the same sequence of values at times t_M and t_A. Consequently, u_M and u_A cannot produce different values at times t_M and t_A. That is, the lemma cannot fail at t_M.

From Case 1 and Case 2, we know u_M and u_A cannot produce different values at times t_M and t_A, respectively. This completes the proof of the lemma. □

LEMMA 3. *Suppose the new integration algorithm successfully integrates two variants A and B with respect to the base program Base and produces a merged program M. Then for any initial state* σ *on which A, B, and Base all terminate normally, M terminates normally on* σ.

PROOF. We prove this lemma by contradiction. Suppose *M* does not terminate normally on σ. 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 *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 Lemma 2, *u* and u_A produce the same sequence of values. The same fault must also occur at u_A. Thus, *A* cannot terminate normally on the initial state σ, which contradicts the assumption that *A* terminates 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 Lemma 2, *u* and u_A produce the same sequence of values. 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. 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 σ. □

LEMMA 4. *Suppose the new integration algorithm successfully integrates two variants A and B with respect to the base program Base and produces a merged program M. Then for any initial state* σ *on which A, B, and Base all terminate normally:*

(1) *For any program component v_A in A, if v_A produces a different sequence of values than the analogous component in Base, then there is a component v in M that produces the same sequence of values as v_A.*

(2) *For any program component v_B in B, if v_B produces a different sequence of values than the analogous component in Base, then there is a component v in M that produces the same sequence of values as v_B.*

(3) *For any program component v_{Base} in Base, if v_{Base} produces the same sequence of values as the analogous components in both A and B, then there is a component v in M that produces the same sequence of values as v_{Base}.*

Proof. Suppose v_A is a component of *A* that produces a different sequence of value than the analogous component in *Base*. Then $v_A \in New_A$. By the construction of *M*, v_A is an originating vertex of a vertex *v* in *M*. By Lemma 2, since *A*, *B*, and *Base* all terminate normally, *v* and v_A produce the same sequence of values. This proves the first clause. The second clause can be proved by the same argument.

Suppose v_{Base} is a component of *Base* that produces the same sequence of values as the analogous components, v_A and v_B, in *A* and *B*, respectively. Then either $v_{Base} \in Unchanged$ or $v_A \in Affected_A$ or $v_B \in Affected_B$ depending on whether v_{Base}, v_A, and v_B are put in the same congruence classes in the first step of the new integration algorithm and depending on whether they have the same text. If $v_{Base} \in Unchanged$, then v_A and v_B are originating vertices of a vertex *v* in *M*. By Lemma 2, since *A*, *B*, and *Base* all terminate normally, *v* produces the same sequence of values as v_A and v_B. Because v_{Base} also produces the same sequence of values as v_A and v_B, the sequences of values produced at *v* and v_{Base} must be identical.

If $v_A \in Affected_A$, then v_A is an originating vertex of a vertex *v* in *M*. By Lemma 2, since *A*, *B*, and *Base* terminate normally, *v* produces the same sequence of values as v_A. Because v_{Base} also produces the same sequence of values as v_A, the sequences of values produced at *v* and v_{Base} are identical.

By the same argument, we know if $v_B \in Affected_B$, then there is a component *v* in *M* that produces the same sequence of values as v_{Base}. This proves the last clause. □

The Integration Theorem follows immediately from Lemmas 3 and 4.

ACKNOWLEDGEMENT

Thanks are due to Bowen Alpern for his participation in early discussions that led to the development of the Sequence-Congruence Algorithm.

References

1. Aho, A., Hopcroft, J.E., and Ullman, J., *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).
2. Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).
3. Allen, F.E. and Cocke, J.A., "A catalogue of optimizing transformations," pp. 1-30 in *Design and Optimization of Compilers*, ed. R. Rustin,Prentice-Hall, Englewood Cliffs, NJ (1972).
4. Alpern, B., Wegman, M.N., and Zadeck, F.K., "Detecting equality of variables in programs," pp. 1-11 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), (1988).
5. Berzins, V., "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), (1989).
7. Downey, P.J., Sethi, R., and Tarjan, R.E., "Variations on the common subexpression problem," *JACM* **27**(4) pp. 758-771 (1980).
8. Feather, M.S., "Detecting interference when merging specification evolutions," Unpublished report, Information Sciences Institute, University of Southern California, Marina del Rey, CA (1989).
9. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

10. Hopcroft, J.E., “An *n* log *n* algorithm for minimizing the states of a finite automaton,” *The Theory of Machines and Computations*, pp. 189-196 (1971).

11. Horwitz, S., Prins, J., and Reps, T., *On the suitability of dependence graphs for representing programs*, Department of Computer Sciences, University of Wisconsin—Madison (August 1988).

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

13. Horwitz, S. and Reps, T., “Efficient comparison of program slices,” Technical Report 982, Department of Computer Sciences, University of Wisconsin—Madison (December 1990).

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

15. Lakhotia, A. and Sterling, L., “Composing recursive logic programs with clausal join,” *New Generation Computing* **6**(2) pp. 211-225 (1988).

16. Loveman, D. B., “Program Improvement by Source-to-Source Transformation,” *JACM* **20**(1) pp. 121-145 (January 1977).

17. Nelson, G. and Oppen, D.C., “Fast decision procedures based on congruence closure,” *JACM* **27**(2) pp. 356-364 (April 1980).

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

19. Reps, T. and Yang, W., “The semantics of program slicing,” Technical Report 777, Department of Computer Sciences, University of Wisconsin—Madison (June 1988).

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

21. Rich, C., “Inspection methods in programming: clichés and plans,” A.I. Memo No. 1005, Artificial Intelligence Laboratory, M.I.T., Cambridge, MA (December 1987).

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

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

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

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

26. Yang, W., Horwitz, S., and Reps, T., “A program integration algorithm that accommodates semantics-preserving transformations,” pp. 133-143 in *Proceedings of the Fourth Symposium on Software Development Environments*, (Irvine, CA, December 3-5, 1990), ACM, New York, NY (1990).