Abstract

Programmers frequently face the problem of integrating several variants of a base program. Semantics-based program integration is a technique that attempts to create an integrated program that incorporates the changed *computations* of the variants as well as the *computations* of the base program that are preserved in all variants.

Horwitz, Prins, and Reps were the first to address the problem of semantics-based program integration. They presented an integration algorithm that creates the integrated program by merging certain program slices of the variants. Our study provides semantic foundations for their approach: we show that the integrated program produced by their algorithm includes all required *computations*.

We also develop a new program-integration algorithm with the same semantic properties. In addition, the new integration algorithm has two significant characteristics: (1) it is extendible in that it can incorporate any techniques for detecting program components with equivalent behaviors and (2) it can accommodate semantics-preserving transformations. The new integration algorithm improves on the integration algorithm of Horwitz, Prins, and Reps in that there are classes of program modifications for which their algorithm reports interference while the new integration algorithm produces satisfactory integrated programs.

One fundamental problem in program integration is to detect program components with equivalent behaviors. For this purpose. we devised the *Sequence-Congruence Algorithm*, which divides program components into equivalence classes by a partitioning scheme. We show that components in the same equivalence classes have equivalent behaviors.

The new integration algorithm is actually a family of algorithms, parameterized by the techniques used to detect equivalent components. Any equivalence-detection techniques can be used. Many techniques, such as constant propagation and invariant code movement, can be combined with the Sequence-Congruence Algorithm to detect larger classes of equivalent components.

The new integration algorithm is capable of accommodating semantics-preserving transformations. It allows different stages of computations to be modified independently in different variants as long as the same values are computed in each stage. Due to the use of a technique called *limited slicing*, these semantics-preserving transformations can be accommodated by the new integration algorithm.

Chapter 1

Introduction

1.1. Program Integration

Programmers frequently face the problem of merging several slightly different variants of a base program. A tool that analyzes the difference between these variants and the base program and creates an integrated version that incorporates all the changes made in the variants is, thus, desirable. This thesis concerns the design of such a tool to ease the task of integrating the variants and the base program.

It is easy to imagine the wide applications of such a program-integration tool. For instance, a group of people may develop a software project together. Without a program-integration tool, their work must be carefully divided so that each individual can work independently. If the software project cannot be decomposed in a modular, independent fashion, their work cannot proceed simultaneously due to the interdependences among each individual's share of work. An alternative is to let each person work with a separate copy of the source program and then merge the different versions into a common version *manually*, which is obviously a tedious and error-prone task. A program-integration tool is aimed at lessening this restriction; it allows parallel development of software in the presence of interdependences among each person's share of work. Another application area is concerned with the maintenance of software. As software evolves, new features are added and bugs are fixed, possibly by different groups of people whose work is hardly coordinated. A program-integration tool is needed in order to merge such modifications.

A program-integration tool should identify the changes made in the variants as well as the preserved part that is common to the base program and the variants. The changes and the preserved part are to be incorporated into the merged program when they do not "interfere" with one another (the notion of "interference" will be defined later). The integration tool should either produce a merged program that incorporates the changes and the preserved part or report that there is interference among the variants.

There already exist several integration tools such as the UNIX utility *diff3* that are based on the idea of merging program *text*. These text-based tools are not *safe* for merging programs because they do not guarantee how the execution behavior of the merged program produced by a successful integration relates to that of the variants and the base program. Since programmers are interested in the execution behavior of programs as well as the text of programs, the merged program produced by a text-based integration tool might well be far from reasonable when judged from the standpoint of execution behavior. For instance, consider the following three program fragments and the merged program fragment that might be produced by a text-based integration tool:

Program *Base*	Variant *A*	Variant *B*	Merged program *M*
$a := 1$	$a := 1000$	$a := 1$	$a := 1000$
$b := 2$	$b := 2$	$b := 2000$	$b := 2000$
$c := a + b$	$c := a + b$	$c := a + b$	$c := a + b$

Variant *A* changes the assignment to *a*; this change is included in the merged program *M*. Variant *B* changes the assignment to *b*; this change is also included in *M*. The assignment to *c* is untouched in both variants, and hence is included in *M*. However, the assignment statement to *c* in the merged program is meaningless since the value computed there, 3000, is not computed anywhere in *Base*, *A*, or *B*. Text-based tools identify *textual* changes of the variants and incorporate the textual changes into the merged program. Since changes in execution behavior are not necessarily indicated by changes in program text (in the above example, the assignment statement to *c* is not changed at all but different values are computed at this statement in the variants), the merged program that incorporates all the textual changes does not necessarily incorporate all the behavioral changes.

Berzins [Berzins86] addresses a part of the program-integration problem from the semantic perspective. Given two programs, his method attempts to find a merged program that is the least (semantic) extension that subsumes both versions, that is, a merged program that incorporates the whole behavior of the two versions. However, as software evolves, not only extensions but also modifications (such as bug-fixes) are made to the base program. Modifications are not addressed by his method.

Horwitz, Prins, and Reps were the first to address the problem of semantics-based program-integration [Horwitz88, Horwitz89]. They defined an integration algorithm (referred to hereafter as the HPR algorithm) that takes as input three programs *Base*, *A*, and *B*,[1] where *A* and *B* are two variants of *Base*. The integrated program is produced by (1) building graphs that represent *Base*, *A*, and *B*, (the graphs are called *program dependence graphs*, which will be defined in Chapter 2), (2) combining *program slices*[2] [Weiser84, Ottenstein84, Horwitz88] of the program dependence graphs of *Base*, *A*, and *B* to form a merged graph, (3) testing the merged graph for certain interference criteria, and (4) reconstituting a program from the merged graph.

Semantics-preserving transformations—transformations that change the way computations or stages of computations are performed without changing the values computed—have long been advocated as an alternative way to develop clear, efficient, reliable, and reusable software [Gerhart75, Darlington76, Burstall77, Huet78, Feather82, Partsch83]. Using this methodology, programmers may write clear and understandable programs without being concerned with the efficiency of the resulting programs. The transformation systems will automatically or semiautomatically transform the clear but inefficient programs into equivalent, efficient ones [Loveman77, Paige83]. The transformation systems help in verifying software in that once the transformation rules have been validated, verification of the programs can be performed on the clear and understandable versions rather than on the transformed versions, which are presumably more complicated and harder to verify. Because programs are written in a clear and understand-

[1]Both the HPR algorithm and the new integration algorithm developed in this thesis (the algorithm is presented in Chapter 5) can accommodate any number of variants; for the sake of exposition, we consider the common case of two variants.

[2]The slice of a program with respect to a component *c* consists of the set of program components that might affect either directly or transitively the values produced at *c*. Program slices can be easily extracted from the program dependence graph of a program.

able manner (probably at a more abstract level than the concrete implementation), software reusability becomes a more attainable goal [Cheatham84].

Experience from several experimental transformation systems shows that the transformational programming methodology, combined with recent advances in verification, syntax-directed editing systems [Reps84], and high level languages, is very promising for software development [Paige83]. It is, thus, desirable to give the integration algorithm the ability to accommodate semantics-preserving transformations so that program integration tools can fit in gracefully with modern software development environments.

The research work reported in this thesis is concerned with semantics-based program integration; it consists of two parts: The first part—Chapter 2—concerns the semantic criterion for program integration, and develops the semantic foundations for the HPR integration algorithm (see also Section 1.2). In the second part, starting in Chapter 3, we develop a new program-integration algorithm that also satisfies the semantic criterion. The most significant improvements of the new integration algorithm are (1) it can accommodate semantics-preserving transformations and (2) it is very flexible and extendible in that additional techniques for detecting program components with equivalent behaviors[3] can be easily incorporated in the new integration algorithm (see Section 1.3). The new integration algorithm improves on the integration algorithm of Horwitz, Prins, and Reps in that there are classes of program modifications for which their algorithm reports interference while the new integration algorithm produces satisfactory integrated programs.

The new integration algorithm uses a new data structure, called *program representation graphs*, to represent programs. Program representation graphs are defined in Chapter 3. One of the steps in the new program-integration algorithm is to detect program components that have equivalent execution behavior. Any algorithm that can detect components with equivalent behavior can be used in the new integration algorithm. In particular, we have developed the *Sequence-Congruence Algorithm* for this purpose, which is discussed in Chapter 4. The new program-integration algorithm is presented in Chapter 5. Chapter 6 shows that the new integration algorithm satisfies the semantic criterion for program integration. In Chapter 7, we demonstrate that, by using Sequence-Congruence Algorithm to detect components with equivalent behavior, the new integration algorithm is strictly better than the HPR integration algorithm. Chapter 8 concludes this thesis and discusses future work.

The remainder of this chapter surveys these results in more detail.

1.2. Semantic Foundations of the HPR Integration Algorithm

The first issue addressed by this thesis is that of providing the semantic foundations for the HPR integration algorithm. This is the subject of Chapter 2, which shows that the integrated program incorporates the changed behaviors of *A* and *B* as well as the preserved behaviors common to *Base*, *A*, and *B*.

The integrated program produced by the HPR algorithm is composed of program slices of the base and the variant programs. The slice of a program with respect to a component *c* consists of the set of program components that might affect either directly or transitively the values produced at *c*. For example, consider the following two programs:

---

[3]For the time being, we say that two components have equivalent behaviors if they produce the same sequence of values when the programs containing them run and terminate normally on an initial state. A precise definition is given in Chapter 4.

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

Program P sums the integers from 1 to 10 and adds the sum to the variable *result*; program Q is the slice of program P with respect to the assignment statement $x := x + 1$. Because the assignment statements $x := 1$ and $x := x + 1$, and the predicate $x < 11$ of the *while* loop can affect the values produced at the assignment $x := x + 1$, these three components are included in the slice Q; all other statements are discarded.

The semantic foundations for the HPR integration algorithm are established using two results about the semantic properties of program slices. First, we show that program slices capture portions of the program execution behavior in the following sense:

> Suppose program Q is a slice of program P. Then for any initial state σ on which P terminates normally,[4] Q also terminates normally on σ and P and Q compute the same sequence of values at each corresponding program component.

This property tells us that if the program terminates normally on an initial state then slices of the program also terminate normally on the same initial state. Second, we demonstrate that if all the constituent slices of the merged program terminate normally on an initial state σ, then the merged program also terminates normally on σ. Based on the semantic properties of program slices, we are able to prove the following semantic properties about the HPR integration algorithm:

Suppose the HPR integration algorithm successfully integrates two variants A and B with respect to the base program *Base* and produces an integrated program M. Then for any initial state σ on which *Base*, A, and B all terminate normally,

(1) M terminates normally on σ.

(2) For every variable x that is defined in the final state of A, if x's final values after executing *Base* and A are different, then x's final value in M is the same as in A (that is, M agrees with A on x).

(3) For every variable y that is defined in the final state of B, if y's final values after executing *Base* and B are different, then y's final value in M is the same as in B (that is, M agrees with B on y).

(4) For every variable z that is defined in the final state of *Base*, if z's final values after executing *Base*, A, and B are the same, then z's final value in M is the same as in all three (that is, M agrees with all three on z).

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

---

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

Note that properties (1)-(4) can be taken as the *semantic criterion* for program integration. Any program that satisfies (1)-(4) can be viewed as the integrated version of the two variants with respect to the base program. If no such program exists, the changes made in the variants interfere. However, this criterion is not decidable; consequently, any integration *algorithm* will fail on some examples for which a program satisfying (1)-(4) exists.

The semantic criterion for program integration leaves a great deal of freedom for constructing the integrated program, but would be plagued by the familiar undecidable problems of automated program derivation if the integration system is allowed to make up new statements for the integrated program. Therefore, we further require that the integrated program be composed of exactly the statements and control structures that appear as components of the two variants.

1.3. A New Program-Integration 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 when one or both variants change the *way* different stages of a computation are performed—without changing the values computed— and new code is added that uses the result of the changed computation.

This situation is illustrated in Figure 1-1, which shows two sets of programs, each set containing a base program and two variants. The HPR algorithm will report interference in both cases; however, there is actually no interference according to the semantic criterion of the previous section, and a satisfactory integrated program is shown in each case. In the first example, variant *A* adds an assignment to variable *vol* that uses the value of *area*; variant *B* renames variable *P* to *PI*, and moves the assignment "*rad* := 2" inside the conditional. In the second example, variant *A* adds new statements to compute *sumAV* and *prodAV* and changes the way *ratio* is computed (but the value of *ratio* computed remains unchanged); variant *B* changes the initial assignments to *sum* and *x* (but still computes the same final values for *sum* and *prod*) and adds a new statement to compute *percentage* from *ratio*.

In the two examples, one or both variants perform semantics-preserving transformations to different stages of the computation. These transformations change the program slices of some components but preserve the values computed at these components. By recognizing these semantics-preserving transformations, we can determine that these components in *Base*, *A*, and *B* actually compute the same values even though they have different program slices.

There are two reasons why the HPR integration algorithm reports interfence for the examples in Figure 1-1: The first is because the HPR algorithm *conservatively* determines whether two components have equivalent behavior (by comparing program slices). Certain components that actually have equivalent behavior might be pessimistically classified as having inequivalent behavior. The second is due to the way program fragments are extracted from *Base*, *A*, and *B* to form the merged program (see Chapter 2); the extracted program fragments may sometimes include unnecessary components. To create an integration algorithm that could handle examples like the ones in Figure 1-1, it was necessary to develop a number of new concepts and techniques. These are discussed below.

Program *Base*	Variant *A*	Variant *B*	Integrated Program

program
P := 3.14
rad := 2
if debug
   then rad := 4
fi
area := P * (rad**2)
end(area)
	program
	P := 3.14
	rad := 2
	if debug
	   then rad := 4
	fi
	area := P * (rad**2)
	$\boxed{height := 4}$
	$\boxed{vol := height*area}$
	end(area, $\boxed{vol}$)
		program
		$\boxed{PI := 3.14}$
		if debug
		   then rad := 4
		   $\boxed{\textbf{else}\ rad := 2}$
		fi
		$\boxed{area := PI * (rad**2)}$
		end(area)
			program
			PI := 3.14
			if debug
			   then rad := 4
			   else rad := 2
			fi
			area := PI * (rad**2)
			height := 4
			vol := height*area
			end(area,vol)

program
sum := 0
prod := 1
x := 1
while x < 11 **do**
   sum := sum + x
   prod := prod * x
   x := x + 1
od
ratio := sum / prod
end(ratio)
	program
	sum := 0
	prod := 1
	x := 1
	while x < 11 **do**
	   sum := sum + x
	   prod := prod * x
	   x := x + 1
	od
	$\boxed{sumAV := sum / 10}$
	$\boxed{prodAV := prod / 10}$
	$\boxed{ratio{:=}sumAV/prodAV}$
	end(ratio, $\boxed{sumAV}$,
	      $\boxed{prodAV}$)
		program
		$\boxed{sum := 1}$
		prod := 1
		$\boxed{x := 2}$
		while x < 11 **do**
		   sum := sum + x
		   prod := prod * x
		   x := x + 1
		od
		ratio := sum / prod
		$\boxed{percentage{:=}ratio*100}$
		end($\boxed{percentage}$)
			program
			sum := 1
			prod := 1
			x := 2
			while x < 11 **do**
			   sum := sum + x
			   prod := prod * x
			   x := x + 1
			od
			sumAV := sum / 10
			prodAV := prod / 10
			ratio:=sumAV/prodAV
			percentage:=ratio*100
			end(percentage,
			      sumAV,prodAV)

Figure 1-1. Two example integration problems that illustrate the limitation of the HPR algorithm with regard to semantics-preserving transformations. Modifications in variants *A* and *B* are enclosed in boxes. In both examples, the HPR algorithm will report interference even though there is no interference according to the semantic criterion for integration. The **end** statements will be explained in Chapter 2.

The Sequence-Congruence Algorithm

A fundamental problem in program integration is 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 *affected components*.) However, it is an undecidable problem to find the exact set of affected components. Any program-integration algorithm must conservatively determine the set of affected components; that is, the set of affected components determined by a program-integration algorithm must be a superset of the set of components that actually produce different values than the counterparts in the base program on some initial states.

The HPR algorithm compares program slices to find affected components. If a component *c*'s slice in the base program differs from its slice in a variant, then the way *c*'s values are computed differs in the base program and the variant, and thus the values themselves might differ. Therefore, any component whose slice in a variant differs from its slice in the base program is considered to be an affected component by the HPR algorithm. Comparing program slices is a *safe* method in that two components must have equivalent execution behaviors if the slices with respect to them are isomorphic and they are corresponding components under the isomorphism. (This is a corollary of the Slicing Theorem in Chapter 2.) However, comparing program slices is conservative in that two program components may have equivalent execution behavior even if the slices are different.

In this thesis work, we have developed alternative techniques for detecting affected components—techniques that are strictly more powerful than comparing slices. These techniques make use of an idea first introduced by Alpern, Wegman, and Zadeck [Alpern88], which is to first optimistically group possibly equivalent components in an initial partition and then find the coarsest partition compatible with the initial partition and dependences among program components.

However, the equivalence-testing algorithm of Alpern, Wegman, and Zadeck was not suitable for our purposes; the property that holds for components in the same final "equivalence class" is that components of a *single* program that are in the same final partition produce the same value at *certain moments* during program execution. There are two reasons why this is not the appropriate property for our purposes: (1) for integration, it is necessary to be able to identify equivalent components in *several* programs simultaneously; (2) for integration, we need a different notion of equivalence: components are equivalent only if they produce identical *sequences* of values.

Consequently, we developed a new algorithm, called the Sequence-Congruence Algorithm [Yang89], that uses the partitioning idea to find components that have equivalent behaviors (in the stronger sense indicated above). The affected components determined by the Sequence-Congruence Algorithm are a subset of the affected components determined using program slicing (but are still a safe approximation to the exact set of affected components), hence the technique is strictly more powerful than just comparing program slices.

The Sequence-Congruence Algorithm can be used to identify the semantic differences between several versions of a program [Yang89a, Horwitz90]. Although the Sequence-Congruence Algorithm is better than comparing program slices, it still produces a safe *approximation* to the exact set of affected components (*i.e.*, a *superset* of the exact set of affected components). For instance, the Sequence-Congruence Algorithm can determine that *Base*, *A*, and *B* compute the same value for the variable *area* in the first example of Figure 1-1. But, for the second example of Figure 1-1, the Sequence-Congruence Algorithm cannot determine that *Base*, *A*, and *B* compute the same value for the variable *ratio*. In general, the Sequence-Congruence Algorithm can recognize a limited set of semantics-preserving transformations, such as renaming variables, moving certain statements into/out of conditional statements, and adding copy statements (*i.e.*, statements of the form "$x := y$").

Limited Slices

The second reason why the HPR algorithm reports interference for the two sets of examples in Figure 1-1 is due to the way program fragments are extracted from *Base*, *A*, and *B* to form the merged program. In the HPR algorithm, whenever a component is assumed to be an affected component, the *whole* slice with respect to the component must be included in the merged graph since this is the only way to guarantee that the changed behaviors of the affected components are incorporated in the merged program. For instance, consider the first example of Figure 1-1. The slices with respect to the assignment to the variable *area* in *A* must be included in the merged program because this slice is part of the slice with respect to the assignment to *vol* (the assignment to *vol* is an affected component). Similarly, the slices with respect to the assignment to *area* in *B* must be included in the merged program because this slice has been changed in *B*. However, since the two slices with respect to the assignment to *area* in *A* and *B* are different, they cannot be combined together in the merged program. Hence, the HPR algorithm will report interference. In the second example of Figure 1-1, because the slices with respect to the assignment to *ratio* in *A* and *B* differ from the corresponding slice of *Base*, and because both slices must be included in the merged program, the HPR algorithm will report interference.

To address this limitation of the HPR algorithm, we propose a new integration algorithm that employs a new operation, called *limited slicing*, to extract program fragments from *Base*, *A*, and *B* to form a merged graph. Instead of including the whole slices with respect to the affected components, the new integration algorithm includes only the neighborhood of the affected components.

Because the new integration algorithm uses limited slicing to extract program fragments, it can accommodate semantics-preserving transformations. For instance, there are two stages in the first example in Figure 1-1: one is to compute *area*, the other is to compute *vol*. Variant *B* has applied a semantics-preserving transformation to the former stage while variant *A* added the latter stage. Limited slicing allows the new integration algorithm to extract the former stage from *B* and the latter stage from *A*. In the second example in Figure 1-1, there are three stages: one is to compute *sum* and *prod*, the second to compute *ratio*, and the third to compute *percentage*. Given an algorithm that can determine that the values of *sum* (and *prod*, respectively) are identical after the loops in *A* and *B*, the new integration algorithm will extract the first stage from variant *B* (*B* has performed semantics-preserving transformations to this stage). Similarly, the second stage is taken from *A*, and the third stage is taken from *B*. In this way, a satisfactory integrated program is constructed.

Properties of the New Integration Algorithm

The new integration algorithm is actually a *family* of algorithms, parameterized by the techniques used to detect components with equivalent behaviors. Any techniques, including the Sequence-Congruence Algorithm, that can detect components with equivalent behaviors can be employed. In Chapter 6, we demonstrate that the new integration algorithm also satisfies the semantic criterion given in Section 1.2. In Chapter 7, we show that if the Sequence-Congruence Algorithm is used to detect components with equivalent behaviors, the new integration algorithm improves on the HPR algorithm in the following sense:

(1) The new integration algorithm successfully integrates the two variants with respect to the base program whenever the HPR algorithm succeeds.

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

Although the new integration algorithm/Sequence-Congruence Algorithm combination improves on the HPR algorithm, it does not succeed in integrating all examples for which a reasonable integrated program exists. For instance, our techniques fail to handle the second example given in Figure 1-1: The Sequence-Congruence Algorithm is not powerful enough to detect that the final values of the variables *sum*, *prod*, and *ratio* are identical in *A*, *B*, and *Base*. As a result, certain components are pessimistically classified as affected components, which causes the new integration algorithm to report interference.

However, all is not lost; the new integration algorithm can use any of a number of algorithms to detect components with equivalent behaviors, not just the Sequence-Congruence Algorithm. Many techniques used in compiler optimization [Allen72, Loveman77, Aho86], 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 program components with equivalent behavior. Knowledge of semantics-preserving transformations that have been applied to a program or certain parts of the program, either detected by an equivalence-detection algorithm or obtained from the editor front-end of a program-transformation system, can be exploited to detect larger classes of program components with equivalent behaviors. If such techniques can be used to determine that the final values of *sum*, *prod*, and *ratio* are identical in *A*, *B*, and *Base*, the new integration algorithm does succeed in integrating the second example in Figure 1-1.

Chapter 2

Semantic Foundations

for the HPR Integration Algorithm

This chapter reviews the HPR integration algorithm [Horwitz88, Horwitz89] and provides the semantic foundations for that algorithm.[5] We prove that program slices capture a portion of a program's execution behavior. The results are formulated as the Slicing Theorem and the Termination Theorem. The semantic foundations of the HPR integration algorithm are stated as the Integration Theorem, which follows as a corollary of the slicing properties.

2.1. Review of the HPR Integration Algorithm

2.1.1. Program Dependence Graphs

While our goal is to design a semantics-based program-integration tool for a full-fledged programming language, in the study of both the HPR integration algorithm and the new integration algorithm, we restrict ourselves to a simplified programming language. This language possesses the essential features of the problem, and thus permits us to avoid inessential details while conducting our research.

The simplified programming language has the following characteristics: expressions contain only scalar variables and constants; statements are assignments, conditionals, while-loops, or **end** statements. An **end** statement, which can appear only as the last statement of a program, names zero or more of the variables used in the program. When execution terminates normally, only those variables will have values in the final state; intuitively, the variables named by the **end** statement are those whose final values are of interest to the programmer. Thus, a program is of the form

program
    stmtlist
end(*idlist*)

The language has no input statements, but a variable may be used before being assigned to, in which case the variable receives the value provided for that variable in the initial state.

Our discussion of the language's semantics is in terms of the following informal model of execution. We assume a standard operational semantics for sequential execution of the corresponding flowchart (control flow graph [Aho86]): at any moment there is a single locus of control together with a global execution state map-

[5] The material in this chapter has been previously published as a technical report [Reps88] and in a conference proceedings [Reps89]. From now on, we will omit all references to the two papers.

ping program variables to values; the execution of each assignment statement or predicate passes control to a single successor; the execution of each assignment statement changes the global execution state. An execution of the program on some initial state also yields a (possibly infinite) sequence of values for each predicate and assignment statement in the program; the i^{th} element in the sequence for program component e consists of the value computed when e is executed for the i^{th} time. The variables named in the **end** statement are those whose final values are of interest to the programmer; when execution terminates normally, the final state is defined on only those variables in the **end** statement.

The HPR integration algorithm operates on program dependence graphs. Different definitions of program dependence graphs have been given; they are all variations on a theme introduced in [Kuck72]. The following definition of program dependence graphs for our restricted language is taken from [Horwitz88, Horwitz89]. (From now on, PDGs refer specifically to the program dependence graphs defined in [Horwitz88, Horwitz89].)

The program dependence graph (PDGs) for a program P, denoted by G_P, is a multigraph which consists of vertices designating program components and edges designating control and data dependences among components. There is a vertex for each assignment statement and for each predicate in the program. In addition, PDGs include three other categories of vertices: there is a distinguished vertex called the *Entry vertex*; for each variable x for which there is a path in the standard control-flow graph for P [Aho86] on which x is used before being defined, there is a vertex called the *initial definition of x* (labeled "$x := InitialState(x)$"); and for each variable x named in P's end statement, there is a vertex called the *final use of x* (labeled "$FinalUse(x)$").

The source of a control dependence edge, which is labeled *true* or *false*, is always the *Entry* vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge from v_1 to v_2, then the component represented by v_2 must be executed before the program terminates normally; if the value does not match the label on the control dependence edge, the component v_2 may not be executed.

Methods for determining control dependence edges for programs with unrestricted flow of control are given in [Ferrante87, Cytron89]; however, for our restricted language, control dependence edges can be determined in a simpler fashion: The control dependence edges merely reflect the nesting structure of the program.

- There is a control dependence edge from *Entry* to a vertex v if v represents a component that is not nested within any control constructs; this control dependence edge is labeled *true*.
- There is a control dependence edge from a predicate vertex u to a vertex v if v represents a component nested immediately within the control construct whose predicate is represented by u. If u is a predicate for a *while* loop, this control dependence edge is labeled *true*. If u is a predicate for an *if* statement, this control dependence edge is labeled by the truth value of the branch in which v occurs.

There are two kinds of data dependence edges: flow and def-order dependence edges. A flow dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \longrightarrow_f v_2$, means that during execution, the value produced at the component v_1 might be used at the component v_2. A flow dependence edge is *loop-carried* by a loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if the value produced at the component v_1 reaches the component v_2 via a back-edge of the loop L in the standard control-flow graph and both v_1 and v_2 are enclosed in loop L; otherwise it is *loop-independent*, denoted by $v_1 \longrightarrow_{li} v_2$.

There is a def-order edge from vertex v_1 to vertex v_2 if (1) both v_1 and v_2 assign values to a common variable and there is a third vertex w that is flow-dependent on both vertices; (2) v_1 and v_2 are in the same branch

(a) **program** *Main*
  sum := 0
  x := 1
  while *x* < 11 **do**
    sum := *sum* + *x*
    x := *x* + 1
  od
  result := *result* + *sum*
end(*result*)

(b)

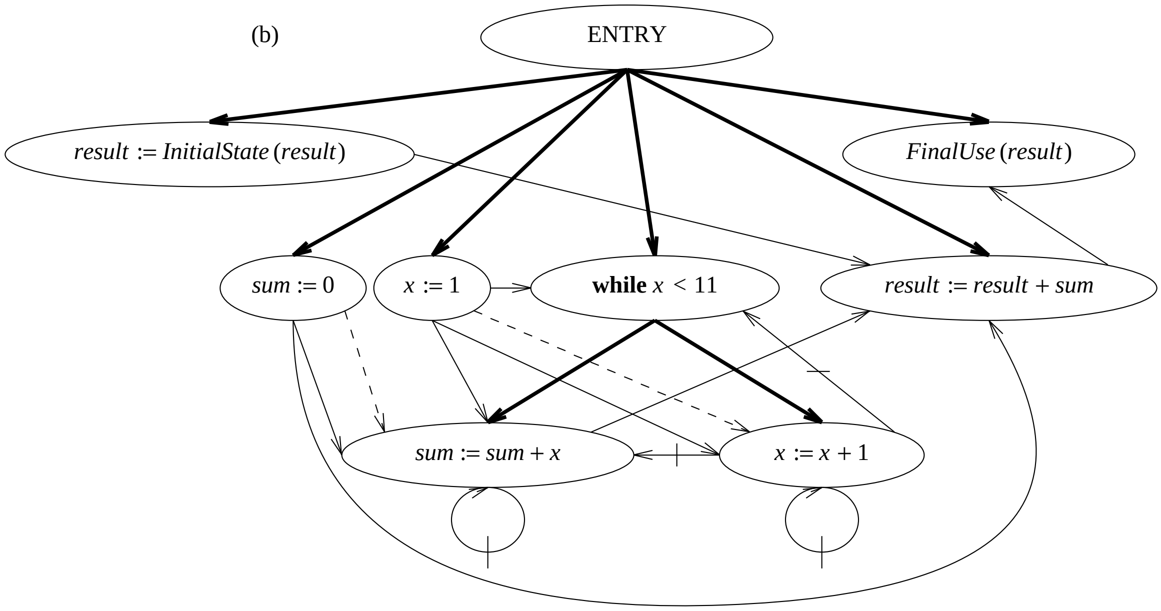


Figure 2-1. (a) is an example program. This program sums the integers 1 to 10 and adds the sum to the variable *result*. (b) is its program dependence graph. The boldface arrows represent control dependence edges, dashed arrows represent def-order dependence edges, solid arrows represent loop-independent flow dependence edges, and solid arrows with a hash mark represent loop-carried flow dependence edges.

of any conditional statement that encloses both vertices; and (3) the component v_1 occurs before the component v_2 in a pre-order traversal of the program's abstract syntax tree. A def-order edge from vertex v_1 to vertex v_2 witnessed by vertex w is denoted by $v_1 \longrightarrow_{do(w)} v_2$.

The data dependence edges can be computed using data flow analysis [Hecht77, Aho86]. For the restricted language considered in this thesis, the necessary computations can be defined in a syntax-directed manner (see [Horwitz87, Horwitz89] for details.)

Example. Figure 2-1(b) shows the program dependence graph of the example program in Figure 2-1(a). The boldface arrows represent control dependence edges, dashed arrows represent def-order dependence edges, solid arrows represent loop-independent flow dependence edges, and solid arrows with a hash mark represent loop-carried flow dependence edges.

2.1.2. Program Slices

A slice of a program with respect to a program component c consists of all the statements and predicates in the program that may affect the values computed at c during program execution [Weiser84]. Program slicing can be used to isolate individual computation threads within a program, which helps a programmer understand complicated code. Weiser gave an algorithm (formulated as a sequence of data-flow analysis problems) to extract program slices [Weiser84].

In [Ottenstein84], Ottenstein and Ottenstein developed a different method to extract program slices: first create a dependence graph that represents dependences among program components, then traverse the graph to find all vertices that can reach an initial set of vertices. In the HPR algorithm, program slices are extracted from the program dependence graphs in the same way as the method developed by Ottenstein and Ottenstein.

Program slices are used in the HPR algorithm to compute a safe approximation to the computation threads that have changed between a program and its variants, and to help determine whether two modifications to the base program interfere.

For a vertex s of a PDG G, the *slice* of G with respect to s, denoted by G/s, is a graph containing all vertices on which s has a transitive flow or control dependence (*i.e.*, all vertices that can reach s via a path of flow and control dependence edges): $V(G/s) = \{ w \mid w \in V(G) \wedge w \longrightarrow^*_{c,f} s \}$. We extend the definition to a set of vertices $S = \bigcup_i s_i$ as follows: $V(G/S) = V(G/(\bigcup_i s_i)) = \bigcup_i V(G/s_i)$. The edges in the graph G/S are essentially those in the subgraph of G induced by $V(G/S)$, with the exception that a def-order edge $v \longrightarrow_{do(u)} w$ is only included if, in addition to v and w, $V(G/S)$ also contains the vertex u that is directly flow dependent on the definitions at v and w. In terms of the three types of edges in a PDG we have:

$$\begin{aligned} E(G/S) = \; & \{ (v \longrightarrow_f w) \mid (v \longrightarrow_f w) \in E(G) \wedge v, w \in V(G/S) \} \\ & \cup \{ (v \longrightarrow_c w) \mid (v \longrightarrow_c w) \in E(G) \wedge v, w \in V(G/S) \} \\ & \cup \{ (v \longrightarrow_{do(u)} w) \mid (v \longrightarrow_{do(u)} w) \in E(G) \wedge u, v, w \in V(G/S) \}. \end{aligned}$$

Example. The slice of the example PDG shown in Figure 2-1(b) with respect to the assignment statement $x := x + 1$ is shown in Figure 2-2(b). This slice corresponds to the program in Figure 2-2(a).

We say a graph is a *feasible* program dependence graph if and only if it is isomorphic[6] to the program dependence graph of some program. Our first result concerns a syntactic property of program slices: we are able to show that for any program P and vertex set S, the slice G_P/S is a feasible PDG. The proof proceeds by showing that G_P/S is isomorphic to the PDG of the program whose components are the statements and predicates in $V(G_P/S)$ arranged in the same order as they occur in P.

---

[6]Two multigraphs G_1 and G_2 are isomorphic, denoted by $G_1 \approx G_2$, if and only if the following conditions are satisfied:

(1) There is a 1-to-1, onto mapping f from the vertex set of G_1 to the vertex set of G_2 and for every vertex v in G_1, v and $f(v)$ have the same text.

(2) There is a 1-to-1, onto mapping g from the edge set of G_1 to the edge set of G_2 and for every edge e in G_1, e and $g(e)$ are of the same type (*i.e.*, both are control dependence edges or both are flow dependence edges) and have the same label.

(3) For every edge $u_1 \longrightarrow v_1$ in G_1 and its image $u_2 \longrightarrow v_2$ in G_2, u_2 is $f(u_1)$ and v_2 is $f(v_1)$.

When $G_1 \approx G_2$ or when we are trying to prove $G_1 \approx G_2$, for brevity, we will say v and $f(v)$ are *the same* vertex and e and $g(e)$ are *the same* edge.

(a)

program *Main*
$x := 1$
while $x < 11$ **do**
$x := x + 1$
od
end

(b)

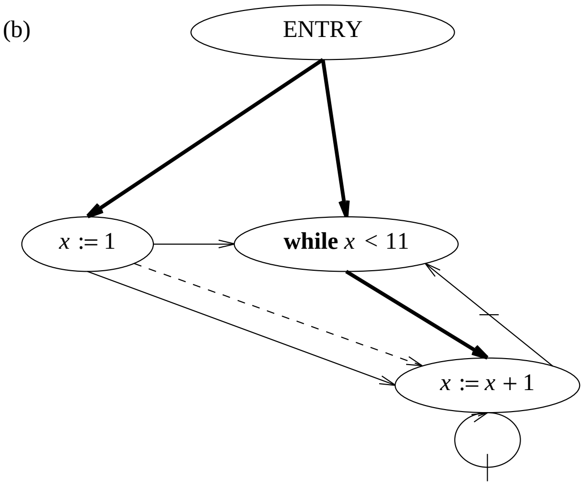


Figure 2-2. (b) is the slice of the program dependence graph in Figure 2-1 with respect to the assignment statement $x := x+1$. It corresponds to the program shown in (a).

Lemma. (Feasibility Lemma for Program Dependence Graph). *For any program P, if G_Q is a slice of G_P (with respect to some set of vertices), then G_Q is a feasible* PDG.

Proof. We construct a new program Q' from P and G_Q as follows: the components (statements and predicates) of Q' correspond to the vertices of G_Q; each component of Q' is subordinate to the same component that it is subordinate to in P; the components of Q' occur in the same relative order as they do in P. The variables that appear in the end statement of Q' are those variables whose final-use vertices are in G_Q. We use $G_{Q'}$ to denote the program dependence graph of Q'. We want to show that $G_Q \approx G_{Q'}$.

Because each component of Q' is subordinate to the same component that it is subordinate to in P, and because components of Q' occur in the same order as they occur in P, the control flow graph for program Q' is the projection of the control flow graph for program P onto the components of Q'. That is, if a and b are components of Q', the projection of any path from a to b in the control flow graph of P onto the set of components of Q' is a path in the control flow graph of Q'. Furthermore, every path from a to b in the control flow graph of Q' is the projection of some path from a to b (and possibly several such paths) in the control flow graph of P.

From the construction of Q', the only possible differences between the vertex sets of G_Q and $G_{Q'}$ is in their initial-definition vertices. By the definition of the vertex set of a slice, if there is an initial-definition vertex a for variable x in $V(G_Q)$, there must be a flow edge $a \longrightarrow_f b$ in $E(G_Q)$, where b is not an initial-def vertex. Since $a \longrightarrow_f b \in E(G_Q)$, it must be that $a \longrightarrow_f b \in E(G_P)$. This means that there is a path from the beginning of the control flow graph of P to b that is free of any definition to x. The projection of this path onto the components of Q' is a path in Q' from the beginning of the control flow graph of Q' to b and contains no definition to x. Consequently, $V(G_{Q'})$ must contain an initial-definition vertex for x, which corresponds to vertex a in $V(G_Q)$.

Arguing in the other direction, suppose there is an initial-definition vertex a for variable x in $V(G_{Q'})$ and a flow edge $a \longrightarrow_f b$ that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $b \in V(G_Q)$ but $a \longrightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \longrightarrow_f b$ cannot be in $E(G_P)$. Therefore, along each path from the *Entry* vertex to b in P there must be a redefinition of x. Along each such path p, let c_p be the last redefinition site. Since $c_p \longrightarrow_f b$ is in $E(G_P)$ and b is in $V(G_Q)$, $c_p \longrightarrow_f b$ is in $E(G_Q)$; the vertex c_p itself must be in $V(G_Q)$ and hence in Q'. Because every path from the *Entry* vertex to b in the control flow graph of Q' is a projection of a path p from the *Entry* vertex to b in the control flow graph of P, there must be a redefinition of x on each path from the *Entry* vertex to b in Q'. This means that $a \longrightarrow_f b$ cannot be in $E(Q')$, which is a contradiction, hence there does exist a flow edge $a \longrightarrow_f b$ in $E(G_P)$ where a is the initial-definition vertex for x in P. Because $b \in V(G_{Q'})$ and b itself is not an initial-definition vertex, by the construction of Q' it must be that $b \in V(G_Q)$. Consequently, by the definition of the vertex set of a slice, $a \in V(G_Q)$ and $a \longrightarrow_f b \in E(G_Q)$.

We have shown above that G_Q and $G_{Q'}$ have the same vertex sets, what remains to be shown is that G_Q and $G_{Q'}$ have the same edge sets.

Sub-proof 1. If the edge $a \longrightarrow b$ is in G_Q, then $a \longrightarrow b$ is in $G_{Q'}$.

1) By the definition of the edge set of a slice, if $a \longrightarrow_c b$ is a control edge in $E(G_Q)$, then $a \longrightarrow_c b$ is a control edge in $E(G_P)$, which means that b is subordinate to a in program P. Because a component in program Q' is subordinate to the same component that it is subordinate to in P, $a \longrightarrow_c b$ is in $E(G_{Q'})$ as well.

2) By the definition of the edge set of a slice, if $a \longrightarrow_f b$ is a flow edge in $E(G_Q)$, then $a \longrightarrow_f b$ is in $E(G_P)$, which means that there is a path in the control flow graph of P from a to b without any redefinition to the target variable of a. The projection of this path onto the components of Q' is a path in Q' that contains no redefinition to the target variable of a; thus, $a \longrightarrow_f b$ is in $E(G_{Q'})$.

3) By the definition of the edge set of a slice, if $a \longrightarrow_{do(c)} b$ is a def-order edge in $E(G_Q)$, then there are flow edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ in $E(G_Q)$. From the argument in (2), the edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ also occur in $E(G_{Q'})$. Because a occurs before b in P, a occurs before b in Q'. Therefore, $a \longrightarrow_{do(c)} b$ is in $E(G_{Q'})$.

Sub-proof 2. If the edge $a \longrightarrow b$ is in $G_{Q'}$, then $a \longrightarrow b$ is in G_Q.

1) If $a \longrightarrow_c b$ is a control edge in $E(G_{Q'})$, then b is subordinate to a in Q', hence b is subordinate to a in P. Therefore, $a \longrightarrow_c b$ is a control edge in $E(G_P)$ and, by the definition of the edge set of a slice, the control edge $a \longrightarrow_c b$ is a member of $E(G_Q)$.

2) Suppose $a \longrightarrow_f b$ is a flow edge that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $a, b \in V(G_Q)$ but $a \longrightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \longrightarrow_f b$ cannot be in $E(G_P)$. Therefore, along each path from a to b in P there must be a redefinition of the target variable of a.

   Along each such path p, let c_p be the last redefinition site. Since $c_p \longrightarrow_f b$ is in $E(G_P)$ and b is in $V(G_Q)$, $c_p \longrightarrow_f b$ is in $E(G_Q)$; the vertex c_p itself must be in $V(G_Q)$ and hence in Q'. Because every path from a to b in the control flow graph of Q' is a projection of a path p from a to b in the control flow graph of P, there must be a redefinition of the target variable of a on each path from a to b in Q'. This means that $a \longrightarrow_f b$ cannot be in $E(Q')$, which is a contradiction; therefore, $a \longrightarrow_f b$ is a flow edge in $E(G_Q)$.

3) If $a \longrightarrow_{do(c)} b$ is a def-order edge in $E(G_{Q'})$, then $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are in $E(G_{Q'})$. From the argument in (2), the edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are in $E(G_P)$ and $E(G_Q)$ as well. Because $a \longrightarrow_{do(c)} b$ is in $E(G_{Q'})$ a occurs before b in Q', hence a occurs before of b in P and therefore $a \longrightarrow_{do(c)} b$ is in $E(G_P)$. Now a,

b, and c are all in $V(G_Q)$; thus, by the definition of the edge set of a slice, $a \longrightarrow_{do(c)} b$ is in $E(G_Q)$.

We have shown that G_Q is isomorphic to $G_{Q'}$ and hence corresponds to program Q'. Therefore, a slice of a feasible PDG is always a feasible PDG. $\Box$

Since a slice of a PDG is a feasible PDG, "a slice of a program" is as meaningful as "a slice of a program dependence graph." We say program Q is a slice of program P with respect to a set of components S when G_Q is isomorphic to G_P / S, and write this as P / S.

2.1.3. The HPR Integration Algorithm

A basic assumption of the HPR integration algorithm (and also of the new integration algorithm in Chapter 5) is that the integrated program must be composed of only the statements and control structures that appear as components of the variants. In conjunction with this assumption, we assume that there is a unique-naming mechanism so that program components are identified consistently in all three versions. Each component that occurs in a program is assigned a tag so that components that have the same tag are considered to be "the same" component occurring in different versions of the programs. Components' tags can be maintained by the editor or they can be generated by an algorithm which, after analyzing the base and the variant programs, can consistently identify the components. The exact source of tags is irrelevant to the integration algorithms.

We say that two graphs G_1 and G_2 are "the same graph," denoted by $G_1 = G_2$, if they are isomorphic and corresponding components under the isomorphism have the same tag.

The HPR integration algorithm takes as input a base program *Base*, and two variants A and B. Whenever the changes made to *Base* to create A and B do not "interfere" (in the sense defined below), a merged program M is produced that exhibits the changed execution behavior of A and B with respect to *Base*, as well as the execution behavior preserved in all three versions. The integration algorithm consists of three steps. The first step determines slices that represent a safe approximation to the changed computation threads of A and B and the computation threads of *Base* preserved in both A and B; the second step combines these slices to form the merged graph G_M; the third step tests G_M for interference.

Step 1: Determining changed and preserved computation threads

If the slice of variant G_A with respect to vertex v differs from the slice of G_{Base} with respect to v, then G_A and G_{Base} may compute different values at v. In other words, vertex v is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of G_A with respect to G_{Base}, denoted by $AP_{A,\,Base}$, to be the subset of vertices of G_A whose slices in G_{Base} and G_A differ: $AP_{A,\,Base} = \{\, v \mid v \in V(G_A) \wedge (G_{Base} / v) \neq (G_A / v) \,\}$. We define $AP_{B,\,Base}$ similarly. It follows that the slices $G_A / AP_{A,\,Base}$ and $G_B / AP_{B,\,Base}$ capture the respective computation threads of A and B that differ from *Base*.

If the slice of G_{Base} with respect to vertex v is identical to the slices of G_A and G_B with respect to v, then all three programs compute the same sequence of values at v (this assertion is a corollary of the Slicing Theorem, which will be proved in Section 2.2). We define the *preserved points*, denoted by $PP_{Base,\,A,\,B}$, to be the subset of vertices of G_{Base} with identical slices in G_{Base}, G_A, and G_B: $PP_{Base,\,A,\,B} = \{\, v \mid v \in V(G_{Base}) \wedge (G_{Base} / v) = (G_A / v) = (G_B / v) \,\}$. Thus, the slice $G_{Base} / PP_{Base,\,A,\,B}$ captures the computation threads of *Base* that are preserved in both A and B.

Step 2: Forming the merged graph

The merged graph, G_M, is formed by unioning the three slices that represent the changed and preserved computation threads:

$G_M = (G_A / AP_{A,\ Base}) \cup (G_B / AP_{B,\ Base}) \cup (G_{Base} / PP_{Base,\ A,\ B})$.

Step 3: Testing for interference

There are two possible ways by which the graph G_M may fail to represent a satisfactory integrated program; both types of failure are referred to as "interference." The first interference criterion is based on a comparison of slices of G_A, G_B, and G_M. The slices $G_A / AP_{A,\ Base}$ and $G_B / AP_{B,\ Base}$ represent the changed computation threads of programs *A* and *B* with respect to *Base*. *A* and *B* interfere if G_M does not preserve these slices; that is, there is *no* interference of this kind if $G_M / AP_{A,\ Base} = G_A / AP_{A,\ Base}$ and $G_M / AP_{B,\ Base} = G_B / AP_{B,\ Base}$.

The final step of the HPR integration algorithm involves reconstituting a program from the merged graph. However, it is possible that there is no program whose PDG is isomorphic to the merged graph. This is the second kind of interference that may occur. (Determining whether a graph is a feasible program dependence graph has been shown to be NP-complete [Horwitz88a]. The crux of the problem is to order each predicate's control-successors. A backtracking algorithm has been written and proved correct [Ball90].)

If neither kind of interference occurs, one of the programs that corresponds to the graph G_M will be returned as the result of the integration operation.

2.2. The Slicing Theorem

We now turn to the relationship between the execution behavior of a program and the execution behavior of a slice of the program. Because of the way a program slice is derived from a program, it is not unreasonable to expect that the program and the slice exhibit similar execution behavior. However, because a diverging computation may be "sliced out," a program and a slice of the program do not necessarily exhibit identical execution behaviors; in particular, a slice may produce a result on some initial states for which the original program diverges. For example, the program shown below on the left always diverges, whereas the program on the right, obtained by slicing the left-hand-side program with respect to variable *x* at the program's **end** statement, always converges:

```
program Main
   x := 1
   while true do
      x := x + 1
   od
   x := 0
end(x)
```

```
program Main
   x := 0
end(x)
```

The main result of this section is the Slicing Theorem, which asserts that a slice captures a portion of the program's behavior in the sense that, for any initial state on which the program terminates normally, the program and the slice compute the same sequence of values at each component of the slice.

Theorem. (Slicing Theorem). *Let Q be a slice of program P with respect to a set of program components. If* σ *is a state on which P terminates normally, then for any state* σ' *that agrees with* σ *on all variables for which there are initial-definition vertices in* G_Q*: (1) Q terminates normally on* σ'*, (2) P and Q compute the same sequence of values at each component of Q, and (3) the final states agree on all variables for which* *there are final-use vertices in* G_Q.

(The second clause means that the sequence of values produced at each component of Q is identical to that produced at the corresponding component of P. The third clause of the theorem's conclusion is implied by the second clause; it is stated explicitly to emphasize what the theorem says about programs viewed as state-transformers.)

The proof of the Slicing Theorem relies on a lemma, the Subtree Slicing Lemma, which is stated and proven in Section 2.2.2.

2.2.1. Additional Terminology and Results

The abstract syntax of the language is defined as the terms of the types *id*, *exp*, *stmt*, *stmt_list*, and *program* constructed using the operators *Assign*, *While*, *IfThenElse*, *StmtList*, and *Program*. The five operators of the abstract syntax have the following definitions:

$$\begin{array}{ll} \mathit{Assign}: & \mathit{id} \times \mathit{exp} \rightarrow \mathit{stmt} \\ \mathit{While}: & \mathit{exp} \times \mathit{stmt\_list} \rightarrow \mathit{stmt} \\ \mathit{IfThenElse}: & \mathit{exp} \times \mathit{stmt\_list} \times \mathit{stmt\_list} \rightarrow \mathit{stmt} \\ \mathit{StmtList}: & \mathit{stmt}^{+} \rightarrow \mathit{stmt\_list} \\ \mathit{Program}: & \mathit{id} \times \mathit{stmt\_list} \times \mathit{id}^{*} \rightarrow \mathit{program} \end{array}$$

In operator *Program*, the initial *id* represents the program name, and the id^* component represents the variables named in the end statement.

We also introduce an overloaded constant, *Null*, which is used to represent null trees of type *id*, *exp*, *stmt*, and *stmt_list*:

$$\begin{array}{ll} \mathit{Null}: & \rightarrow \mathit{id} \\ \mathit{Null}: & \rightarrow \mathit{exp} \\ \mathit{Null}: & \rightarrow \mathit{stmt} \\ \mathit{Null}: & \rightarrow \mathit{stmt\_list} \end{array}$$

Null is introduced solely for technical reasons, and is never an element of a program's abstract syntax tree.

The subgraph induced by the *control* dependences of program dependence graph G_P forms a tree that is closely related to the abstract syntax tree for program P. The control dependence subtree is rooted at the *Entry* vertex of G_P, which corresponds to the Program node at the root of P's abstract syntax tree. Each predicate vertex v of G_P corresponds to an interior node of the abstract syntax tree; the node is a While node or an IfThenElse node depending on whether v is labeled with *while* or *if*, respectively. Each assignment vertex of G_P corresponds to an Assign node of the abstract syntax tree.

The control dependence subtree rooted at a vertex v of G_P corresponds to the subtree of the abstract syntax tree that is rooted at the control construct that corresponds to v. Because of this correspondence, for brevity we use phrases, such as "the flow edges whose source is in subtree T," which are, strictly speaking, not correct when T is a subtree of the abstract syntax tree. What "T" refers to is the subgraph induced by T in G_P's control dependence subgraph.

Imported and exported variables

Our goal is to show that a slice of a program exhibits a portion of the program's behavior in the sense that the slice always terminates normally on an initial state whenever the program terminates normally on a sufficiently similar initial state and when they both terminate normally, they compute the same sequence of values at each corresponding program component; in particular, they are equivalent state transformers with respect to certain variables when they both terminate normally. In making this argument, it is necessary to discuss the state-transforming properties of individual statements, or equivalently, subtrees of the program's abstract syntax tree. The state-transforming properties of such subtrees are characterized in terms of the subtrees' *imported* and *exported* variables.

Definition. The *outgoing flow edges* of a subtree *T* consist of all the loop-independent flow edges whose source is in *T* but whose target is not in *T*, together with all the loop-carried flow edges for which the source is in *T* and the edge is carried by a loop that encloses *T*. Note that the target of an outgoing loop-carried flow edge may or may not be in *T*. The variables *exported* from a subtree *T* are the variables defined at the source of an outgoing flow edge.

Definition. The *incoming flow edges* of a subtree *T* consist of all the loop-independent flow edges whose target is in *T* but whose source is not in *T*, together with all the loop-carried flow edges for which the target is in *T* and the edge is carried by a loop that encloses *T*. Note that the source of an incoming loop-carried flow edge may or may not be in *T*. The *incoming def-order edges* of a subtree *T* consist of all the def-order edges whose target is in *T* but whose source is not in *T*. The variables *imported* by a subtree *T* are the variables defined at the source of an incoming flow edge or at the source of an incoming def-order edge.

By definition, the imported variables of a program *P* consist of all the variables for which there are *Initial-State* vertices in *P*'s PDG and the exported variables of a program *P* consist of all the variables for which there are *FinalUse* vertices in *P*'s PDG.

Corresponding subtrees

Let *Q* be a slice of *P* with respect to a set of program components. There is a natural correspondence between subtrees in *P* and subtrees in *Q*, defined as follows:

Definition. Let *Q* be a slice of *P* with respect to some set of program components. For each subtree *U* of *Q* with root *u*, *U corresponds* to the subtree of *P* whose root is *u*. For each subtree *T* of *P*, if the root *t* of *T* occurs in *Q*, *T corresponds* to the subtree of *Q* rooted at *t*; if *t* does not occur in *Q*, *T corresponds* to the tree *Null*.

Thus, for each subtree of *Q*, there is always a corresponding subtree of *P*, and *vice versa*, although for a subtree of *P* the corresponding subtree of *Q* may be the tree *Null*.

Note that the "corresponds to" relation respects the hierarchical structure of programs: children of roots of corresponding subtrees are the roots of corresponding subtrees.

2.2.2. The Subtree Slicing Lemma

The Subtree Slicing Lemma characterizes the relationship between a subtree and a slice of the subtree in terms of the slice's imported and exported variables. The Lemma asserts that, for certain initial states, corresponding subtrees of a program and a slice of the program compute the same sequence of values at common program components.

Lemma. (Subtree Slicing Lemma). *Let Q be a slice of program P with respect to a set of program components. Let T be a subtree of program P and U be the corresponding subtree of Q. If* σ *is a state on which T terminates normally, then for any state* σ' *that agrees with* σ *on U's imported variables (as defined in the context given by Q): (1) U terminates normally on* σ'*, (2) T and U compute the same sequence of values at each program component of U, and (3) the final states agree on U's exported variables (as defined in the context* *given by Q).*

Proof. The proof is by structural induction; it splits into five cases. Throughout the proof, we use Imp_U and Exp_U to denote the imported and exported variables of U; we use σ_1 and σ_1' to denote states that agree on U's imported variables, Imp_U. We use σ_i to denote a sequence of states in the execution of T initiated on σ_1, and we use σ_i' to denote the corresponding sequence of states in the execution of U initiated on σ_1'.

Case 1. The operator at the root of T is the Assign operator. Because T is a single assignment statement, either U is the tree *Null* or $U = T$. If U is *Null*, then $Imp_U = Exp_U = \emptyset$. Hence U always terminates normally and the final states agree on Exp_U (since Exp_U is empty).

Now suppose $U = T$ and that U is of the form "$x := exp$" where exp is an expression that uses variables $\{y_j\}$. The set Imp_U is either $\{y_j\}$ or $\{y_j\} \cup \{x\}$. (Imp_U is $\{y_j\} \cup \{x\}$ when U is the target of a def-order edge.) Since the value of exp is a function of $\{y_j\}$ and $\{y_j\} \subseteq Imp_U$, evaluating exp in both σ_1 and σ_1' yields the same value because they agree on Imp_U. Exp_U is either $\emptyset$ or $\{x\}$. For any combination of these possibilities, σ_2 and σ_2' agree on x, and hence they agree on Exp_U.

Case 2. The operator at the root of T is the While operator. If the vertex corresponding to T's *exp* component is not in U, then U is the tree *Null*. If U is *Null*, then $Imp_U = Exp_U = \emptyset$. Hence U always terminates normally and the final states agree on Exp_U.

We use Imp_{exp} to denote the imported variables of U's *exp* component. Imp_{stmt_list} and Exp_{stmt_list} denote the imported and exported variables of U's *stmt_list* component, respectively. We use σ_i and σ_i' to denote the execution states before executing the i^{th} iterations of the loops of T and U starting from two states, σ_1 and σ_1', that agree on Imp_U.

Suppose the vertex corresponding to T's *exp* component is in U. Since T terminates normally we may assume the execution of T terminates normally after the j^{th} iteration, for some j. It is sufficient to show that (1) U also terminates normally after the j^{th} iteration, (2) in each iteration, T and U compute the same sequence of values at each program point of U, and (3) the final states, σ_{j+1} and σ_{j+1}' agree on Exp_U. Because for a loop $Exp_U \subseteq Imp_U$,[7] it suffices to show that if σ_i and σ_i' agree on Imp_U then either T and U terminate normally in the states σ_i and σ_i', respectively, or the i^{th} iterations compute the same sequence of values at each program point of U and result in the states σ_{i+1} and σ_{i+1}' that agree on Imp_U.

First, we show that $Imp_U = Imp_{exp} \cup Imp_{stmt\_list}$. It is clear that we could have written this with $\subseteq$, noting that Imp_{stmt_list} can include a variable x that is used at the target t of a loop-carried flow dependence edge where the dependence is carried by U. However, there then has to exist an incoming loop-independent flow edge to t, which implies that $x \in Imp_U$.

Let σ_i and σ_i' be states that agree on Imp_U. Therefore they agree on Imp_{exp}. Evaluating the predicate (the *exp* component of U) in σ_i and σ_i' yields the same value. Hence, T and U compute the same (sequence of) values at the control predicate of the loop in the i^{th} iteration. If the predicate evaluates to *false*, then both executions terminate normally in the states σ_i and σ_i', which agree on Exp_U.

Now suppose the predicate evaluates to *true*. Let σ_i and σ_i' be states that agree on Imp_U; therefore they agree on Imp_{stmt_list}. Now T_{stmt_list} and U_{stmt_list} are corresponding subtrees. Since T terminates normally on

[7]If $x \in Exp_U$, then U contains an assignment statement a that assigns to x with an outgoing flow edge $a \longrightarrow_f b$. Because the loop may execute zero times, the assignment statement that assigns to x must be the target of a def-order edge $\ldots \longrightarrow_{do(b)} a$, hence $x \in Imp_U$.

σ_i, T_{stmt_list} also terminates normally on σ_i. By the induction hypothesis, (1) U_{stmt_list} terminates normally on σ_i', (2) during the i^{th} iteration T_{stmt_list} and U_{stmt_list} compute the same sequence of values at each program point of U_{stmt_list}, and (3) the final states, σ_{i+1} and σ_{i+1}', agree on Exp_{stmt_list}.

If σ_{i+1} and σ_{i+1}' do not also agree on Imp_U, then let $x \in Imp_U$ be a variable on which they disagree (so $x \notin Exp_{stmt\_list}$). Now, by assumption, σ_i and σ_i' agree on Imp_U; therefore, at least one of the two executions of T_{stmt_list} and U_{stmt_list}, respectively, executed an assignment statement a that assigned a value to x and reached the end of the *stmt_list*. There are two cases to consider:

(1) One possibility is that $x \in Imp_U$ because x is used in a predicate or statement b that is the target of an incoming flow edge $\ldots \longrightarrow_f b$ in U. If this were the case, then there must be a loop-carried flow edge $a \longrightarrow_{lc(T)} b$ or $a \longrightarrow_{lc(U)} b$, depending on whether T_{stmt_list} or U_{stmt_list} executed a. However, in either case, a is in U because b is in U; therefore, a is in U_{stmt_list} and $x \in Exp_{stmt\_list}$, which contradicts our previous assumption.

(2) The other possibility is that $x \in Imp_U$ because the U_{stmt_list} has an incoming def-order edge $\ldots \longrightarrow_{do(c)} d$. However, this implies that there is an outgoing flow edge $a \longrightarrow_f c$ from T_{stmt_list} or U_{stmt_list}, depending on whether T_{stmt_list} or U_{stmt_list} executed a. In either case, a must be in U because c is in U; therefore, a is in U_{stmt_list} and $x \in Exp_{stmt\_list}$, which contradicts our previous assumption.

We conclude that σ_{i+1} and σ_{i+1}' agree on Imp_U. Therefore U terminates normally after the j^{th} iteration, during each iteration T and U compute the same sequence of values at each program point of U, and σ_{j+1} and σ_{j+1}' agree on Exp_U.

Case 3. The operator at the root of T is the IfThenElse operator. If the vertex corresponding to T's *exp* component is not in U, then U is the tree *Null*. If U is *Null*, $Imp_U = Exp_U = \emptyset$. Therefore, U always terminates normally and the final states agree on Exp_U.

Suppose the vertex corresponding to T's *exp* component is in U. Evaluating the predicate (the *exp* component of U) in σ_1 and σ_1' yields the same value. Therefore, T and U compute the same (sequence of) values at the control predicate of the IfThenElse statement.

Without loss of generality, assume that the predicate evaluates to *true*. We use T_{true}, T_{false}, U_{true}, and U_{false} to denote the respective branches of T and U. We use Imp_{true}, Imp_{false}, Exp_{true}, and Exp_{false} to denote the imported and exported variables of U_{true} and U_{false}, respectively.

When execution is initiated in state σ_1, T terminates normally in σ_2; consequently T_{true} also terminates normally in σ_2. Since σ_1 and σ_1' agree on Imp_U and $Imp_{true} \subseteq Imp_U$, σ_1 and σ_1' agree on Imp_{true}. Because T_{true} and U_{true} are corresponding subtrees, the induction hypothesis tells us that, when execution is initiated in state σ_1', (1) U_{true} terminates normally in state σ_2' (hence U terminates normally in σ_2'), (2) T_{true} and U_{true} compute the same sequence of values at each program point of U_{true} (hence T and U compute the same sequence of values at each program point of U), and (3) σ_2 and σ_2' agree on Exp_{true}.

Note that $Exp_U = Exp_{true} \cup Exp_{false}$ so what remains to be shown is that σ_2 and σ_2' agree on Exp_{false}. If σ_2 and σ_2' do not also agree on Exp_{false}, then let $x \in Exp_{false}$ be a variable on which they disagree (so $x \notin Exp_{true}$). Because $x \in Exp_{false}$, there is an assignment statement a in the false branch of U that assigns to x and is the source of an outgoing flow edge from that branch (say $a \longrightarrow_f b$).

We must consider whether it is possible that $x \notin Imp_U$. By assumption, $x \notin Exp_{true}$. Consider an execution path p, from the beginning of the program to the beginning of statement U, that does not include the back-edges of any loops. Let c be the last assignment statement that assigns to x along p, or, if no such statement exists, let c be the initial-definition vertex for x. Because we can extend path p to first follow the true branch of U and then continue from the join point of U via the path by which a reaches b, we deduce that there is a dependence $c \longrightarrow_f b$. By construction, vertex c occurs before a, hence $c \longrightarrow_{do(b)} a$. We conclude that $x \in Imp_U$.

Since $x \in Imp_U$, σ_1 and σ_1' agree on x. Because σ_2 and σ_2' disagree on x, at least one of the two executions of the true branches of T and U, respectively, executed an assignment statement d that assigned a value to x and reached the end of the true branch. But this implies the existence of a flow edge $d \longrightarrow_f b$ in either T or U, depending on whether T_{true} or U_{true} executed the assignment to x. In either case, the flow edge $d \longrightarrow_f b$ is in Q since b is in Q, and hence d is in U_{true}. Therefore, $x \in Exp_{true}$, which contradicts a previous assumption. We conclude that σ_2 and σ_2' agree on Exp_{false}. This, together with the fact that σ_2 and σ_2' agree on Exp_{true}, means that they agree on Exp_U.

Case 4. The operator at the root of T is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ denote the immediate subtrees of T in the order as they occur in program P. Note that all loop-independent flow edges and def-order edges from one subtree to another go from left to right; that is, if there is a loop-independent flow edge or a def-order edge from a vertex in a subtree T_i to a vertex in a different subtree T_j then $i < j$. Let $U_1, U_2, \cdots, U_n$ denote the immediate subtrees of U in the order as they occur in program Q. Each T_i corresponds to some subtree $U_{\pi(i)}$ and *vice versa*, where the mapping π is a permutation over the interval $1, \cdots, n$. (Note that some of the U_is may be the tree *Null*. Since *Null* trees do not actually appear in program Q, we may think of the *Null* trees as being placed in arbitrary (but fixed) positions in the sequence $U_1, U_2, \cdots, U_n$.) Let π^{-1} denote the inverse permutation of π.

We use σ_i and $\sigma_{\pi(i)}'$ to denote the execution states before executing T_i and $U_{\pi(i)}$, respectively; we use Imp_i and Exp_i to denote the imported and exported variables, respectively, of $U_{\pi(i)}$. Since U is a slice of T, the imported (or exported) variables of $U_{\pi(i)}$ are a subset of the imported (or exported, respectively) variables of T_i.

The proof of this case is done in two steps. We first show, by induction over i, that for all i, $1 \le i \le n$, if σ_1 and σ_1' agree on Imp_U and T terminates normally on σ_1, then (1) σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i, (2) T_i and $U_{\pi(i)}$ compute the same sequence of values at each corresponding program point, and (3) σ_{i+1} and $\sigma_{\pi(i)+1}'$ agree on Exp_i. Secondly, we show that the final states after executing T and U, σ_{n+1} and σ_{n+1}', agree on the exported variables of U.

Note that, by the induction hypothesis of the structural induction, if σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i then T_i and $U_{\pi(i)}$ either both diverge or both terminate normally and compute the same sequence of values at each corresponding program point and σ_{i+1} and $\sigma_{\pi(i)+1}'$ agree on Exp_i. Thus, we will concentrate on proving that σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i, for all i, $1 \le i \le n$,

Base case. $i = 1$. First we show that σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1. (Note that Imp_1 is the set of the imported variables of $U_{\pi(1)}$.) Let x be any variable in Imp_1. It suffices to show that there is no assignment statement that assigns to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(1)-1}$. If there are assignment statements that assign to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(1)-1}$, then choose the largest k in $1, 2, \cdots, \pi(1)-1$ such that U_k contains an assignment statement that assigns to x. Since x is an imported variable of $U_{\pi(1)}$, $U_{\pi(1)}$ has an incoming loop-independent flow edge or an incoming def-order edge whose source is in U_k. Since U is a slice of T, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_1. Therefore, $\pi^{-1}(k) < 1$, which is impossible because π is a permutation over the interval $1, \cdots, n$. We conclude that σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1.

Furthermore, because there are no statements in the initial subsequence $U_1, U_2, \cdots, U_{\pi(1)-1}$ that assign to any variables in Imp_1, $Imp_1 \subseteq Imp_U$.

Because σ_1 and σ_1' agree on Imp_U and $Imp_1 \subseteq Imp_U$, σ_1 and σ_1' agree on Imp_1. Because σ_1 and σ_1' agree on Imp_1 and σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1, σ_1 and $\sigma_{\pi(1)}'$ agree on Imp_1. By the induction hypothesis of the structural induction, T_1 and $U_{\pi(1)}$ compute the same sequence of values at each corresponding program point on σ_1 and $\sigma_{\pi(1)}'$, respectively, and σ_2 and $\sigma_{\pi(1)+1}'$ agree on Exp_1.

Induction step. The induction hypothesis is: if σ_1 and σ_1' agree on Imp_U and T terminates normally on σ_1, then, for $1 \le j < i$, (1) σ_j and $\sigma_{\pi(j)}'$ agree on Imp_j, (2) T_j and $U_{\pi(j)}$ compute the same sequence of values at each corresponding program point, and (3) σ_{j+1} and $\sigma_{\pi(j)+1}'$ agree on Exp_j. Thus, if $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ are arbitrary states that agree on Imp_U and T terminates normally on $\hat{\sigma}_1$, we need to show that (1) $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on Imp_i, (2) T_i and $U_{\pi(i)}$ compute the same sequence of values at each corresponding program point, and (3) $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i)+1}'$ agree on Exp_i.

First we show that $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on Imp_i. (Note that Imp_i is the set of the imported variables of $U_{\pi(i)}$.) Let x be any variable in Imp_i. It suffices to show that $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on x. There are now two cases to consider:

(1) Suppose there is no assignment statement that assigns to x in the initial subsequence $T_1, T_2, \cdots, T_{i-1}$. We want to show that there is no assignment statment that assigns to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$. If there are assignment statments that assign to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$, choose U_k from the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$ such that there is an assignment statement a that assigns to x in U_k and there is a loop-independent flow edge or a def-order edge $a \longrightarrow \ldots$ from a vertex in U_k to a vertex in $U_{\pi(i)}$. Since U is a slice of T, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_i. Therefore, $\pi^{-1}(k) < i$. Because U_k has an assignment statement that assigns to x, $T_{\pi^{-1}(k)}$ has a corresponding assignment statement that assigns to x, which contradicts the previous assumption that there is no assignment statement that assigns to x in the initial subsequence $T_1, T_2, \cdots, T_{i-1}$. We conclude that there is no assignment statement that assigns to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$.

Because $x \in Imp_i$ and there is no assignment statement that assigns to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$, $x \in Imp_U$. Since there is no assignment statement that assigns to x in the initial subsequence $T_1, T_2, \cdots, T_{i-1}$, $\hat{\sigma}_1$ and $\hat{\sigma}_i$ agree on x. Similarly, since there is no assignment statement that assigns to x in the initial subsequence $U_1, U_2, \cdots, U_{\pi(i)-1}$, $\hat{\sigma}_1'$ and $\hat{\sigma}_{\pi(i)}'$ agree on x. Since $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ agree on Imp_U, $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ agree on x. Thus, $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on x.

(2) Suppose there are assignment statements that assign to x in the initial subsequence $T_1, T_2, \cdots, T_{i-1}$. Let m be the largest number in $1, 2, \cdots, i-1$ such that T_m contains an assignment statement that assigns to x. Because x is an imported variable of $U_{\pi(i)}$ it is an imported variable of T_i; hence, T_i has an incoming loop-independent flow edge or an incoming def-order edge whose source is in T_m. Since U is a slice of T and x is an imported variable of $U_{\pi(i)}$, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $U_{\pi(m)}$ to a vertex in $U_{\pi(i)}$ and hence $\pi(m) < \pi(i)$. Note that $x \in Exp_m$ because $U_{\pi(m)}$ has an outgoing loop-independent flow edge whose source is an assignment statement that assigns to x.

Since $m < i$, by the induction hypothesis, $\hat{\sigma}_m$ and $\hat{\sigma}_{\pi(m)}'$ agree on Imp_m. Because $\hat{\sigma}_m$ and $\hat{\sigma}_{\pi(m)}'$ agree on Imp_m and T_m and $U_{\pi(m)}$ are corresponding subtrees, by the induction hypothesis of the structural induction, the execution states after executing T_m and $U_{\pi(m)}$, $\hat{\sigma}_{m+1}$ and $\hat{\sigma}_{\pi(m)+1}'$, agree on Exp_m; in particular, they agree on x.

By the choice of m, we know that there is no assignment statement that assigns to x in the subsequence $T_{m+1}, T_{m+2}, \cdots, T_{i-1}$. We want to show that there is no assignment statement that assigns to x in the subsequence $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i)-1}$. If there is an assignment statement that assigns to x in the subsequence $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i)-1}$, let U_k be one of $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i)-1}$ that contains an assignment statement that assigns to x such that there is a loop-independent flow edge or a def-order edge from a vertex in U_k to a vertex in $U_{\pi(i)}$ and a def-order edge from a vertex in $U_{\pi(m)}$ to a vertex in U_k. Since U is a slice of T, there is a corresponding flow dependence edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_i and a def-order edge from T_m to a vertex in $T_{\pi^{-1}(k)}$. Therefore, $m < \pi^{-1}(k) < i$. But this is impossible since there is no assignment statement that assigns to x in the subsequence $T_{m+1}, T_{m+2}, \cdots, T_{i-1}$. We conclude that there is no assignment statement that assigns to x in the subsequence $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i)-1}$.

Note that $\hat{\sigma}_{m+1}$ and $\hat{\sigma}_i$ agree on x because there is no assignment statement that assigns to x in the subsequence $T_{m+1}, T_{m+2}, \cdots, T_{i-1}$. Similarly, $\hat{\sigma}_{\pi(m)+1}'$ and $\hat{\sigma}_{\pi(i)}'$ agree on x because there is no assignment statement that assigns to x in the subsequence $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i)-1}$. Because $\hat{\sigma}_{m+1}$ and $\hat{\sigma}_{\pi(m)+1}'$ agree on x, $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on x.

We conclude that $\hat{\sigma}_i$ and $\hat{\sigma}_{\pi(i)}'$ agree on Imp_i. By the induction hypothesis of the structural induction, T_i and $U_{\pi(i)}$ compute the same sequence of values at each corresponding program point and and $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i)+1}'$ agree on Exp_i. This completes the induction.

The second step is to show that the final states after executing T and U, σ_{n+1} and σ_{n+1}', agree on Exp_U. Let x be any variable in Exp_U. Since $x \in Exp_U$ (and hence $x \in Exp_T$), there are assignment statements that assign to x in T. Let m be the largest number in $1, 2, \cdots, n$ such that T_m contains an assignment statement that assigns to x. Since U is a slice of T and $x \in Exp_U$, there are also assignment statements that assign to x in $U_{\pi(m)}$. Since T_m and $U_{\pi(m)}$ are corresponding subtrees, by the previous arguments, the states after executing T_m and $U_{\pi(m)}$, σ_{m+1} and $\sigma_{\pi(m)+1}'$, agree on the exported variables of $U_{\pi(m)}$; in particular, σ_{m+1} and $\sigma_{\pi(m)+1}'$ agree on x.

Furthermore, we claim that there cannot be any assignment statements that assign to x in $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_n$. If there are assignment statements that assign to x in $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_n$, let k be the largest number in $\pi(m)+1, \pi(m)+2, \cdots, n$ such that U_k contains a statment that assigns to x. Because $x \in Exp_U$, there is an outgoing flow edge from a vertex in U_k to a vertex outside of U. Because $x \in Exp_T$ and m is that largest number in $1, \cdots, n$ such that T_m contains a statement that assigns to x, there is an outgoing flow edge from a vertex in T_m to a vertex outside of T. Since U is a slice of T, there is a corresponding outgoing flow edge from a vertex in $U_{\pi(m)}$ to a vertex outside of U. Because $\pi(m) < k$ and because there are flow edge from vertices of U_k and of $U_{\pi(m)}$ to vertices outside of U, there is a def-order edge from a vertex of $U_{\pi(m)}$ to a vertex of U_k. Since U is a slice of T, there is a corresponding def-order edge from a vertex of T_m to a vertex of $T_{\pi^{-1}(k)}$. Therefore, $m < \pi^{-1}(k)$ and $T_{\pi^{-1}(k)}$ contains a statement that assigns to x. But this is impossible due to the choice of m. We conclude that there cannot be any assignment statements that assign to x in $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_n$.

Because there are no assignment statements that assign to x in $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_n$, the states $\sigma_{\pi(m)+1}'$ and σ_{n+1}' agree on x. We have already shown that the states σ_{m+1} and σ_{n+1} agree on x and σ_{m+1} and $\sigma_{\pi(m)+1}'$ agree on x. Thus, the states σ_{n+1} and σ_{n+1}' agree on x.

We conclude that *T* and *U* compute the same sequence of values at each corresponding program point and the final states agrees on *U*'s exported variables (as defined in the context given by *Q*).

Case 5. The operator at the root of *T* is the Program operator. Because $Imp_U = Imp_{stmt\_list}$ and $Exp_U = Exp_{stmt\_list}$ where *stmt_list* denotes the *stmt_list* component of *U*, we conclude from the induction hypothesis that *U* terminates normally on σ', *T* and *U* compute the same sequence of values at each program point of *U*, and σ_2 and σ_2' agree on Exp_U. □

2.2.3. The Slicing Theorem

The Slicing Theorem follows as a corollary of the Subtree Slicing Lemma; it is simply the Subtree Slicing Lemma specialized to the case when subtree *T* is the entire program *P*.

Theorem. (Slicing Theorem). *Let Q be a slice of program P with respect to a set of program components. If* σ *is a state on which P terminates normally, then for any state* σ' *that agrees with* σ *on all variables for which there are initial-definition vertices in* G_Q*: (1) Q terminates normally on* σ'*, (2) P and Q compute the same sequence of values at each program component of Q, and (3) the final states agree on all variables for which there are final-use vertices in* G_Q*.*

The Equivalence Theorem for program dependence graphs, proved in [Horwitz87a, Horwitz88b], asserts that two programs with isomorphic program dependence graphs are strongly equivalent. Because a program *Q* can be viewed as a slice of any program *P* whose program dependence graph is isomorphic to *Q*'s program dependence graph, the Equivalence Theorem follows as a corollary of the Slicing Theorem. In fact, what falls out of the Slicing Theorem is a stronger result than the result proved in [Horwitz87a, Horwitz88b], stated below:

Theorem. (Strong Form of the Equivalence Theorem). *Suppose that P and Q are programs for which* $G_P \approx G_Q$*. If* σ *is a state on which P terminates normally, then for any state* σ' *that agrees with* σ *on all variables for which there are initial-definition vertices in* G_P*: (1) Q terminates normally on* σ'*, (2) P and Q compute the same sequence of values at each corresponding program component, and (3) the final states agree on all variables for which there are final-use vertices in* G_P*.*

2.3. The Termination Theorem

The Slicing Theorem tells us that if a program terminates normally on an initial state then the program's slices also terminate normally on a similar initial state. The Termination Theorem looks at this relationship from the opposite point of view: it tells us that if a program is decomposed into two slices, the termination of both slices on an initial state implies the termination of the program on that initial state. It is straightforward to generalize the theorem to the case when the program is decomposed into more than two slices.

2.3.1. The Subtree Termination Lemma

As in the Slicing Theorem, the proof of the Termination Theorem relies on a lemma about subtrees. The Subtree Termination Lemma states that if a program is decomposed into two slices, a subtree of the program will terminate normally on a state when the corresponding subtrees of the two slices terminate normally on some similar states.

Lemma. (Subtree Termination Lemma). *Let P be a program. Suppose X and Y are sets of vertices such that* $G_P = G_P / X \cup G_P / Y$*. Let T be a subtree of program P and U and V be the corresponding subtrees of* P / X *and* P / Y*, respectively. Suppose* σ_U *is a state on which U terminates normally, and* σ_V *is a state on* *which V terminates normally. Then for any state σ, where σ agrees with σ_U on U's imported variables and σ agrees with σ_V on V's imported variables, T terminates normally on σ.*

Proof. By the Equivalence Theorem of PDGs, all programs with isomorphic program dependence graphs are strongly equivalent. We choose P / X and P / Y to be the versions of the slices whose statements are in the same order as in P.

The proof is by structural induction; it again splits into five cases as in the proof of the Subtree Slicing Theorem. Since the argument is quite similar, we show only the second case − the case for a While loop. Readers who want to see the details are referred to the technical report [Reps88].

Case 2. The operator at the root of T is the While operator. Since $G_P = G_P / X \cup G_P / Y$, if U is a *Null* tree, then $T = V$. Similarly, if V is a *Null* tree, then $T = U$. Without loss of generality, suppose $T = U$. Because σ and σ_U agree on U's imported variables and U terminates normally on σ_U, by the Equivalence Theorem, T terminates normally on σ.

Now suppose neither U nor V is a *Null* tree. Since U terminates normally on σ_U, we may assume that the execution of U terminates normally after the j^{th} iteration, for some j. We want to prove that T and V terminate normally on σ and σ_V, respectively, after exactly j iterations.

Because for a loop $Exp_U \subseteq Imp_U$, it suffices to show that if σ and σ_U agree on U's imported variables and σ and σ_V agree on V's imported variables, then either T, U, and V terminate normally in the states σ, σ_U, and σ_V, respectively, or T, U, and V successfully finish one iteration and the execution states that result after one iteration of the loops (σ', σ_U', and σ_V', respectively) are ones such that σ' and σ_U' agree on U's imported variables and σ' and σ_V' agree on V's imported variables.

We use T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} to denote the *stmt_list* components of T, U, and V, respectively. Note that T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} are corresponding subtrees of P, P / X, and P / Y, respectively.

Because σ and σ_U agree on U's imported variables, evaluating the control predicates in σ and σ_U yields the same value. Because σ and σ_V agree on V's imported variables, evaluating the control predicates in σ and σ_V yields the same value. If the control predicate evaluates to *false*, then T, U, and V terminate normally in the states σ, σ_U, and σ_V, respectively.

Now suppose the control predicate evaluates to *true*. Because σ and σ_U agree on U's imported variables and the imported variables of U_{stmt_list} are a subset of U's imported variables, σ and σ_U agree on the imported variables of U_{stmt_list}. Similarly, σ and σ_V agree on the imported variables of V_{stmt_list}. Note that T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} are corresponding subtrees of P, P / X, and P / Y, respectively. Because U_{stmt_list} and V_{stmt_list} terminate normally on σ_U and σ_V, respectively, by the induction hypothesis, T_{stmt_list} terminates normally on σ. Therefore, T, U, and V successfully finish one iteration.

Let σ', σ_U', and σ_V' denote the execution states of T, U, and V after one iteration of the loop, respectively. By the Subtree Slicing Lemma, σ' and σ_U' agree on U's exported variables and σ' and σ_V' agree on V's exported variables. By the same argument as in the proof of the Subtree Slicing Lemma, Case 2, σ' and σ_U' agree on U's imported variables. Similarly, σ' and σ_V' agree on V's imported variables. We conclude that T, U, and V terminate normally on σ, σ_U, and σ_V, respectively, after the j^{th} iteration. □

2.3.2. The Termination Theorem

The Termination Theorem follows as a corollary of the Subtree Termination Lemma; it is simply the Subtree Termination Lemma specialized to the case when subtree T is the entire program P.

Theorem. (Termination Theorem). *Let P be a program. Suppose X and Y are sets of vertices such that* $G_P = G_P / X \cup G_P / Y$. *If* P / X *and* P / Y *terminate normally on a state* σ, *then P terminates normally on* σ *as well.*

Note that the Termination Theorem and clause (1) of the Slicing Theorem are complementary: clause (1) of the Slicing Theorem asserts that if a program terminates normally then each slice also terminates normally; the Termination Theorem asserts that when a program can be decomposed into two slices, if each slice terminates normally then the program terminates normally. We can then apply clause (2) of the Slicing Theorem to conclude that the two slices (collectively) compute the same sequence of values as the entire program.

The following Corollary generalizes the Termination Theorem to the case when the program is decomposed into three slices. It is used in the proof of the Integration Theorem in the next section; the integrated program that is the subject of the proof is formed by taking the union of three slices.

Corollary. *Let P be a program. Suppose X, Y, and Z are sets of vertices such that* $G_P = G_P / X \cup G_P / Y \cup G_P / Z$. *If* P / X, P / Y, *and* P / Z *terminate normally on a state* σ, *then P terminates normally on* σ *as well.*

Proof. From the definition of program slices, it is obvious that $G_P / X \cup G_P / Y = G_P / (X \cup Y)$. Let $P / (X \cup Y)$ denote a program whose program dependence graph is isomorphic to $G_P / (X \cup Y)$. Since P / X and P / Y terminate normally on σ, by the Termination Theorem, $P / (X \cup Y)$ terminates normally on σ. Similarly, $G_P = G_P / X \cup G_P / Y \cup G_P / Z = G_P / (X \cup Y) \cup G_P / Z$. Since $P / (X \cup Y)$ and P / Z terminate normally on σ, P terminates normally on σ. □

2.4. The Integration Theorem

The HPR integration algorithm is justified by the Integration Theorem, which characterizes the execution behavior of the merged program in terms of the behaviors of the base program and the two variants. Because the merged program is composed of three slices, the Integration Theorem is a consequence of the Slicing Theorem and the Termination Theorem.

Theorem. (Integration Theorem for the HPR Algorithm). *If A and B are two variants of Base for which integration succeeds (and produces 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 that produces a different sequence of values than the corresponding component* v_{Base} *in Base during the executions of A and Base, there is a corresponding program component v in M that produces the same sequence of values as* v_A *during the executions of M and A.*

(3) *For any program component* v_B *in B that produces a different sequence of values than the corresponding component* v_{Base} *in Base during the executions of B and Base, there is a corresponding program component v in M that produces the same sequence of values as* v_B *during the executions of M and B.*

(4) *For any corresponding program components* v_{Base} *in Base,* v_A *in A, and* v_B *in B that produce the same sequence of values during the respective executions of Base, A, and B, there is a corresponding program component v in M that produces the same sequence of values as* v_{Base} *during the executions of M and Base.*

Restated less formally, *M* preserves the changed behaviors of both *A* and *B* (with respect to *Base*) as well as the unchanged behavior of all three.

The merged graph G_M is formed by unioning the three slices $G_A / AP_{A,\ Base}$, $G_B / AP_{B,\ Base}$, and $G_{Base} / PP_{Base,\ A,\ B}$. Because the premise of the theorem is that integration succeeds, we know that $G_M / AP_{A,\ Base} = G_A / AP_{A,\ Base}$ and $G_M / AP_{B,\ Base} = G_B / AP_{B,\ Base}$. One detail that must be shown is that, in testing G_M for interference, it is unnecessary to test whether $G_{Base} / PP_{Base,\ A,\ B} = G_M / PP_{Base,\ A,\ B}$.

This matter is addressed by the Preserved Behavior Lemma, which shows that, regardless of whether or not the integration algorithm detects interference, the slice $G_{Base} / PP_{Base,\ A,\ B}$ is always preserved in G_M.

Lemma. *If* $w \in AP_{A,\ Base}$, *then* $w \notin V(G_{Base} / PP_{Base,\ A,\ B})$.

Proof. Using the symbol $\subseteq$ to denote "is-a-slice-of", from the definition of program slicing we have:

$$G_{Base} / PP_{Base,\ A,\ B} = G_{Base} / \{ v \in V(G_{Base}) \mid (G_{Base} / v) = (G_A / v) = (G_B / v) \}$$

$$\subseteq G_{Base} / \{ v \in V(G_{Base}) \mid (G_{Base} / v) = (G_A / v) \}$$

$$= \bigcup_{v \in V(G_{Base}) \mid (G_{Base} / v) = (G_A / v)} G_{Base} / v$$

But if $w \in V(G_{Base} / v)$ for some v, then $G_{Base} / w \subseteq G_{Base} / v$; because $w \in AP_{A,\ Base}$, $G_{Base} / w \neq G_A / w$, and hence $G_{Base} / v \neq G_A / v$. Therefore, $w \notin V(G_{Base} / PP_{Base,\ A,\ B})$. □

Lemma. (Preserved Behavior Lemma).
Let $G_M = (G_A / AP_{A,\ Base}) \cup (G_B / AP_{B,\ Base}) \cup (G_{Base} / PP_{Base,\ A,\ B})$.
Then $G_{Base} / PP_{Base,\ A,\ B} = G_M / PP_{Base,\ A,\ B}$.

Proof. Let $PRE = G_{Base} / PP_{Base,\ A,\ B}$ and $PRE' = G_M / PP_{Base,\ A,\ B}$. Suppose $PRE \neq PRE'$. Because G_M is created by unioning *PRE* with $G_A / AP_{A,\ Base}$ and $G_B / AP_{B,\ Base}$, and the slices that generate *PRE* and PRE' are both taken with respect to the same set, $PP_{Base,\ A,\ B}$, it must be that $PRE \subset PRE'$.

Thus, there are three cases to consider: either PRE' contains an additional vertex, an additional control or flow edge (in the latter case either loop independent or loop carried), or an additional def-order edge.

Case 1. PRE' contains an additional vertex. Because the slices that generate *PRE* and PRE' are both taken with respect to the set, $PP_{Base,\ A,\ B}$, PRE' can only contain an additional vertex v if there is an additional control or flow edge $v \longrightarrow_{c,\ f} w$ whose target w is a component of both *PRE* and PRE'. Thus, this case reduces to the one that follows.

Case 2. PRE' contains an additional control or flow edge. The slice operation is a backward traversal of a dependence graph's edges; because the slices that generate *PRE* and PRE' are both taken with respect to the same set, namely $PP_{Base,\ A,\ B}$, if PRE' were to contain control or flow edges not in *PRE*, then there is at least one such edge whose target vertex occurs in both *PRE* and PRE'. That is, there is at least one edge $e \colon v \longrightarrow w$, where $e \in E(PRE')$, $v,\ w \in V(PRE')$, $w \in V(PRE)$, and $v \notin V(PRE)$.

Because G_M is created by a graph union, e must occur in $E(G_A / AP_{A,\ Base})$, $E(G_B / AP_{B,\ Base})$, or both. Without loss of generality, assume that $e \in E(G_A / AP_{A,\ Base})$, so that $e \in E(G_A)$.

The slice operation is a backward traversal of a dependence graph's edges, so $e \notin E(PRE)$ and $w \in V(PRE)$ imply $e \notin E(G_{Base})$. Taking this together with the previous observation that $e \in E(G_A)$, we conclude, from the definition of $AP_{A,\ Base}$, that $w \in AP_{A,\ Base}$.

This yields a contradiction as follows: because $w \in AP_{A,\ Base}$, by the previous lemma we conclude that $w \notin V(PRE)$.

Case 3. *PRE′* contains an additional def-order edge. Suppose $E(PRE')$ contains a def-order edge $e: v \longrightarrow_{do(u)} w$ that does not occur in $E(PRE)$. By the definition of the edge set of a slice, there must exist flow edges $v \longrightarrow_f u$ and $w \longrightarrow_f u$ in $E(PRE')$; by case (2), these edges must occur in both $E(PRE)$ and $E(PRE')$ (implying that $u, v, w \in V(PRE), V(PRE')$).

Because G_M was created by a graph union, e must occur in $E(G_A / AP_{A,\ Base})$, $E(G_B / AP_{B,\ Base})$, or both. Without loss of generality, assume that $e \in E(G_A / AP_{A,\ Base})$, so that $e \in E(G_A)$.

The slice operation is a backward traversal of a dependence graph's edges, so $e \notin E(PRE)$ and $u \in V(PRE)$ imply $e \notin E(G_{Base})$; by the definition of $AP_{A,\ Base}$, we conclude that $u \in AP_{A,\ Base}$.

This yields a contradiction analogous to the one that arose in case (2): because $u \in AP_{A,\ Base}$, by the previous lemma we conclude that $u \notin V(PRE)$.

We conclude that *PRE* and *PRE′* cannot differ; that is, even if variants *A* and *B* interfere with respect to base program *Base*, the slice $G_{Base} / PP_{Base,\ A,\ B}$ is preserved in G_M. □

The base program, the two variants, and the merged program share common slices; thus, the next matter to address is the relationship between the execution behaviors of two programs when they share a common slice. An immediate consequence of the Slicing Theorem is that two programs that share a slice compute the same sequence of values at corresponding components of the slice.

Corollary. (Slicing Corollary). *Let P and Q be two programs that share a slice with respect to a set of program components S (i.e.* $(P / S) = (Q / S)$*). Then, for any initial state* σ *on which both P and Q terminate normally, P and Q compute the same sequence of values at each component in S.*

Using the Slicing Corollary, the definition of the merged graph G_M, and the Preserved Behavior Lemma, we can prove the Integration Theorem for the HPR algorithm.

Proof of the Integration Theorem. We use $A / AP_{A,\ Base}$, $B / AP_{B,\ Base}$, and $Base / PP_{Base,\ A,\ B}$ to denote programs whose program dependence graphs are $G_A / AP_{A,\ Base}$, $G_B / AP_{B,\ Base}$, and $G_{Base} / PP_{Base,\ A,\ B}$, respectively.

Since the integration succeeds, $G_A / AP_{A,\ Base} = G_M / AP_{A,\ Base}$ and $G_B / AP_{B,\ Base} = G_M / AP_{B,\ Base}$. By the Preserved Behavior Lemma, $G_{Base} / PP_{Base,\ A,\ B} = G_M / PP_{Base,\ A,\ B}$. Therefore, $G_M = G_A / AP_{A,\ Base} \cup G_B / AP_{B,\ Base} \cup G_{Base} / PP_{Base,\ A,\ B} = G_M / AP_{A,\ Base} \cup G_M / AP_{B,\ Base} \cup G_M / PP_{Base,\ A,\ B}$.

Since *A* terminates normally on σ, by the Slicing Theorem, $A / AP_{A,\ Base}$ terminates normally on σ as well. Similarly, $B / AP_{B,\ Base}$ and $Base / PP_{Base,\ A,\ B}$ terminate normally on σ as well. Note that $A / AP_{A,\ Base}$ $B / AP_{B,\ Base}$, and $Base / PP_{Base,\ A,\ B}$ are programs whose program dependence graphs are $G_M / AP_{A,\ Base}$, $G_M / AP_{B,\ Base}$, and $G_M / PP_{Base,\ A,\ B}$, respectively. Since $A / AP_{A,\ Base}$ $B / AP_{B,\ Base}$, and $Base / PP_{Base,\ A,\ B}$ terminate normally on σ, by the Corollary of the Termination Theorem, *M* terminates normally on σ.

Let v_A be a program component in *A* that produces a difference sequence of values than the corresponding component v_{Base} in *Base* during the executions of *A* and *Base*. By the Slicing Theorem, $G_{Base} / v_{Base} \neq G_A / v_A$. Therefore, $v_A \in AP_{A,\ Base}$. Since $v_A \in AP_{A,\ Base}$ and $G_A / AP_{A,\ Base} = G_M / AP_{A,\ Base}$, there is a corresponding program component v in *M* such that $G_A / v_A = G_M / v$. By the Slicing Corollary, v and v_A produce the same sequence of values during the executions of *M* and *A*. This proves clause (2). (Clause (3) holds by the same argument.)

Let v_{Base} in *Base*, v_A in *A*, and v_B in *B* be corresponding program components that produce the same sequence of values during the respective executions of *Base*, *A*, and *B*. If $v_A \in AP_{A,\,Base}$, since $G_A \,/\, AP_{A,\,Base} = G_M \,/\, AP_{A,\,Base}$, there is a corresponding program component v in M such that $G_A \,/\, v_A = G_M \,/\, v$. By the Slicing Corollary, v and v_A produce the same sequence of values during the executions of M and A. Similarly, if $v_B \in AP_{B,\,Base}$, since $G_B \,/\, AP_{B,\,Base} = G_M \,/\, AP_{B,\,Base}$, there is a corresponding program component v in M such that $G_B \,/\, v_B = G_M \,/\, v$. By the Slicing Corollary, v and v_B produce the same sequence of values during the executions of M and B. If $v_A \notin AP_{A,\,Base}$ and $v_B \notin AP_{B,\,Base}$, then $v_{Base} \in PP_{Base,\,A,\,B}$. Because $G_{Base} \,/\, PP_{Base,\,A,\,B} = G_M \,/\, PP_{Base,\,A,\,B}$. there is a corresponding program component v in M such that $G_{Base} \,/\, v_{Base} = G_M \,/\, v$. By the Slicing Corollary, v and v_{Base} produce the same sequence of values during the executions of M and *Base*. In any case, v, v_A, v_B, and v_{Base} produce the same sequence of values during the respective executions of *M*, *A*, *B*, and *Base*. □

It is easy to see that, from the Integration Theorem, the HPR integration algorithm satisfies the semantic criterion of program integration discussed in Chapter 1. For example, if there is a variable x whose final value after executing A on σ differs from its final value after executing *Base* on σ, then (1) there is a final-use vertex for variable x in A, and (2) the (sequences of) values produced at the final-use vertices for x in M and in A are identical. Thus, x's final value after executing M on σ is equal to the final value of x after executing A on σ.

Chapter 3

Program Representation Graphs

Having provided the semantic foundations for the HPR integration algorithm, we will start developing the new program-integration algorithm. In this chapter, we define the data structure used in the new integration algorithm, which is called the program representation graph (PRG). Program representation graphs are similar to the program dependence graphs (PDGs) used in the HPR algorithm: in both PRGs and PDGs, vertices designate program components and edges designate dependences among the components. However, the def-order edges of PDGs are replaced by a new kind of pseudo-assignment vertex in PRGs. We show that PRGs are an adequate representation for programs in the sense that there is a version of the Equivalence Theorem for PRGs.

3.1. Program Representation Graphs

Program representation graphs combine features of static-single-assignment forms [Rosen88, Alpern88, Shapiro69] and program dependence graphs [Kuck81, Ferrante87], both of which have been widely studied in vectorizing and optimizing compilers [Ottenstein78, Allen83, Padua86, Baxter89].

Shapiro and Saint [Shapiro69] first introduced pseudo assignments by inserting trivial assignments at appropriate places in the program so that exactly one assignment to a variable *x*, either an assignment from the original program or a pseudo assignment, can reach a programmer-specified use of *x* in the program. For instance, consider the following example program fragments:

```
<T1>   x := 1
       y := 10
       if p then
<T2>      x := 2
          y := 20
       fi
<T5>   y := x + 3
```

```
<T1>   x := 1
       y := 10
       if p then
<T2>      x := 2
          y := 20
       fi
<T3>   φ_if: x := x
<T4>   φ_if: y := y
<T5>   y := x + 3
```

In the source program (on the left), both assignments to *x* at T1 and T2 can reach the use of *x* at T5; after the insertion of the pseudo assignment "ϕ_{if}: $x := x$" at T3 (on the right), only the pseudo assignment to *x* can reach the use of *x* at T5. The pseudo-assignment statements assign the value of a variable to itself; the importance of pseudo assignments is that they summarize the reaching definitions [Kennedy78, Aho86] of a variable at appropriate places in the programs. Data-flow information is more compactly represented by pseudo assignments.

In the static-single-assignment form of programs, pseudo assignments contain special ϕ functions. Using such ϕ functions explicitly leads to simpler formulation of various algorithms, such as algorithms for constant propagation [Wegman85] and code motion [Cytron86]. The value graph of Alpern, Wegman, and Zadeck [Alpern88] is built from the static-assignment-form to represent the symbolic execution of the program.

PRGs employ somewhat fewer pseudo-assignment statements than static-single assignment forms—a pseudo-assignment statement for a variable *x* is inserted at a point of the program only if *x* is *live* at that point. For instance, in the above example the ϕ_{if} statement for variable *y* at T4 will *not* be included in a PRG because *y* is not live at T4. Due to the exclusion of dead pseudo-assignment statements, there is a path from every ϕ vertex to a non-ϕ vertex in PRGs.

Program representation graphs contain *InitialState*, *FinalUse*, and *Entry* vertices together with vertices representing assignments and predicates (*i.e.*, exactly the same vertices as those in PDGs as defined in Chapter 2). In addition, program representation graphs include ϕ vertices for live variables, and edges representing control and flow dependences among the vertices. The control and flow dependence edges of program representation graphs are similar to those used in program dependence graphs defined in the previous chapter, except that these edges are defined in terms of the augmented control flow graph of a program rather than the original program (see below).

The program representation graph of a program *P*, denoted by R_P, is constructed in two steps. First an augmented control flow graph is built and then the program representation graph is constructed from the augmented control flow graph.

Step 1:

The control flow graph[8] [Aho86, Allen70] of program *P* is augmented by adding *InitialState*, *FinalUse*, ϕ_{if}, ϕ_{enter}, and ϕ_{exit} vertices, as follows:

(1) A vertex labeled "$x := InitialState\,(x)$" is added at the beginning of the control flow graph for each variable *x* that may be used before being defined in the program. If there are many *InitialState* vertices for a program, their relative order is not important as long as they come immediately after the *Entry* vertex.

(2) A vertex labeled "$FinalUse\,(x)$" is added at the end of the control flow graph for each variable *x* that appears in the *end* statement of the program. If there are many *FinalUse* vertices for a program, their relative order is not important as long as they come immediately before the *Exit* vertex.

(3) For every variable *x* that is defined within an *if* statement, and that may be used before being redefined after the *if* statement, a vertex labeled "ϕ_{if}: $x := x$" is added immediately after the *if* statement. If there are many ϕ_{if} vertices for an *if* statement, their relative order is not important as long as they come immediately after the *if* statement.

(4) For every variable *x* that is defined inside a loop, and that may be used before being redefined inside the loop or may be used before being redefined after the loop, a vertex labeled "ϕ_{enter}: $x := x$" is added immediately before the predicate of the loop. If there are many ϕ_{enter} vertices for a loop, their relative order is not important as long as they come immediately before the loop predicate. After the insertion of ϕ_{enter} vertices, the first ϕ_{enter} vertex of a loop becomes the entry point of the loop.

[8] In our control flow graphs, vertices represent assignment statements and predicates; in addition, there are two additional vertices, *Entry* and *Exit*, which represent the beginning and the end of the program.

(5) For every variable x that is defined inside a loop, and that may be used before being redefined after the loop, a vertex labeled "ϕ_{exit}: $x := x$" is added immediately after the loop. If there are many ϕ_{exit} vertices for a loop, their relative order is not important as long as they come immediately after the loop.

Note that ϕ_{enter} vertices are placed inside of loops, whereas ϕ_{exit} vertices are placed outside of loops.

Step 2:

Next, the program representation graph is constructed from the augmented control flow graph. Vertices of the program representation graph are those in the augmented control flow graph (except the *Exit* vertex). Edges represent control and flow dependences among program components.

Control and flow dependence edges in PRGs are similar to those in PDGs, except that they are defined in terms of the augmented control flow graphs:

- There is a control dependence edge from *Entry* to a vertex v if v occurs on every path from *Entry* to *Exit* in the augmented control flow graph. This control dependence edge is labeled *true*.
- There is a control dependence edge from a predicate vertex u to a vertex v if, in the augmented control flow graph, v occurs on every path from u to *Exit* along one branch out of u but not the other. This control dependence edge is labeled by the truth value of the branch in which v always occurs.
- There is a flow dependence edge $u \longrightarrow_f v$ if there is a variable x that is assigned a value at u and used at v, and there is a path from u to v in the augmented control flow graph along which no other (ϕ or non-ϕ) assignments to x occur. Note that there is a flow dependence edge incident on a non-ϕ vertex v for each use of a variable at v. For instance, if variable x is used twice at v, say v contains the expression $x + x$, there are two flow dependence edges incident on v whose sources are the last (ϕ or non-ϕ) assignment to x.

Note that there is a control dependence edge from a *while* predicate to itself.[9] This is because the predicate itself is on every path from the predicate, via the *true* branch of the predicate, to the *Exit* vertex on the augmented control flow graph, but not on every path via the *false* branch. Note also that there are no def-order edges in PRGs.

Methods for determining the placement of ϕ assignments and for determining control dependence edges for programs with reducible or irreducible flow of control are given in [Cytron89, Rosen88, Ferrante87, Reif82], all of which take time effectively quadratic in the size of the program [Cytron89]; however, for our restricted language, they can be determined in a simpler way. The placement of ϕ assignments can be determined by computing the set of live variables at each program statement and by computing the set of variables that have been assigned a value within the *if* and *while* statements. These two sets of variables can be computed in a syntax-directed manner. Except the control dependence edges that form self-loops on the *while* predicates and the control dependence edges incident on the ϕ vertices, control dependence edges can be determined from the nesting structure of the programs.

The flow dependence edges of a program representation graph are computed using data-flow analysis [Hecht77, Aho86]. For the restricted language considered in this thesis, the necessary computations can be

[9] In the program dependence graphs of [Horwitz88, Horwitz89], the control dependence edge from a *while* predicate to itself is omitted. However, such edges are needed for the Sequence-Congruence Algorithm (discussed in Chapter 4); so they are included in program representation graphs.

defined in a syntax-directed manner.

(a) **program** *Main*
    sum := 0
    x := 1
    while *x* < 11 **do**
        sum := *sum* + *x*
        x := *x* + 1
    od
    result := *result* + *sum*
end(*result*)

(b)

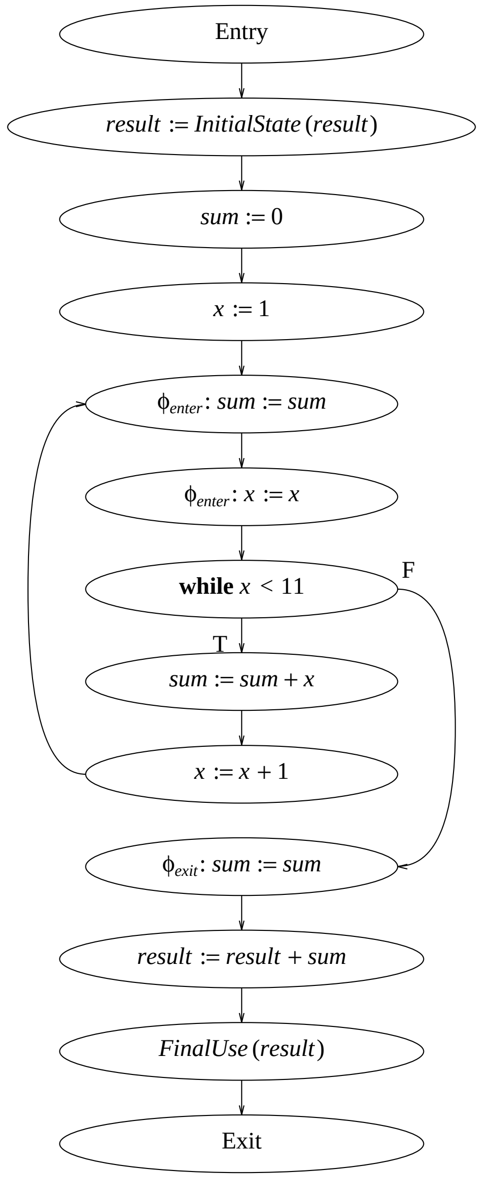


Figure 3-1. (a) is the same example program that appeared in Figure 2-1. (b) is the augmented control flow graph of the program. Note the absence of *InitialState* and *FinalUse* vertices for variables *sum* and *x* and of a ϕ_{exit} vertex for *x*.

Example. An example program and its augmented control flow graph are shown in Figure 3-1. Its program representation graph is shown in Figure 3-2. Note the absence of *InitialState* and *FinalUse* vertices for variables *sum* and *x* and of a ϕ_{exit} vertex for *x*. Note also that there is a control dependence edge from the *while* predicate $x < 11$ to itself. The boldface arrows represent control dependence edges; thin arrows represent flow dependence edges. The label on each control dependence edge—*true* or *false*—has been omitted.

3.2. Comparison with Static-Single-Assignment Form

Program representation graphs include some features of static-single-assignment forms. However, there are three differences between our PRGs and (the value graphs of) static-single-assignment forms:

(1) PRGs contain control dependence edges, whereas the value graphs of static-single-assignment forms do not. Control dependence edges were added so that the Sequence-Congruence Algorithm (to be discussed in Chapter 4) could take control dependences into account during partitioning.

(2) The ϕ statements in PRGs are slightly different from those in static-single-assignment forms. In the static-single-assignment forms defined in [Alpern88, Rosen88, Cytron89] a ϕ operator in a ϕ statement is a binary operator; that is, a ϕ statement is of the form "$x_1 := \phi(x_2, x_3)$." Variable occurrences are renamed (by adding subscripts, for example) so that each variable is assigned to exactly once in the pro-

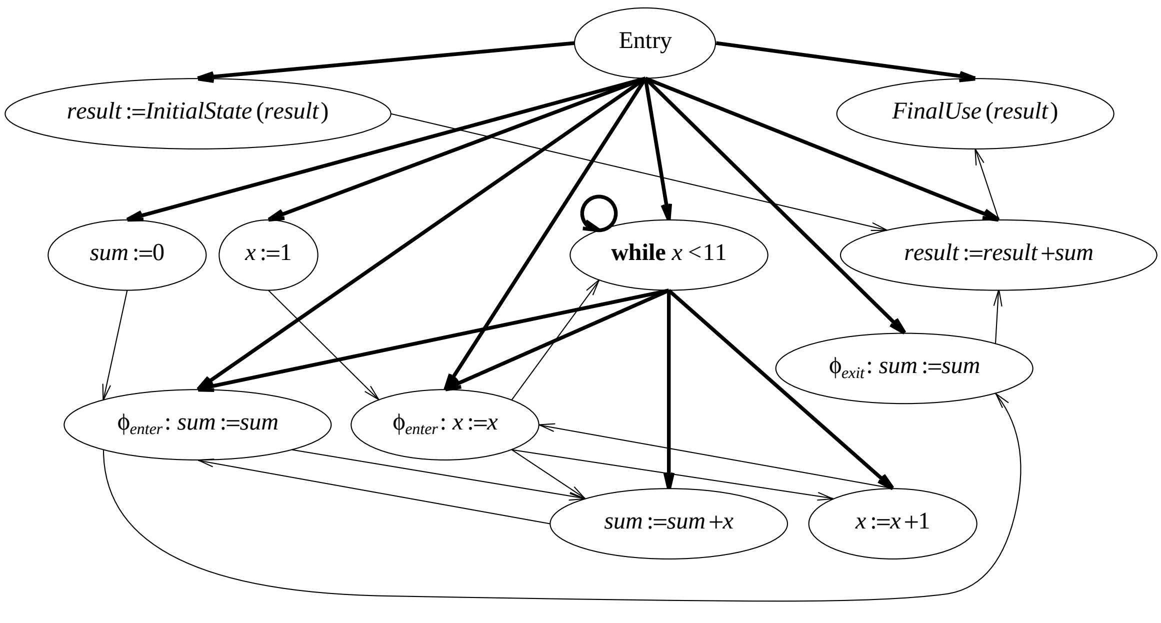


Figure 3-2. The program representation graph of the program in Figure 3-1(a). Note that there is a control dependence edge from the *while* predicate $x < 11$ to itself. The boldface arrows represent control dependence edges; thin arrows represent flow dependence edges. The label on each control dependence edge—*true* or *false*—has been omitted.

gram text (whence the name "static-*single-assignment* form"). Because variable renaming is not necessary for our purposes, we chose a simpler form of ϕ statement.

(3) Another difference between the ϕ statements in PRGs and those in static-single-assignment forms is that PRGs include only ϕ statements whose left-hand side variable is live (*i.e.*, there is a path from every ϕ vertex to a non-ϕ vertex). For instance, consider the following program fragment (which has been augmented with those ϕ statements that are included in its PRG):

```
       a := 0
<T1>   x := 1
       if p then
           x := 2
           a := x
       fi
       ϕ_if: a := a
<T2>   x := 3 + a
```

If the PRG were to include non-live ϕ statements, there would be a ϕ statement "ϕ_{if}: $x := x$" immediately after the *if* statement; however, this ϕ_{if} statement is not included in the PRG because the variable x is not live at this point.

The decision to exclude these extra ϕ statements from PRGs is motivated by the following concerns:

(1) The exclusion of non-live ϕ statements permits larger sets of semantically equivalent programs to have isomorphic PRGs. For example, the same PRG represents not only the program shown above, but also a version of the program in which T1 comes *after* the *if* statement (but before T2). It will be shown in the next section that PRGs and PDGs are equivalent program representations in the sense that two programs have isomorphic PRGs if and only if their PDGs are isomorphic. The two representations would not be equivalent in this way if PRGs were to contain the additional ϕ statements of previous definitions.

(2) Program slices can also be extracted from PRGs. The slice of a PRG R with respect to a set of vertices S is the subgraph of R induced by all vertices from which there is a path to an element of S in R. For the Feasibility Lemma to hold for PRGs, the non-live ϕ statements must *not* be included in PRGs. For instance, if non-live ϕ vertices were required in PRGs, the example given above would be

```
       a := 0
<T1>   x := 1
       if p then
           x := 2
           a := x
       fi
       ϕ_if: x := x
       ϕ_if: a := a
<T2>   x := 3 + a
```

Its slice with respect to T2 would correspond to the fragment

```
        a := 0
        if p then
            x := 2
            a := x
        fi
        φ_if: a := a
<T2>    x := 3 + a
```

However, the slice of the PRG does not correspond to any program; the fragment shown above is not annotated properly with ϕ statements so as to correspond to any program. In particular, because it lacks a (non-live) ϕ vertex ϕ_{if}: $x := x$ just after the *if* statement, it does *not* correspond to the program

```
        a := 0
        if p then
            x := 2
            a := x
        fi
<T2>    x := 3 + a
```

By excluding non-live ϕ statements from PRGs, infeasible slices do not arise; every slice of a PRG (with respect to a set of non-ϕ vertices) is isomorphic to the PRG of some program (see Section 7.1).

3.3. Equivalence of PDGs and PRGs

PRGs are very similar to the PDGs defined in Chapter 2. Both graphs have explicit representations of control and data dependences among program components. In this section, we will show that PDGs and PRGs are, in fact, equivalent program representations in that the PDGs of two programs are isomorphic if and only if their PRGs are isomorphic. Since PRGs and PDGs are equivalent program representations, many semantic properties of PDGs, in particular, the Equivalence Theorem, are also applicable to PRGs.

PDGs and PRGs differ in that PRGs do not include def-order edges; ϕ assignments are used instead. Both PDGs and PRGs have explicit representations of control and flow dependences among program components, but control and flow dependences cannot completely characterize programs' execution behaviors. There are inequivalent programs that have identical control and flow dependences among program components. PDGs use the order of certain assignment statements, that is, def-order dependences,[10] to supplement control and flow dependences. In contrast, PRGs use ϕ statements to annotate the reaching definitions at join points of (the control flow graphs of) the programs to distinguish inequivalent programs that have identical control and flow dependences. In essence, the ϕ statements introduce extra flow dependence edges that substitute for def-order edges.

Before we prove the theorem, we first clarify our notations. A *definition* to a variable x is a non-ϕ assignment statement that assigns a value to x. A ϕ assignment is *not* considered a definition to a variable. An *x-definition-free flow dependence path* is a sequence of flow dependence edges $a_1 \longrightarrow_f a_2$, $a_2 \longrightarrow_f a_3$, $\ldots$, $a_{k-1} \longrightarrow_f a_k$ so that the variable x is assigned a value at a_1 and is used at a_k, and all vertices except a_1 and a_k are ϕ vertices. An x-definition-free flow dependence path in the PRG of program P corresponds to an x-definition-free path in the control flow graph of P, which, in turn, corresponds to a flow dependence edge in

[10]Def-order dependences are similar to output dependences, which are widely discussed in the literature. By adopting def-order dependences instead of output dependences, PDGs allow larger collections of strongly equivalent programs.

the PDG of P.

A vertex a is a *control ancestor* of vertex b if there is a control dependence path from a to b (*while* predicates are not considered to be control ancestors of themselves). A vertex a is the *least* common control ancestor of vertices b and c if a is a common control ancestor of b and c and all other common control ancestors of b and c are control ancestors of a.

Note that in PRGs but not in PDGs, each *while* predicate has a control dependence edge that forms a self-loop on the predicate vertex. Since it is clear from the program texts of the vertices in PDGs and PRGs whether a vertex represents a *while* predicate, we will ignore this control dependence edge in the proof of Theorem 3.1.

The following lemma follows directly from the definitions of PDGs and PRGs. This lemma asserts that the PDG and the PRG of a program have the same set of non-ϕ vertices and the same set of control dependence edges among the non-ϕ vertices (ignoring the self-loops on the *while* predicates).

Lemma. For any program P with PDG G_P *and* PRG R_P, *G_P and R_P have the same set of non-ϕ vertices and the control dependence subgraph of G_P is isomorphic to the control dependence subgraph of R_P with all the ϕ vertices and the control dependence edges that form self-loops on* while *predicate vertices removed.*

Theorem 3.1 is main result of this section. The theorem states that PDGs and PRGs are equivalent program representations.

Theorem 3.1. *The* PDG*s of two programs are isomorphic if and only if their* PRG*s are isomorphic.*

Proof. Let P and Q be two programs and G_P, G_Q, R_P, and R_Q be their PDGs and PRGs, respectively. We want to prove that G_P and G_Q are isomorphic if and only if R_P and R_Q are isomorphic. The proof is divided into two parts, one for each direction.

Part I: $G_P \approx G_Q$ implies $R_P \approx R_Q$.

Suppose $G_P \approx G_Q$. Because $G_P \approx G_Q$, by the previous lemma, R_P and R_Q have the same set of non-ϕ vertices and the same incoming control dependence edges for the non-ϕ vertices. We need to show that (1) R_P and R_Q have the same ϕ vertices, (2) R_P and R_Q have the same incoming control dependence edges for the ϕ vertices, and (3) R_P and R_Q have the same flow dependence edges. Note that there are no outgoing control dependence edges from ϕ vertices in a program representation graph.

For any vertex c in R_P labeled "ϕ_{if}: $x := x$", there must be a non-ϕ vertex a inside the *if* statement that assigns a value to x and a non-ϕ vertex b that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P, which means there is an x-definition-free path from a to b in the control flow graph of P. Therefore, there is a flow dependence edge $a \longrightarrow_f b$ in G_P. Because $G_P \approx G_Q$, this flow dependence edge $a \longrightarrow_f b$ also exists in G_Q. Therefore, there is an x-definition-free path from a to b in the control flow graph of Q. If b is outside the *if* statement, then the variable x is live after the *if* statement; if b is inside the *if* statement, then the flow edge $a \longrightarrow_f b$ is carried by a loop that encloses the *if* statement, which means x is live after the *if* statement. In either case, x is live after the *if* statement. Since a is a vertex inside the *if* statement that assigns a value to x and x is live after the *if* statement, from the definition of PRGs, vertex c is added to the augmented control flow graph of Q; hence c is in R_Q. Conversely, for any vertex c' in R_Q labeled "ϕ_{if}: $x := x$", c' is also in R_P. Therefore, R_P and R_Q have the same ϕ_{if} vertices.

For any pair of associated vertices c and d in R_P labeled "ϕ_{enter}: $x := x$" and "ϕ_{exit}: $x := x$", respectively, there must be a non-ϕ vertex a inside the loop that assigns a value to x and a non-ϕ vertex b that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f d \longrightarrow_f^* b$ in R_P, which means there is an x-

definition-free path from a, via the predicate of the loop, to b in the control flow graph of P. Therefore, there is a flow dependence edge $a \longrightarrow_f b$ in G_P. Because $G_P \approx G_Q$, this flow dependence edge $a \longrightarrow_f b$ also exists in G_Q. Therefore, there is an x-definition-free path from a, via the predicate of the loop, to b in the control flow graph of Q. If b is outside the loop, then the variable x is live after the loop of c; if b is inside the loop, then the flow edge $a \longrightarrow_f b$ is carried by a loop that encloses the loop of c, which means x is live after the loop. In either case, x is live after the loop. Since a is a vertex inside the loop that assigns a value to x and x is live after the loop, from the definition of PRGs, vertices c and d are added to the augmented control flow graph of Q; hence c and d are in R_Q. Conversely, for any pair of associated vertices c' and d' in R_Q labeled "ϕ_{enter}: $x := x$" and "ϕ_{exit}: $x := x$", respectively, c' and d' are also in R_P. Therefore, R_P and R_Q have the same pairs of ϕ_{enter} and ϕ_{exit} vertices.

For any vertex c in R_P labeled "ϕ_{enter}: $x := x$" (without the associated "ϕ_{exit}: $x := x$" vertex), there must be a non-ϕ vertex a inside the loop that assigns a value to x and a non-ϕ vertex b inside the loop that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P, which means there is an x-definition-free path from a, via the predicate of the loop, to b in the control flow graph of P. Therefore, there is a loop-carried flow dependence edge $a \longrightarrow_f b$ in G_P. Because $G_P \approx G_Q$, this loop-carried flow dependence edge $a \longrightarrow_f b$ also exists in G_Q. Therefore, there is an x-definition-free path from a, via the predicate of the loop, to b in the control flow graph of Q. Since a is a vertex inside the loop that assigns a value to x and b is a vertex inside the loop that uses x and there is an x-definition-free path from a, via the predicate of the loop, to b in the control flow graph of Q, from the definition of PRGs, the vertex c is added to the augmented control flow graph of Q; hence vertex c is in R_Q. Conversely, for any vertex c' in R_Q labeled "ϕ_{enter}: $x := x$" (without the associated "ϕ_{exit}: $x := x$" vertex), c' is also in R_P. Therefore, R_P and R_Q have the same ϕ_{enter} vertices.

We have shown that R_P and R_Q have the same ϕ vertices. Next we want to prove that R_P and R_Q have the same incoming control dependence edges for the ϕ vertices.

From the definition of PRGs, a ϕ_{if} vertex has the same control predecessor as the associated *if*-predicate in R_P (or R_Q). By the previous lemma, the *if*-predicate has the same control predecessor in both R_P and G_P (or in both R_Q and G_Q). Because $G_P \approx G_Q$, the *if*-predicate has the same control predecessor in G_P and in G_Q. So a ϕ_{if} vertex has the same control predecessor in R_P and in R_Q. Thus, R_P and R_Q have the same incoming control dependence edges for the ϕ_{if} vertices.

From the definition of PRGs, a ϕ_{exit} vertex has the same control predecessor as the associated *while*-predicate in R_P (or R_Q). By the previous lemma, the *while*-predicate has the same control predecessor in both R_P and G_P (or in both R_Q and G_Q). Because $G_P \approx G_Q$, the *while*-predicate has the same control predecessor in G_P and in G_Q. So a ϕ_{exit} vertex has the same control predecessor in R_P and in R_Q. Thus, R_P and R_Q have the same incoming control dependence edges for the ϕ_{exit} vertices.

From the definition of PRGs, a ϕ_{enter} vertex has two control predecessors in R_P and in R_Q: the associated *while*-predicate and its control predecessor. By the same arguments as above, these two control predecessors of a ϕ_{enter} vertex in R_P are the same as the two control predecessors of the same ϕ_{enter} vertex in R_Q. Thus, R_P and R_Q have the same incoming control dependence edges for the ϕ_{enter} vertices.

We have shown that R_P and R_Q have the same ϕ vertices and the same incoming control dependence edges for the ϕ vertices. Finally, we need to show that R_P and R_Q have the same flow dependence edges.

For any flow dependence edge $a \longrightarrow_f b$ in R_P, if b is a ϕ vertex, let b' be a non-ϕ vertex in R_P such that there is a flow dependence path $b \longrightarrow_f^* b'$ in R_P and all vertices on this path, except b', are ϕ vertices. If b is not a ϕ vertex, then let b' be b.

Let $a \longrightarrow_f b$ be any flow dependence edge in R_P and b' be defined as above. We want to show that the edge $a \longrightarrow_f b$ is in R_Q.

Because a and b may be ϕ_{if}, ϕ_{enter}, ϕ_{exit}, or non-ϕ vertices, there are four cases, each with four subcases, to consider:

Case 1. Suppose a is a vertex labeled "ϕ_{if}: $x := x$." Let S_a be the *if* statement for a. There must be a non-ϕ vertex c inside S_a such that x is assigned a value at c and there is an x-definition-free flow dependence path $c \longrightarrow_f^* a$ in R_P. Since there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f b \longrightarrow_f^* b'$ in R_P, the definition to x at c can reach the use of x at b'. Thus, there is a flow dependence edge $c \longrightarrow_f b'$ in G_P.

(1) Suppose b is a vertex labeled "ϕ_{if}: $x := x$." Let S_b be the *if* statement for b. Since there is a flow dependence edge $a \longrightarrow_f b$ in R_P, there are two possibilities: either S_a occurs before S_b or S_a is nested within S_b. (It is not possible that S_b occurs before S_a, nor is it possible that S_b is nested within S_a due to the existence of the flow dependence edge $a \longrightarrow_f b$ in R_P.)

First consider the possibility that S_a occurs before S_b in R_P. There must be a non-ϕ vertex d inside S_b such that x is assigned a value at d and there is an x-definition-free flow dependence path $d \longrightarrow_f^* b$ in R_P. Since there is an x-definition-free flow dependence path $d \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_P, the definition to x at d can reach the use of x at b'. Thus, there is a flow dependence edge $d \longrightarrow_f b'$ in G_P. Because there is a flow dependence edge $c \longrightarrow_f b'$ in G_P, and because S_a occurs before S_b, there must be a def-order edge $c \longrightarrow_{do(b')} d$ in G_P. Because $G_P \approx G_Q$, the flow dependence edges $c \longrightarrow_f b'$ and $d \longrightarrow_f b'$ and the def-order edge $c \longrightarrow_{do(b')} d$ are also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S sitting in between S_a and S_b and there is a non-ϕ vertex e inside S which assigns a value to x and reaches the end of S. If this were the case, there would be two def-order edges $c \longrightarrow_{do(b')} e$ and $e \longrightarrow_{do(b')} d$ in G_Q. Because $G_P \approx G_Q$, the two def-order edges $c \longrightarrow_{do(b')} e$ and $e \longrightarrow_{do(b')} d$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

Next consider the possibility that S_a is nested within S_b in R_P. Because there is a flow dependence edge $c \longrightarrow_f b'$ in G_P and because $G_P \approx G_Q$, the flow dependence edge $c \longrightarrow_f b'$ is in G_Q as well. Note that c is inside S_a, which is nested within S_b, and b' is outside S_b in G_Q. Because there is a flow dependence edge $c \longrightarrow_f b'$ in G_Q, there must be an x-definition-free path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S which is in the same branch of S_b as S_a and which occurs after S_a and there is a non-ϕ vertex e inside S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $c \longrightarrow_{do(b')} e$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $c \longrightarrow_{do(b')} e$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

(2) Suppose b is a vertex labeled "ϕ_{enter}: $x := x$." Let S_b be the *while* statement for b. Since there is a flow dependence edge $a \longrightarrow_f b$ in R_P, there are two possibilities: either S_a occurs before S_b or S_a is nested within S_b. (It is not possible that S_b occurs before S_a nor is it possible that S_b is nested within S_a due to the existence of the flow dependence edge $a \longrightarrow_f b$ in R_P.)

First consider the possibility that S_a occurs before S_b in R_P. There must be a non-ϕ vertex d inside S_b such that x is assigned a value at d and there is an x-definition-free flow dependence path $d \longrightarrow_f^* b$ in R_P. Since there is an x-definition-free flow dependence path $d \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_P, the definition to x at d can reach the use of x at b'. Thus, there is a flow dependence edge $d \longrightarrow_f b'$ in G_P. Because there is a flow dependence edge $c \longrightarrow_f b'$ in G_P, there must be a def-order edge $c \longrightarrow_{do(b')} d$ in G_P. Because $G_P \approx G_Q$, the flow dependence edges $c \longrightarrow_f b'$ and $d \longrightarrow_f b'$ and the def-order edge $c \longrightarrow_{do(b')} d$ are also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S sitting in between S_a and S_b and there is a non-ϕ vertex e inside S that assigns a value to x and reaches the end of S. If this were the case, there would be two def-order edges $c \longrightarrow_{do(b')} e$ and $e \longrightarrow_{do(b')} d$ in G_Q. Because $G_P \approx G_Q$, the def-order edges $c \longrightarrow_{do(b')} e$ and $e \longrightarrow_{do(b')} d$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

Next consider the possibility that S_a is nested within S_b in R_P. Because there is a flow dependence edge $c \longrightarrow_f b'$ in G_P and because $G_P \approx G_Q$, the flow dependence edge $c \longrightarrow_f b'$ is in G_Q as well. Note that c is inside S_a, which is nested within S_b, and there is a flow dependence edge $c \longrightarrow_f b'$ in G_Q. If b' is outside S_b in G_Q, then the x-definition-free path from c to b' must pass through the predicate of S_b in the control flow graph since the flow dependence edge $c \longrightarrow_f b'$ is G_Q. If b' is inside S_b in G_Q, then $c \longrightarrow_f b'$ is loop-carried by S_b; in this case, the x-definition-free path from c to b' must also pass through the predicate of S_b in the control flow graph since the flow dependence edge $c \longrightarrow_f b'$ is G_Q. In either case, there must be an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S inside S_b which occurs after S_a and there is a non-ϕ vertex e inside S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $c \longrightarrow_{do(b')} e$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $c \longrightarrow_{do(b')} e$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

(3) Suppose b is a vertex labeled "ϕ_{exit}: $x := x$." This case is impossible since there cannot be a flow dependence edge from a ϕ_{if} vertex to a ϕ_{exit} vertex in R_P.

(4) Suppose b is a non-ϕ vertex. In this case b' is the same vertex as b. Because $G_P \approx G_Q$, the loop-independent flow edge $c \longrightarrow_f b$ is also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S sitting in between S_a and b and there is a non-ϕ vertex e in S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $c \longrightarrow_{do(b)} e$ and a loop-independent flow edge $e \longrightarrow_f b$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $c \longrightarrow_{do(b)} e$ and the flow edge $e \longrightarrow_f b$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

Case 2. Suppose a is a vertex labeled "ϕ_{exit}: $x := x$." This case is similar to *Case 1* above; the only difference is that c is contained in a *while* statement rather than in an *if* statement.

Case 3. Suppose a is a non-ϕ vertex. This case is similar to *Case 1* above; the only difference is that c and a are the same vertex.

Case 4. Suppose a is a vertex labeled "ϕ_{enter}: $x := x$." Let S_a be the *while* statement for a. There must be a non-ϕ vertex c occurring before S_a such that x is assigned a value at c and there is an x-definition-free flow dependence path $c \longrightarrow_f^* a$ in R_P. Since there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f b \longrightarrow_f^* b'$ in R_P, the definition to x at c can reach the use of x at b'. Thus, there is a flow dependence edge $c \longrightarrow_f b'$ in G_P.

(1) Suppose b is a vertex labeled "ϕ_{if}: $x := x$." Let S_b be the *if* statement for b. Since there is a flow dependence edge $a \longrightarrow_f b$ in R_P, S_b is nested within S_a. Because b is a vertex labeled "ϕ_{if}: $x := x$," there must be a non-ϕ vertex d inside S_b such that x is assigned a value at d and there is an x-definition-free flow dependence path $d \longrightarrow_f^* b$ in R_P. Since there is an x-definition-free flow dependence path $d \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_P, the definition to x at d can reach the use of x at b'. Thus, there is a flow dependence edge $d \longrightarrow_f b'$ in G_P. Because $G_P \approx G_Q$, the flow dependence edges $c \longrightarrow_f b'$ and $d \longrightarrow_f b'$ are also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S inside S_a which occurs before S_b and there is a non-ϕ vertex e inside S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $e \longrightarrow_{do(b')} d$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $e \longrightarrow_{do(b')} d$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

(2) Suppose b is a vertex labeled "ϕ_{enter}: $x := x$." Let S_b be the *while* statement for b. Since there is a flow dependence edge $a \longrightarrow_f b$ in R_P, S_b is nested within S_a. Because b is a vertex labeled "ϕ_{enter}: $x := x$," there must be a non-ϕ vertex d inside S_b such that x is assigned a value at d and there is an x-definition-free flow dependence path $d \longrightarrow_f^* b$ in R_P. Since there is an x-definition-free flow dependence path $d \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_P, the definition to x at d can reach the use of x at b'. Thus, there is a flow dependence edge $d \longrightarrow_f b'$ in G_P. Because $G_P \approx G_Q$, the flow dependence edges $c \longrightarrow_f b'$ and $d \longrightarrow_f b'$ are also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b \longrightarrow_f^* b'$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S inside S_a which occurs before S_b and there is a non-ϕ vertex e in S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $e \longrightarrow_{do(b')} d$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $e \longrightarrow_{do(b')} d$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

(3) Suppose b is a vertex labeled "ϕ_{exit}: $x := x$." Note that R_P and R_Q have the same pairs of ϕ_{enter} and ϕ_{exit} vertices and there is always a flow dependence edge from ϕ_{enter} vertex to the accompanying ϕ_{exit} vertex in both R_P and R_Q. Thus, R_P and R_Q both contains the flow dependence edge $a \longrightarrow_f b$.

(4) Suppose b is a non-ϕ vertex. In this case b' is the same vertex as b. Because $G_P \approx G_Q$, the loop-independent flow edge $c \longrightarrow_f b$ is also in G_Q. Therefore, there is an x-definition-free flow dependence path $c \longrightarrow_f^* a \longrightarrow_f^* b$ in R_Q. Under this condition, the only possibility that the edge $a \longrightarrow_f b$ is not in R_Q (that is, there are other ϕ vertices on the flow dependence path from a to b) is that there is a *while* (or *if*) statement S inside S_a which occurs before b and there is a non-ϕ vertex e in S that assigns a value to x and reaches the end of S. If this were the case, there would be a def-order edge $c \longrightarrow_{do(b)} e$ and a loop-independent flow edge $e \longrightarrow_f b$ in G_Q. Because $G_P \approx G_Q$, the def-order edge $c \longrightarrow_{do(b)} e$ and the flow edge $e \longrightarrow_f b$ would occur in G_P as well, which would imply that the flow edge $a \longrightarrow_f b$ was absent in R_P. This contradicts our assumption, so we conclude that the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

Therefore, in any of the above cases, the flow dependence edge $a \longrightarrow_f b$ also appears in R_Q. Conversely, all flow dependence edges in R_Q also appear in R_P. Therefore, R_P and R_Q have the same flow dependence edges.

This completes the proof that $G_P \approx G_Q$ implies $R_P \approx R_Q$.

Part II: $R_P \approx R_Q$ implies $G_P \approx G_Q$.

Suppose $R_P \approx R_Q$. Note that all vertices in G_P and G_Q are non-ϕ vertices. From the previous lemma, G_P and G_Q have the same (non-ϕ) vertices and the same control dependence edges. We need to show that G_P and G_Q have the same flow dependence and def-order edges.

For any loop-independent flow dependence edge $a \longrightarrow_{li} b$ in G_P, a must be either an *InitialState* or an assignment statement vertex. Let x be the variable that is assigned a value at a. Because $a \longrightarrow_{li} b$ is a loop-independent flow dependence edge in G_P, there is an x-definition-free path from a to b in the control flow graph of P that does not include the back edge of any loop L that encloses both a and b. From the definition of PRGs, there is an x-definition-free flow dependence path $a \longrightarrow_f^* b$ in R_P. Because the x-definition-free path from a to b in the control flow graph of P does not include the back edge of any loop L that encloses both a and b, the flow dependence path $a \longrightarrow_f^* b$ in R_P does not include the "ϕ_{enter}: $x := x$" vertex for any loop L that encloses both a and b. Because $R_P \approx R_Q$, there is an identical flow dependence path $a \longrightarrow_f^* b$ in R_Q. Therefore, there is an x-definition-free path from a to b in the control flow graph of Q that does not include the back edge of any loop L that encloses both a and b; hence there is a loop-independent flow dependence edge $a \longrightarrow_{li} b$ in G_Q.

For any loop-carried flow dependence edge $a \longrightarrow_{lc(L)} b$ in G_P, a must be an assignment statement vertex. Let x be the variable that is assigned a value at a. From the definition of PDGs, both a and b are enclosed in loop L in G_P. Because G_P and G_Q have the same control dependence edges, both a and b are enclosed in loop L in G_Q. Because the loop-carried flow dependence edge $a \longrightarrow_{lc(L)} b$ is carried by loop L in G_P, there is an x-definition-free path from a to b in the control flow graph of P that includes the back edge of loop L. From the definition of PRGs, there is an x-definition-free flow dependence path $a \longrightarrow_f^* b$ in R_P. Because the x-definition-free path from a to b in the control flow graph of P includes the back edge of loop L, the flow dependence path $a \longrightarrow_f^* b$ in R_P includes the "ϕ_{enter}: $x := x$" vertex for loop L. Because $R_P \approx R_Q$, there is an identical flow dependence path $a \longrightarrow_f^* b$ in R_Q. Therefore, there is an x-definition-free path from a to b in the control flow graph of Q that includes the back edge of loop L; hence there is a loop-carried flow dependence edge $a \longrightarrow_{lc(L)} b$ in G_Q.

We have shown that all flow dependence edges in G_P are in G_Q. Similarly, all flow dependence edges in G_Q are in G_P. Therefore, G_P and G_Q have the same flow dependence edges.

For any def-order edge $a \longrightarrow_{do(c)} b$ in G_P, there are flow dependence edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ in G_P and a and b are in the same branch of any *if* statement that encloses both of them and a occurs before b in P. We have already shown that G_P and G_Q have the same control and flow dependence edges. Thus, there are flow dependence edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ in G_Q and a and b are in the same branch of any *if* statement that encloses both of them. What remains to be shown is that a occurs before b in Q.

Since a occurs before b in P, there are three cases to consider, depending on the relative order of a, b, and c in P.

Case 1. Suppose the relative order is a, c, and b in P. Since a occurs before c and there is a flow dependence edge $a \longrightarrow_f c$ in G_P, there must be a loop-independent flow dependence edge $a \longrightarrow_{li} c$ in G_P. Since G_P and G_Q have the same flow dependence edges, there is an identical loop-independent flow dependence edge

$a \longrightarrow_{li} c$ in G_Q. Therefore, a occurs before c in Q. Since b occurs after c in P, there cannot be a loop-independent flow dependence edge $b \longrightarrow_{li} c$ in G_P. Since G_P and G_Q have the same flow dependence edges, there cannot be a loop-independent flow dependence edge $b \longrightarrow_{li} c$ in G_Q. The flow dependence edge $b \longrightarrow_f c$ in G_Q must be loop-carried; that is, b occurs after c in Q. Because a occurs before c and b occurs after c in Q, a occurs before b in Q.

Case 2. Suppose the order is a, b, and c in P. In this case, the control predecessor of b cannot be a control ancestor of a for otherwise there cannot be a flow dependence edge $a \longrightarrow_f c$. Based on the control structure of P, there are two subcases to consider:

(1) The least common control ancestor of b and c is not a control ancestor of a. Let d be the least common control ancestor of b and c. Since a occurs before c and there is a flow dependence edge $a \longrightarrow_f c$ in G_P, there must be a loop-independent flow dependence edge $a \longrightarrow_{li} c$ in G_P. Since G_P and G_Q have the same flow dependence edges, there is an identical loop-independent flow dependence edge $a \longrightarrow_{li} c$ in G_Q. Therefore, a occurs before c in Q. Since d is a control ancestor of c but is not a control ancestor of a and a occurs before c, a occurs before d in Q. Since a occurs before d and d is a control ancestor of b in Q, a must occur before b in Q.

(2) The least common control ancestor of b and c is a control ancestor of a. Let x be the variable that is assigned a value at a and b; let d be the highest control ancestor of b that is not a control ancestor of a. Note that d cannot be b and that the control predecessor of d is a control ancestor of a in G_P. If d is a predicate of an *if* statement, let e be the associated "ϕ_{if}: $x := x$" vertex of the *if* statement. If d is a predicate of a *while* statement, let e be the associated "ϕ_{exit}: $x := x$" vertex of the *while* statement. Since $a \longrightarrow_f c$ is a flow dependence edge in G_P and the relative order is a, b, and c, there must be an x-definition-free flow dependence path $a \longrightarrow_f^* e$ in R_P that does not include the ϕ_{enter} vertices of any loop that encloses both a and b. Because $R_P \approx R_Q$, there is an identical flow dependence path $a \longrightarrow_f^* e$ in R_Q. Therefore, a must occur before d in Q. Since a occurs before d and d is a control ancestor of b in Q, a occurs before b in Q.

Case 3. Suppose the order is c, a, and b in P. In this case, the control predecessor of b cannot be a control ancestor of a for otherwise there cannot be a flow dependence edge $a \longrightarrow_f c$. Based on the control structure of P, there are two subcases to consider:

(1) The least common control ancestor of a and c is not an control ancestor of b. Let d be the least common control ancestor of a and c. Since b occurs after c, there cannot be a loop-independent flow dependence edge $b \longrightarrow_{li} c$ in G_P. Since G_P and G_Q have the same flow dependence edges, there cannot be a loop-independent flow dependence edge $b \longrightarrow_{li} c$ in G_Q. The flow dependence edge $b \longrightarrow_f c$ in G_Q must be loop-carried; that is, b occurs after c in Q. Since d is a control ancestor of c but is not a control ancestor of b and b occurs after c, b occurs after d in Q. Since b occurs after d and d is a control ancestor of a in Q, b must occur after a in Q.

(2) The least common control ancestor of a and c is a control ancestor of b. Let x be the variable that is assigned a value at a and b; let d be the highest control ancestor of b that is not a control ancestor of a. Note that d cannot be b and that the control predecessor of d is a control ancestor of a in G_P. If d is a predicate of an *if* statement, let e be the associated "ϕ_{if}: $x := x$" vertex for the *if* statement. If d is a predicate of a *while* statement, let e be the associated "ϕ_{exit}: $x := x$" vertex for the *while* statement. Since $a \longrightarrow_f c$ is a flow dependence edge in G_P and the relative order is c, a, and b, there must be an x-definition-free flow dependence path $a \longrightarrow_f^* e$ in R_P that does not include the ϕ_{enter} vertices of any loop that encloses both a and b. Because $R_P \approx R_Q$, there is an identical flow dependence path $a \longrightarrow_f^* e$ in R_Q.

Therefore, a must occur before d in Q. Since a occurs before d and d is a control ancestor of b in Q, a occurs before b in Q.

In all of the above three cases, a occurs before b in Q. Therefore, there is a def-order edge $a \longrightarrow_{do(c)} b$ in G_Q. Similarly, for an def-order edge $a \longrightarrow_{do(c)} b$ in G_Q, there is an identical def-order edge $a \longrightarrow_{do(c)} b$ in G_P. We conclude that G_P and G_Q have the same def-order edges.

This completes the proof that $R_P \approx R_Q$ implies $G_P \approx G_Q$. □

Though PRGs and PDGs are equivalent program representations, PRGs are, in general, smaller than PDGs. This is because there are fewer ϕ vertices in a PRG than def-order edges in the equivalent PDG. For instance, consider the following example.

```
<T1>    x := 1
        if p2 then
<T2>        x := 2
        fi
        if p3 then
<T3>        x := 3
        fi
        . . .
        if pk then
<Tk>        x := k
        fi
        y1 := x
        y2 := x * 2
        . . .
        ym := x * m
```

In the PDG for this program fragment, for each use of x in the assignments to y_i's, there are def-order edges from T1 to T2, T3, ..., Tk and from T2 to T3, ..., Tk, *etc*. In total, there are $mk(k-1)/2$ def-order edges and mk flow dependence edges. However, in the PRG for this program fragment, there are $(k-1)$ ϕ_{if} vertices, $(k-1)$ control dependence edges for these ϕ_{if} vertices, and $(2k-2+m)$ flow dependence edges. In the worst case, the total number of edges and vertices in the PRG is $O(dn)$, whereas it is $O(n^3)$ in the program's PDG, where n is the number of non-ϕ components in the program and d is the maximum nesting depth.

3.4. Equivalence Theorem for Program Representation Graphs

Since PRGs and PDGs are equivalent program representations, many semantic properties of PDGs can be directly adapted for PRGs; in particular, the Equivalence Theorem, which states that two programs that have isomorphic PDGs have equivalent execution behavior, also holds for PRGs: two programs that have isomorphic PRGs have equivalent execution behavior.

Textually different programs may have isomorphic PRGs. The difference among these textually different programs that have isomorphic PRGs is that certain independent statements are in different order in these programs. Because there is no dependence among these out-of-order statements, the execution behaviors of programs are not affected except that these out-of-order statements are executed in different order. This is confirmed by the Equivalence Theorem for PRGs.

The following Equivalence Theorem for PRGs follows immediately from Theorem 3.1 and the Equivalence Theorem for PDGs, which is proved as a corollary of the Slicing Theorem in Chapter 2.

Theorem. (Equivalence Theorem for Program Representation Graphs). *Suppose that P and Q are programs for which* $R_P \approx R_Q$. *If* σ *is a state on which P terminates normally, then for any state* σ' *that agrees with* σ *on all variables for which there are initial-definition vertices in* R_P*: (1) Q terminates normally on* σ'*, (2) P and Q compute the same sequence of values at each corresponding program component, and (3) the final states agree on all variables for which there are final-use vertices in* R_P.

Chapter 4

The Sequence-Congruence Algorithm

One of the steps in the new program-integration algorithm is to determine whether two program components have equivalent execution behavior. Any technique that can detect program components with equivalent behaviors can be used in this step of the new program integration algorithm. In particular, we have developed the Sequence-Congruence Algorithm for this purpose.

The Sequence-Congruence Algorithm is based on an idea of Alpern, Wegman and Zadeck [Alpern88] for finding equivalence classes of program components by first optimistically grouping possibly equivalent components in an initial partition of a value graph and then finding the coarsest partition of the graph compatible with the initial partition and dependences among components. However, in refining the initial partition, the algorithm of Alpern et al. considers only flow dependences among components. They showed that components in the same final partition produce the same values at *certain moments* during execution. In contrast, the Sequence-Congruence Algorithm given in this chapter considers control dependences as well as flow dependences and can detect components with equivalent *behaviors*.

A further point of contrast between our work and that of Alpern et al. concerns the idea of applying partitioning to more than one program simultaneously. This idea was not studied by Alpern et al. [Alpern88], and the semantic property proved there concerning congruent vertices does not characterize the result of applying the algorithm to multiple programs simultaneously. The algorithm of Alpern et al. is essentially the first phase of our Sequence-Congruence Algorithm, and our first result (the Data-Congruence Lemma) establishes a semantic property of components in the same equivalence class when the algorithm is applied to multiple programs. We then go on to show that with an additional partitioning phase, it is possible to detect program components with equivalent behaviors even though they occur in different programs.

This chapter defines the notion of equivalent behavior of program components, presents the Sequence-Congruence Algorithm, and shows that the Sequence-Congruence Algorithm can detect program components with equivalent behavior. Finally, Section 4.4 describes three simple enhancements to the Sequence-Congruence Algorithm.

4.1. Equivalent Execution Behavior

A fundamental problem in program integration is to determine whether two program components have equivalent execution behavior. It is useful first to define precisely the notion of equivalent execution behavior. In Chapter 2, when the semantic foundations of the HPR algorithm were discussed, two program components were said to have equivalent behavior if they produce the same sequences of values during program execution. However, in that discussion, we were concerned only with those initial states on which the programs terminate normally. In order to account for initial states on which programs do not terminate normally, the notion of equivalent behavior is generalized as follows.

Definition. Two components c_1 and c_2 of programs P_1 and P_2, respectively, have *equivalent behavior* if and only if all four 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 sequences of values produced at c_1 and c_2 are identical.

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

By "the sequence of values produced at a component" we mean: For an assignment statement (including initial-definition statements and ϕ statements), the sequence of values assigned to the left-hand-side variable; for a predicate, the sequence of boolean values to which the predicate evaluates; and for a variable named in the **end** statement, the final value of that variable. Note that a fault such as integer overflow is considered to be a special "value" in the above definition. The program stops immediately after such a value is produced.

Our study of the semantic foundations for the HPR algorithm given in Chapter 2 considered only Case 1 of the above definition; that is, the Slicing Theorem, the Termination Theorem, and the Integration Theorem are concerned only with the case where both P_1 and P_2 terminate normally on an initial state. However, since the Slicing Theorem can be viewed as a special case of the Sequence-Congruence Theorem given in this chapter (this assertion will be obvious after the Sequence-Congruence Theorem is introduced), the semantic foundations for the HPR algorithm can be generalized to cover all four cases of the definition of equivalent behavior of program components given above.

4.2. The Sequence-Congruence Algorithm

It is an undecidable problem to determine whether two components, which may possibly be located in different programs, have equivalent behavior. A *safe* method must be employed in the integration algorithm. A method is safe if it never identifies two components as equivalent that are actually inequivalent. In the HPR integration algorithm, this is done by comparing program slices; we have shown that two components with isomorphic slices have equivalent behavior (namely, the Slicing Theorem in Chapter 2). The Sequence-Congruence Algorithm can also detect components with equivalent behavior, and that algorithm is strictly stronger than comparing program slicing in that all pairs of components with isomorphic program slices are found by the Sequence-Congruence Algorithm as well, but not *vice versa*.

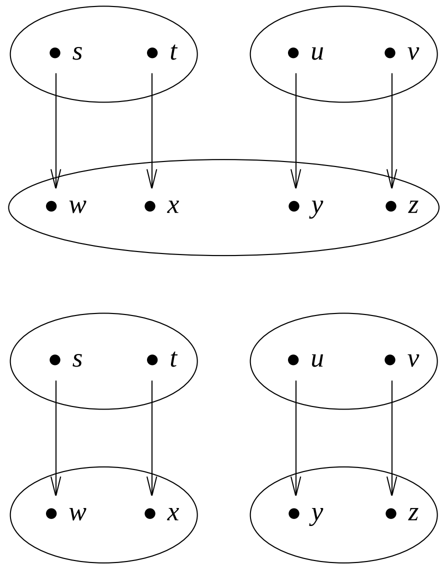
A component's execution behavior depends on three factors: the operator in the component, the operands available when the operator is applied, and the predicates that control the execution of the operation. It is safe to assume that components with different operators, different operands, or different controlling predicates will have different execution behaviors (although there do exist program components that have equivalent behavior but have different operators, inequivalent operands, or inequivalent controlling predicates).

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

The initial partition is refined by a partitioning algorithm, which will be discused in detail later in this section. The partitioning algorithm finds the coarsest partition consistent with the initial partition in which two vertices are in the same class only if their predecessors are in the same class. The following figure illustrates the partitioning process.

(1) Before partitioning.

(2) After partitioning.



In the above figure, vertices *s* and *t* are in the same initial class; *u* and *v* are in the same initial class; *w*, *x*, *y*, and *z* are in the same initial class before partitioning. Because predecessors of *w* and *x* and predecessors of *y* and *z* are in different classes, the class containing *w*, *x*, *y*, and *z* is split into two classes after partitioning: one for *w* and *x* and the other for *y* and *z*.

Program representation graphs are particularly suitable for the Sequence-Congruence Algorithm. Program components are represented as vertices in the graphs. The operators of components are readily identifiable from the text in the vertices. Flow and control dependences are explicitly represented as edges in the graphs.

One way to find program components that have the same operators, equivalent operands, and equivalent controlling predicates is by considering control and flow dependence edges during the same partitioning phase; however, we can find larger equivalence classes by considering flow and control edges at different phases. We are able to divide the partitioning process into two passes: During the first pass, we consider only flow dependence edges (and some additional edges); during the second pass, we consider only control dependence edges.

Given one or more PRGs, the Sequence-Congruence Algorithm consists of two partitioning passes. Vertices (that is, program components) 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.

First, we explain what the operator in a vertex is. 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. Vertices that represent either simple predicates or simple assignments 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" and "$y := x$."

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 PRGs, 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. Thus, we can speak of the "analogous" flow (or control) predecessors of the two vertices. To be more precise, we assign *types* to edges in the PRGs; the notion of analogous flow (or control) predecessors of two vertices is then defined in terms of the types of edges.

Due to the presence of ϕ vertices in PRGs, each use of a variable in a non-ϕ vertex is reached by exactly one assignment to the variable, either an original assignment statement, an *InitialState* assignment, or a ϕ assignment. 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 $op1$, $op2$, . . ., opn, one for each operand.

A vertex u labeled "ϕ_{if}: $x := x$" has two incoming flow dependence edges: one represents the value that flows to u from or via the *true* branch of the associated *if* statement; the other represents the value that flows to u from or via the *false* branch. The flow dependence edges incident on a ϕ_{if} vertex are assigned types *if-true* and *if-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 definition at T1 reaches T3 via the *false* branch of the *if* statement; so the flow dependence edge from T1 to T3 has type *if-false*. The definition at T2 reaches T3 from the *true* branch; so the flow dependence edge from T2 to T3 has type *if-true*.

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

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 *self−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 *enter-true* or *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 analogous flow (or control) predecessors of two vertices u_1 and u_2 are two vertices v_1 and v_2 such that the flow (or control, respectively) dependence edges $u_1 \longrightarrow v_1$ and $u_2 \longrightarrow v_2$ have the same type. If v_1 and v_2 are the analogous flow predecessors of u_1 and u_2 and if v_1 and v_2 assign a value to variables a_1 and a_2, then we say variables a_1 and a_2 are *analogous operands* of vertices u_1 and u_2, respectively.

Having explained the operator in a vertex and analogous flow (control) predecessors, we now proceed to describe the basic partitioning algorithm. The basic partitioning algorithm in Figure 4-1 is adapted from [Alpern88, Aho74], which is based on an algorithm of [Hopcroft71] for minimizing a finite state machine. 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; it guarantees that two vertices v and v' are in the same class after partitioning if and only if they are in the same initial partition, and, for every predecessor u of v, there is an analogous predecessor u' of v' such that u and u' are in the same class after partitioning, 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 4-2 presents the Sequence-Congruence Algorithm, which operates on one or more program representation graphs. When the algorithm operates on more than one PRG, the multiple PRGs are treated as one graph; thus, when we refer below to "the graph," we mean the collection of PRGs.

*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 PRGs. 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 *InitialState* 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} 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 *FinalUse* 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.)

```
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 4-1. The basic partitioning algorithm. This algorithm, which is adapted from [Alpern88, Aho74, Hopcroft71], finds the coarsest partition of a graph's vertices that is compatible with a given initial partition and the edges in the graph.

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.

Definition.[11] Vertices are *data-congruent* if they are in the same class after the first partitioning pass. Vertices are *sequence-congruent* if they are in the same class after the second partitioning pass.

The partitioning algorithm can determine the data-congruence classes in time $O(E_1 \log E_1)$ where E_1 is the number of flow dependence edges plus the number of ϕ_{if} and ϕ_{exit} vertices. The sequence-congruence

[11]Our terminology differs from that of [Alpern88]: our concept of *data-congruence* is similar to that of *congruence* in [Alpern88]; our concept of *sequence-congruence* is a new concept that does not appear in [Alpern88].

classes can be determined by the algorithm in time $O(E_1 \log E_1 + E_2 \log E_2)$ where E_1 is as above and E_2 is the number of control dependence edges in the graph.

Example. Figure 4-3 shows an example of partitioning. The two assignments $x := 1$ and $u := 1$ are in the same initial class and they stay in the same class after the first partitioning pass; however, they are separated during the second pass. Thus, the two components are data-congruent but not sequence-congruent. The two *FinalUse* vertices are sequence-congruent even though they are associated with different variables.

4.3. The Sequence-Congruence Theorem

Program components represented by sequence-congruent vertices (in short, sequence-congruent components) have a nice semantic property on which the new program-integration algorithm relies: We are able to show that sequence-congruent components—possibly in different programs—have equivalent execution behavior. This is stated as the Sequence-Congruence Theorem. Thus, the Theorem relates the partitioning operation to the execution behaviors of program components and establishes the ability of the Sequence-Congruence Algorithm to detect program components with equivalent behavior. In this section, we give a proof of the Sequence-Congruence Theorem.

Since the theorem concerns components' run-time behaviors, we define explicitly the notion of a moment immediately before (or after) an execution step. Assignment statements, predicates, and ϕ assignments of a program P are executed in the order specified by the augmented control flow graph of P. The execution of an assignment statement or a ϕ assignment and the evaluation of a predicate constitute a step of program execution. A *moment* immediately before (or after) the execution of a component u denotes the time when u is about to start executing (or, respectively, has just finished). There is a subtle distinction between a moment immediately after the execution of a vertex and a moment immediately before the execution of the following one. For instance, consider the following program fragment:

The Sequence-Congruence Algorithm:

Pass 1: Add a *flow-if* edge from every *if* predicate to each associated ϕ_{if} vertex.
Add a *flow-while* edge from every *while* predicate to each associated ϕ_{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.
Remove 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 4-2. The Sequence-Congruence Algorithm. The Sequence-Congruence Algorithm consists of two passes. Both passes use the basic partitioning algorithm in Figure 4-1; only the starting partition and the edges considered in the two passes are different.

```
program A                          program B
   x := 1                             if p
   if p                                  then u := 2
      then x := 2                        else u := 1
   fi                                 fi
   φ_if: x := x                       φ_if: u := u
   y := x                             v := u
   z := y + 3                         w := w + 3
end(z)                             end(w)
```

The sequence-congruence classes of program components of the two programs:

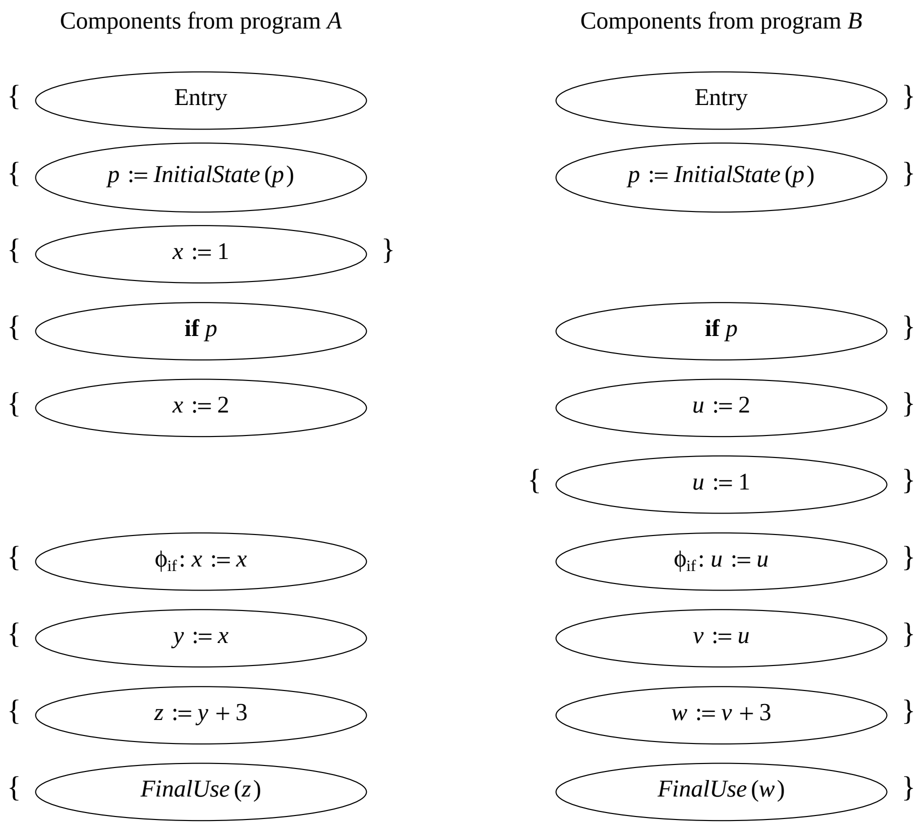


Figure 4-3. The sequence-congruence classes of program components of programs *A* and *B*. The two assignments $x := 1$ and $u := 1$ are in the same initial class and they stay in the same class after the first partitioning pass; however, they are separated during the second pass. Thus, the two components are data-congruent but not sequence-congruent. The two *FinalUse* vertices are sequence-congruent even though they are associated with different variables.

```
<T1>    x := 0
<T2>    while p do
           . . .
        od
```

Suppose there are no ϕ_{enter} vertices for the *while* loop. The locus of control is *outside* the loop at the moment immediately after the execution of the assignment at T1, whereas the locus of control moves *inside* the loop at the moment immediately before the execution of the *while* predicate T2. Note also that the locus of control moves outside the loop at the moment immediately after the predicate T2 evaluates to *false*.

It is important to identify the loops that are executing at a moment during program execution. A loop L is executing at a moment t if the locus of control at t is inside L. The *current loop predicate* at a moment t, written as CLP(t), is the predicate of the innermost loop that is executing at t. If there is no such loop, CLP(t) is the *Entry* vertex. In particular, if t is the moment immediately before (or after) executing a ϕ_{enter} statement, the locus of control is inside the loop of the ϕ_{enter} statement; hence CLP(t) is the predicate of the associated loop.

A loop *encloses* a vertex v (or, equivalently, *v is enclosed in* the loop) if there is a control-dependence path from the predicate of the loop to v.

Since loops may be executed repeatedly, we distinguish *executions* and *iterations* of a loop. During program execution, there may be several *executions* of a loop; during each execution of the loop, there may be one or more *iterations*. There is at least one iteration during an execution of a loop:[12] the ϕ_{enter} vertices and the loop predicate must be executed at least once during an execution of the loop.

Since a vertex enclosed in a loop may be executed repeatedly, a vertex is *active* at a moment t (defined below) if the "appropriate" value produced at the vertex is available for use at t [Alpern88].

Definition. A vertex u in a program representation graph is *active* at a moment t during program execution if (1) u is not enclosed in a loop and has already been executed at t, or (2) the innermost loop that encloses u is executing at t and u has been executed during the current iteration.

According to the definition, ϕ_{enter} vertices and *while* predicates are active only when the locus of control is inside their loops.

In order to compare execution behaviors of components that may belong to *different* programs, it is necessary to relate two moments, t_1 and t_2, during the respective executions of the two programs. We say t_1 and t_2 are *concurrent* if the executions of P_1 and P_2 at t_1 and t_2 are synchronized in the sense defined below.

Definition. Let P_1 and P_2 be two programs. Let t_1 and t_2 be two moments during the executions of P_1 and P_2, respectively. t_1 and t_2 are *concurrent* if (1) CLP(t_1) and CLP(t_2) are at the same loop nesting level, (2) corresponding *while* predicates on the control-dependence paths from *Entry* to CLP(t_1) and CLP(t_2), respectively, in the program representation graphs of P_1 and P_2 are data-congruent, and (3) corresponding *while* predicates have executed the same number of iterations during the current executions of the loops at t_1 and t_2, respectively.

---

[12]The number of iterations during an execution of a loop defined here differs from the traditional point of view: With our definition, the iteration count is one greater than normal. This convention makes the statement of the proof easier; it does not carry any semantic significance.

Note that *if* predicates are ignored in the above definition. For instance, in Figure 4-3, if t_1 is the moment immediately after the statement "$x := 1$" executes and t_2 is the moment immediately after the statement "$u := 1$" executes, t_1 and t_2 are concurrent.

Since sequence-congruence is a refinement of data-congruence, the Sequence-Congruence Theorem (for sequence-congruence classes) is founded on the Data-Congruence Lemma (for data-congruence classes), which states that active, data-congruent vertices have the same values[13] at concurrent moments when the programs run on sufficiently similar initial states.

Lemma. (Data-Congruence Lemma). *Let P_1 and P_2 be two programs with imported variables Imp_1 and Imp_2, respectively. Let σ_1 and σ_2 be two states that agree on $(Imp_1 \cap Imp_2)$. Let t_1 and t_2 be two moments during the executions of P_1 and P_2 on initial states σ_1 and σ_2, respectively. Let x_1 and x_2 be two vertices in P_1 and P_2, respectively. If (1) t_1 and t_2 are concurrent, (2) x_1 is active at t_1, (3) x_2 is active at t_2, and (4) x_1 and x_2 are data-congruent, then x_1 and x_2 have the same values at t_1 and t_2, respectively.*

Proof. We prove this lemma by contradiction. In particular, we argue by considering the earliest counterexample.

Suppose the lemma is not correct; then there exist x_1, x_2, t_1, and t_2 that satisfy (1), (2), (3), and (4) above but x_1 and x_2 have different values at t_1 and t_2, respectively. Let t_1 be the earliest moment during the execution of P_1 on initial state σ_1 such that there is a moment t_2 during the execution of P_2 on initial state σ_2 and there are two vertices x_1 and x_2 of P_1 and P_2, respectively, such that (1), (2), (3), and (4) hold but x_1 and x_2 have different values at t_1 and t_2, respectively. It is possible that, for a given t_1, there are many t_2, x_1, and x_2 that fit the above conditions. In this case, the ones with the earliest t_2 are chosen. It is also possible that, given t_1 and t_2, there are many x_1 and x_2 that fit the above conditions. In this case, the earliest x_1 (in terms of appearance in the augmented control flow graph of P_1) in P_1 is chosen. It is also possible that, given t_1, t_2, x_1, there are many x_2 that fit the above conditions. In this case, the earliest x_2 in P_2 is chosen.

Since x_1 and x_2 are data-congruent, they must either be ϕ statements of the same kind or they must have the same operators. Hence, they have the same incoming flow dependence edges. There are five cases depending on the type of vertex x_1. We will derive a contradiction in each case.

Case 1. Vertex x_1 is a *FinalUse* vertex, a non-ϕ assignment statement vertex, or a predicate vertex. If x_1 is a constant vertex (that is, the expression in x_1 is a constant), so is x_2 and they must be the same constant. In this case, x_1 and x_2 always have same values whenever they are active. So assume x_1 and x_2 are not constant vertices.

Since x_1 and x_2 have the same number of incoming flow dependence edges, let y_1 and y_2 be any analogous flow predecessors of x_1 and x_2, respectively. Since x_1 and x_2 are data-congruent, y_1 and y_2 are also data-congruent. Because there is a flow edge $y_1 \longrightarrow_f x_1$ and x_1 is not a ϕ_{exit} vertex, any loop enclosing y_1 must also enclose x_1 (due to the ϕ_{exit} vertices in the graph, y_1 cannot be nested more deeply than x_1). Because x_1 is active at t_1, the innermost loop enclosing y_1, if any, must be executing at t_1 and x_1 must have been executed during the current iteration of that loop.

---

[13]The value of a vertex at a moment is the most recent value produced at the vertex at that moment. There is a subtle point here: if vertex v assigns to variable a and v is active at moment t, the value of v and that of a in the *current* state at t are not necessarily equal because a might be assigned another value after v is executed.

Note that flow dependence edges incident on any non-ϕ_{enter} vertex run from left to right. Because x_1 has been executed, y_1 must have already been executed during the current iteration of the innermost loop enclosing y_1 (if any); therefore, y_1 is active at t_1. Similarly, y_2 is active at t_2.

Note that y_1 and y_2 come before x_1 and x_2 in P_1 and P_2, respectively. Thus, y_1 and y_2 must have the same values at t_1 and t_2, respectively, for otherwise we would have chosen y_1 and y_2 instead of x_1 and x_2.

Let a_1 and a_2 be the variables that are assigned a value at y_1 and y_2, respectively. Since y_1 and y_2 are analogous flow predecessors of x_1 and x_2, respectively, a_1 and a_2 are analogous operands of x_1 and x_2, respectively.

Note that variable a_1 has the same value as vertex y_1 immediately before the execution of x_1 at time t_1; that is, a_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties of program representation graphs, x_1 and x_2 have exactly one incoming flow dependence edge for each operand of x_1 and x_2). Similarly, variable a_2 has the same value as vertex y_2 immediately before the execution of x_2 at time t_2. Since y_1 and y_2 have the same values at t_1 and t_2, respectively, a_1 and a_2 must have the same values immediately before the executions of x_1 and x_2 at t_1 and t_2, respectively.

Because analogous operands of x_1 and x_2 have the same values and because x_1 and x_2 have the same operator, x_1 and x_2 must evaluate to the same values at times t_1 and t_2, respectively, which contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

Case 2. Vertex x_1 is a ϕ_{if} vertex. Let z_1 and z_2 be the *if* predicates for x_1 and x_2, respectively. Since x_1 and x_2 are data-congruent, z_1 and z_2 are also data-congruent. Because x_1 is active at t_1, z_1 is also active at t_1. Similarly, z_2 is active at t_2. Note that z_1 and z_2 come before x_1 and x_2 in P_1 and P_2, respectively. Thus, z_1 and z_2 must have the same values at t_1 and t_2, respectively, for otherwise we would have chosen z_1 and z_2 instead of x_1 and x_2. Without loss of generality, assume the values of z_1 and z_2 at t_1 and t_2, respectively, are *true*.

Let $y_1 \longrightarrow_f x_1$ and $y_2 \longrightarrow_f x_2$ be the incoming flow dependence edges of x_1 and x_2 from (or around) the *true* branches of z_1 and z_2, respectively. Because there is a flow edge $y_1 \longrightarrow_f x_1$ and x_1 is active at t_1, by the same arguments as in Case 1, y_1 is active at t_1. Similarly, y_2 is active at t_2. Because x_1 and x_2 are data-congruent, y_1 and y_2 are also data-congruent. Thus, y_1 and y_2 must have the same values at t_1 and t_2, respectively, for otherwise we would have chosen y_1 and y_2 instead of x_1 and x_2.

Note that at time t_1, x_1 has the same value as y_1; that is, the variable that is assigned a value at y_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties of program representation graphs, a ϕ_{if} vertex has exactly one incoming flow dependence edge from or around the *true* branch). Similarly, at time t_2, x_2 has the same value as y_2.

Since y_1 and y_2 have the same values at t_1 and t_2, x_1 and x_2 must have the same values at t_1 and t_2, respectively, which contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

Case 3. Vertex x_1 is a ϕ_{enter} vertex. Let z_1 and z_2 be the *while* predicates associated with x_1 and x_2, respectively. Note that a ϕ_{enter} vertex is active only when the locus of control is inside the associated loop. Because t_1 and t_2 are concurrent, corresponding *while* predicates on the control-dependence paths from *Entry* to CLP(t_1) and CLP(t_2) are data-congruent and have executed the same number of iterations during the current executions of the loops at moments t_1 and t_2, respectively. Since x_1 and x_2 are at the same nesting level, z_1 and z_2 are corresponding *while* predicates on the control-dependence paths from *Entry* to CLP(t_1)

and CLP(t_2). Hence z_1 and z_2 are data-congruent and have executed the same number of iterations during the current executions of the loops, at moments t_1 and t_2, respectively.

(1) Suppose, at t_1, it is the first iteration of the loop of z_1 during the current execution of the loop. It is also the first iteration of the loop of z_2 during the current execution of the loop at t_2. Therefore, the values of x_1 and x_2 at t_1 and t_2 come from outside the loops of z_1 and z_2, respectively.

Let y_1 and y_2 be the flow predecessors of x_1 and x_2 from outside the loops of z_1 and z_2, respectively. Since x_1 and x_2 are data-congruent, y_1 and y_2 are also data-congruent. Since x_1 and x_2 are active at t_1 and t_2, respectively, y_1 and y_2 are also active at t_1 and t_2, respectively. Thus, y_1 and y_2 must have the same values at t_1 and t_2, respectively, for otherwise we would have chosen y_1 and y_2 instead of x_1 and x_2.

Note that at time t_1, x_1 has the same value as y_1; that is, the variable that is assigned a value at y_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties of program representation graphs, a ϕ_{enter} vertex has exactly one incoming flow dependence edge from outside the loop). Similarly, at time t_2, x_2 has the same value as y_2.

Since y_1 and y_2 have the same values at t_1 and t_2, x_1 and x_2 must have the same values at t_1 and t_2, respectively, which contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

(2) Suppose, at t_1, it is the k^{th} iteration of the loop of z_1 during the current execution of the loop, for some $k > 1$. It is also the k^{th} iteration of the loop of z_2 during the current execution of the loop at t_2. Therefore, the values of x_1 and x_2 at t_1 and t_2 come from inside the loops of z_1 and z_2, respectively (*i.e.*, the values are produced during the $k - 1^{st}$ iterations).

Let y_1 and y_2 be the flow predecessors of x_1 and x_2 from inside the loops of z_1 and z_2, respectively. Since x_1 and x_2 are data-congruent, y_1 and y_2 are also data-congruent. Let t_1' be the moment immediately before the end of the $k-1^{st}$ iteration of the loop of z_1 and t_2' be the moment immediately before the end of the $k-1^{st}$ iteration of the loop of z_2. Note that y_1 and y_2 are active at t_1' and t_2', respectively. Note also that t_1' and t_2' are earlier than t_1 and t_2, respectively, and t_1' and t_2' are concurrent. Thus, y_1 and y_2 must have the same values at t_1' and t_2', respectively, for otherwise we would have chosen t_1', t_2', y_1, and y_2 instead of t_1, t_2, x_1, and x_2.

Note that the value of x_1 at t_1 is the same as that of y_1 at t_1'; that is, the variable that is assigned a value at y_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties program representation graphs, a ϕ_{enter} vertex has exactly one incoming flow dependence edge from inside the loop). Similarly, the value of x_2 at t_2 is the same as that of y_2 at t_2'.

Because the value of x_1 at t_1 is the same as that of y_1 at t_1' and the value of x_2 at t_2 is the same as that of y_2 at t_2' and because the value of y_1 at t_1' is the same as that of y_2 at t_2', x_1 and x_2 must have the same values at t_1 and t_2, respectively, which contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

Case 4. Vertex x_1 is a ϕ_{exit} vertex. Let z_1 and z_2 be the *while* predicates associated with x_1 and x_2, respectively. Let y_1 and y_2 be the ϕ_{enter} vertices associated with x_1 and x_2, respectively. Since x_1 and x_2 are data-congruent, z_1 and z_2 are data-congruent and y_1 and y_2 are data-congruent. Because y_1 and y_2 are data-congruent, the respective loops of z_1 and z_2 are at the same nesting level.

Because x_1 is a ϕ_{exit} vertex of the loop of z_1 and it is active at t_1, the most recent execution of the loop of z_1 must have finished. Similarly, since x_2 is active at t_2, the most recent execution of the loop of z_2 must have finished. Let n_1 be the number of iterations the most recent execution of the loop of z_1 iterated. Let n_2 be the number of iterations the most recent execution of the loop of z_2 iterated.

We first show that $n_1 = n_2$. Suppose $n_1 \neq n_2$. First assume $n_1 < n_2$. Let s_1 be the moment immediately before the ${n_1}^{th}$ evaluation of z_1 during the most recent execution of the loop of z_1. Let s_2 be the moment immediately before the ${n_1}^{th}$ evaluation of z_2 during the most recent execution of the loop of z_2. Note that the loops of z_1 and z_2 are executing the ${n_1}^{th}$ iteration during the most recent executions at s_1 and s_2, respectively. Therefore, s_1 and s_2 are concurrent. Since the ${n_1}^{th}$ value produced at z_1 is *false* but the ${n_1}^{th}$ value produced at z_2 is *true*, at least one pair of analogous operands of z_1 and z_2 must have different values at the two moments s_1 and s_2, respectively.

However, because s_1 and s_2 are concurrent and analogous operands of z_1 and z_2 are data-congruent and are active at s_1 and s_2, respectively, analogous operands must have the *same* values at s_1 and s_2, respectively, for otherwise we would have chosen s_1 and s_2 instead of t_1 and t_2. Because analogous operands of z_1 and z_2 have the same values at s_1 and s_2, respectively, the ${n_1}^{th}$ values produced at z_1 and z_2, respectively, must be the same, which contradicts the assumption that $n_1 < n_2$. Hence $n_1 \geq n_2$. By the same argument we know $n_2 \geq n_1$. Therefore, $n_1 = n_2$. (Let n be n_1 or, equivalently, n_2.)

Recall that y_1 and y_2 are the ϕ_{enter} vertices associated with x_1 and x_2, respectively. Let ${t_1}'$ be the moment immediately before the n^{th} evaluation of z_1 during the most recent execution of the loop of z_1. Let ${t_2}'$ be the moment immediately before the n^{th} evaluation of z_2 during the most recent execution of the loop of z_2. Note that the loops of z_1 and z_2 are executing the n^{th} iteration during the most recent executions at ${t_1}'$ and ${t_2}'$, respectively. Therefore, ${t_1}'$ and ${t_2}'$ are concurrent. Since y_1 and y_2 are data-congruent, y_1 is active at ${t_1}'$, y_2 is active at ${t_2}'$, and ${t_1}'$ and ${t_2}'$ are concurrent, y_1 and y_2 must have the same values at ${t_1}'$ and ${t_2}'$, respectively, for otherwise we would have chosen ${t_1}'$, ${t_2}'$, y_1, and y_2 instead of t_1, t_2, x_1, and x_2.

Note that the value of x_1 at t_1 is the same as that of y_1 at ${t_1}'$; that is, the variable that is assigned a value at y_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties of program representation graphs, a ϕ_{exit} vertex has exactly one incoming flow dependence edge, whose source is the associated ϕ_{enter} vertex). Similarly, the value of x_2 at t_2 is the same as that of y_2 at ${t_2}'$.

Because the value of x_1 at t_1 is the same as that of y_1 at ${t_1}'$ and the value of x_2 at t_2 is the same as that of y_2 at ${t_2}'$ and because the value of y_1 at ${t_1}'$ is the same as that of y_2 at ${t_2}'$, x_1 and x_2 must have the same values at t_1 and t_2, respectively, which contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

Case 5. Vertex x_1 is an *InitialState* vertex. Since x_1 and x_2 are data-congruent, they must be the *Initial-State* vertices for the same variable. Since x_1 and x_2 are not in any loops, they are executed exactly once. Because σ_1 and σ_2 agree on $Imp_1 \cap Imp_2$, x_1 and x_2 must have the same values whenever they are active. This contradicts the previous assumption that x_1, x_2, t_1, and t_2 violate the lemma.

We have shown that each of the five cases leads to a contradiction. Therefore, it is impossible to find t_1, t_2, x_1, and x_2 such that (1), (2), (3), and (4) are satisfied but x_1 and x_2 have different values at t_1 and t_2. □

The following theorem asserts that sequence-congruent program components produce similar or identical sequence of values when their programs run on sufficiently similar initial states. This theorem is stated in a slightly more general form than what is required in the definition of *equivalent behavior* given in Section 4.1

in that the two programs need only be run on sufficiently similar, but not identical, initial states.

Theorem. *Let P_1 and P_2 be two programs with imported variables Imp_1 and Imp_2, respectively. Let σ_1 and σ_2 be two states that agree on $(Imp_1 \cap Imp_2)$. Let x_1 and x_2 be two vertices in P_1 and P_2, respectively, that are sequence-congruent. Then*
(1) If P_1 and P_2 terminate normally on σ_1 and σ_2, respectively, then the sequences of values produced at x_1 and x_2, respectively, are the same.
(2) If P_1 terminates normally on σ_1 but P_2 does not terminate normally on σ_2, then the sequence of values produced at x_2 is a prefix of the sequence of values produced at x_1.
(3) If P_1 does not terminate normally on σ_1 but P_2 terminates normally on σ_2, then the sequence of values produced at x_1 is a prefix of the sequence of values produced at x_2.
(4) If neither P_1 nor P_2 terminates normally on σ_1 and σ_2, respectively, then either (a) the sequences of values produced at x_1 and x_2, respectively, are identical infinite sequences, or (b) the sequence of values produced at x_1 is finite and is a prefix of the sequence of values produced at x_2, or vice versa.

Proof. We prove the four assertions separately; in each case, we prove the assertion by contradiction.

Case 1. Suppose P_1 and P_2 terminate normally on σ_1 and σ_2, respectively. If there exist two sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, such that the sequences of values produced at x_1 and x_2 are different, then either (1) there is a constant k such that the k^{th} values produced at x_1 and x_2 are different, or (2) the sequences of values produced at x_1 and x_2 are of different lengths and the shorter sequence is a prefix of the longer one.

In the latter case, we may examine the sequences of values produced at corresponding control ancestors of x_1 and x_2. Note that corresponding control ancestors of x_1 and x_2 are sequence-congruent. Since the sequences of values produced at x_1 and x_2 are of different lengths and the shorter sequence is a prefix of the longer one, it is impossible that each pair of corresponding control ancestors of x_1 and x_2 have produced the same sequence of values. Thus, there must be two corresponding control ancestors, x_1' and x_2', of x_1 and x_2, respectively, and a constant k such that the k^{th} value produced at x_1' and x_2' are different.

We conclude that if the first assertion of the Theorem is not correct, then we can always find sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, and a constant k such that the k^{th} values produced at x_1 and x_2 are different. Next, we will show this leads to a contradiction.

We argue by considering the earliest counterexample: There may be many x_1, x_2, and k that satisfy the above condition. In this case, the ones with the earliest t_1 are chosen where t_1 is the moment immediately after the k^{th} value of x_1 is produced. In case there are still many x_1, x_2, and k with the same t_1 that satisfy the above condition, the ones with the earliest t_2 are chosen where t_2 is the moment immediately after the k^{th} value of x_2 is produced.

Because a *while* predicate may not be active immediately after it is executed but all other vertices (that is, assignments, *if* predicates, *InitialState*, *FinalUse*, and ϕ vertices) are always active immediately after execution, we need to distinguish different cases depending on whether x_1 and x_2 are *while* predicates.

Due to the control dependence edges that form self-loops on *while* predicate vertices, a *while* predicate can only be sequence-congruent to other *while* predicates. Since x_1 and x_2 are sequence-congruent, either both x_1 and x_2 are *while* predicates or neither is a *while* predicate. Thus, there are two cases to consider. We will derive a contradiction in each case.

(1) Suppose neither x_1 nor x_2 is a *while* predicate. Because x_1 and x_2 are sequence-congruent, CLP(t_1) and CLP(t_2) and each pair of their corresponding control ancestors are sequence-congruent. In particu-

lar, CLP(t_1) and CLP(t_2) are at the same loop nesting levels. Because t_1 and t_2 are the earliest moments when the first assertion fails, CLP(t_1) and CLP(t_2) and each pair of their corresponding control ancestors must have produced the same sequences of values during the executions of P_1 and P_2 from the beginning to t_1 and t_2, respectively. In particular, all corresponding *while* predicates have executed the same number of iterations during their *current* executions. Therefore, t_1 and t_2 are concurrent.

Because x_1 and x_2 are sequence-congruent, they are also data-congruent. Because x_1 and x_2 are not *while* predicates, x_1 and x_2 are always active immediately after they are executed. That is, x_1 is active at t_1 and x_2 is active at t_2. Thus, from the Data-Congruence Lemma, x_1 and x_2 have the same value at t_1 and t_2, respectively, which contradicts the assumption that the k^{th} values of x_1 and x_2 (at t_1 and t_2, respectively) are different.

(2) Suppose x_1 and x_2 are *while* predicates. Let t_1' and t_2' be the moments immediately before the k^{th} evaluations of x_1 and x_2, respectively. That is, t_1' and t_2' are the moments just one step earlier than t_1 and t_2, respectively. Note that CLP(t_1') is x_1 and CLP(t_2') is x_2. Because x_1 and x_2 are sequence-congruent, each pair of corresponding control ancestors of x_1 and x_2 are sequence-congruent. In particular, x_1 and x_2 are at the same loop nesting levels. Because the theorem does not fail until moments t_1 and t_2, and because moments t_1' and t_2' are earlier than t_1 and t_2, respectively, we know that x_1 and x_2 and each pair of their corresponding control ancestors must have produced the same sequence of values from the beginning to t_1' and t_2', respectively. In particular, all corresponding *while* predicates have executed the same number of iterations during their *current* executions. Therefore, t_1' and t_2' are concurrent.

Let y_1 and y_2 be any analogous flow predecessors of x_1 and x_2, respectively. Since x_1 and x_2 are sequence-congruent, y_1 and y_2 are data-congruent. Furthermore, y_1 is active at t_1' and y_2 is active at t_2'. From the Data-Congruence Lemma, y_1 and y_2 have the same value at t_1' and t_2', respectively.

Let a_1 and a_2 be the variables that are assigned a value at y_1 and y_2, respectively. Since y_1 and y_2 are analogous flow predecessors of x_1 and x_2, respectively, a_1 and a_2 are analogous operands of x_1 and x_2, respectively.

Note that at time t_1', variable a_1 has the same value as vertex y_1; that is, a_1 cannot be assinged a value again after the most recent execution of y_1 for otherwise there cannot be the flow dependence edge $y_1 \longrightarrow_f x_1$ (due to the properties of program representation graphs, x_1 and x_2 have exactly one incoming flow dependence edge for each operand of x_1 and x_2). Similarly, at time t_2', variable a_2 has the same value as vertex y_2. Since y_1 and y_2 have the same values at t_1' and t_2', respectively, a_1 and a_2 must have the same values at t_1' and t_2', respectively.

Because analogous operands of x_1 and x_2 have the same values at t_1' and t_2', respectively, and because x_1 and x_2 have the same operators, x_1 and x_2 must evaluate to the same values at t_1 and t_2, respectively, which contradicts the assumption that the k^{th} values of x_1 and x_2 (at t_1 and t_2, respectively) are different.

In both cases we have shown a contradiction. Thus, we have proved the first assertion.

Case 2. Suppose P_1 terminates normally on σ_1 but P_2 does not terminate normally on σ_2. If there exist two sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, such that the sequence of values produced at x_2 is not a prefix of the sequence of values produced at x_1, then either (1) there is a constant k such that the k^{th} values produced at x_1 and x_2 are different, or (2) the sequence of values produced at x_1 is a proper prefix of the sequence of values produced at x_2.

In the latter case, we may examine the sequences of values produced at corresponding control ancestors of x_1 and x_2. Note that corresponding control ancestors of x_1 and x_2 are sequence-congruent. Since the sequence of values produced at x_1 is a proper prefix of the sequence of values produced at x_2, it is impossible that each pair of corresponding control ancestors of x_1 and x_2 have produced the same sequence of values. Thus, there must be two corresponding control ancestors, x_1' and x_2', of x_1 and x_2, respectively, and a constant k such that the k^{th} value produced at x_1' and x_2' are different.

We conclude that if the second assertion of the Theorem is not correct, then we can always find sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, and a constant k such that the k^{th} values produced at x_1 and x_2 are different. However, by the same argument as in *Case 1*, this leads to a contradiction. Thus, we have proved the second assertion.

Case 3. Suppose P_1 does not terminate normally on σ_1 but P_2 terminates normally on σ_2. This case is similar to *Case 2*.

Case 4. Suppose neither P_1 nor P_2 terminates normally on σ_1 and σ_2, respectively. If there exist two sequence-congruent vertices x_1 and x_2 that violate the fourth assertion of the Theorem, then, depending on whether the sequences of values produced at x_1 and x_2 are infinite, there are two cases to consider.

(1) Suppose the sequences of values produced at x_1 and x_2, respectively, are infinite. If the two infinite sequences are not identical, there must be a constant k such that the k^{th} values produced at x_1 and x_2 are different.

(2) Suppose at least one of the sequences of values produced at x_1 and x_2 is finite. If the sequence of values produced at x_1 is not a prefix of the sequence produced at x_2, or *vice versa*, there must be a constant k such that the k^{th} values produced at x_1 and x_2 are different.

We conclude that if the fourth assertion of the Theorem is not correct, then we can always find sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, and a constant k such that the k^{th} values produced at x_1 and x_2 are different. However, by the same argument as in *Case 1*, this leads to a contradiction. Thus, we have proved the fourth assertion. □

The Sequence-Congruence Theorem follows as an immediate corollary to the above theorem. The Theorem states that sequence-congruent program components have equivalent execution behavior.

Theorem. (Sequence-Congruence Theorem). *Sequence-congruent program components have equivalent execution behavior.*

4.4. Enhancements

In this section we consider three simple enhancements to the Sequence-Congruence Algorithm. The first is concerned with simple assignment statements and simple predicates. Due to the property of the *identity* operator in a simple vertex, a simple vertex is, in fact, always data-congruent to its (sole) flow predecessor although this would not be discovered by the Sequence-Congruence Algorithm as defined above. To permit the computation of larger classes of data-congruent vertices, 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. Vertices u and v may or may not be put in different partitions during the second pass. For instance, consider the following example:

<T1> $a := 1$
<T2> $b := a + 2$

<T3> $c := 1$
<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 sequence-congruent. The proofs in the previous section can be directly adapted to account for this change by extending the notion of "analogous" flow predecessors of two vertices to take simple vertices into account.

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 sequence-congruent because they have different operators.

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

We can detect more sequence-congruent components 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 found to be sequence-congruent 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 sequence-congruent because they have different operators (and different number of incoming flow dependence edges).

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

As before, 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 sequence-congruence classes will result from partitioning.

We close this section with an observation about how some additional enhancements to the Sequence-Congruence Algorithm can be made. Although the Sequence-Congruence Algorithm presented above uses the same basic partitioning algorithm—the one given in Figure 4-1—for both Pass 1 and Pass 2, this is not strictly necessary. The proof of the Sequence-Congruence Theorem depends only on the condition that the equivalence classes used at the start of Pass 2 have the properties listed in the Data-Congruence Lemma. Thus, any techniques applied during Pass 1 that result in larger equivalence classes with these properties will not affect the arguments we have given to establish the properties of the equivalence classes computed by Pass 2; the equivalence classes computed by Pass 2 will still have the properties listed in the Sequence-Congruence Theorem.

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 data-congruence classes, and hence T4 and T8 are also placed in separate data-congruence classes. 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.

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 one sequence-congruence class, and T4 and T8 would be members of another.

Chapter 5

A New Program Integration Algorithm
That Accommodates Semantics-Preserving Transformations

Having defined the notion of equivalent execution behavior of program components and having presented the Sequence-Congruence Algorithm, we now present the new program-integration algorithm. Recall that the motivation for the new integration algorithm is that the HPR algorithm cannot accommodate semantics-preserving transformations. The most significant characteristics of the new integration algorithm are (1) it can accommodate semantics-preserving transformations and (2) it is very flexible and extendible in that additional techniques for detecting program components with equivalent behaviors can be easily incorporated in the new integration algorithm. In this chapter, we will describe the new integration algorithm. Chapter 6 proves that the new algorithm satisfies the integration criterion and Chapter 7 shows that the new integration algorithm improves on the integration algorithm of Horwitz, Prins, and Reps.

5.1. The New Integration Algorithm

Like the HPR integration algorithm, the new integration algorithm takes as input a base program and two variants. The first step of the new integration algorithm detects components with equivalent behaviors. This can be done by an equivalence-detection algorithm, such as the Sequence-Congruence Algorithm, or by information gathered from other sources, or both. The result of the first step is a set of equivalence classes of program components with equivalent behaviors. From the equivalence classes obtained from the first step, the new integration algorithm identifies the changed and preserved computations of the variants, extracts the necessary program components and dependences, combines the extracted components and dependences to form a merged graph, removes “useless” ϕ vertices (explained in Section 5.1.5), and produces an integrated program from the merged graph. The new integration algorithm reports that the changes made in the variants interfere if it cannot produce a merged graph or if it cannot produce an integrated program from the merged graph.

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

(1) Detect program components with equivalent behaviors.

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

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

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

(5) Remove useless ϕ vertices from the merged graph.

(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).

5.1.1. Detecting Equivalent Components

The first step of the new integration algorithm is to detect program components, possibly from different programs, with equivalent behaviors. Since this is an undecidable problem, the integration algorithm necessarily employs some *safe* algorithm to detect program components with equivalent behaviors. Any algorithm that can detect program components with equivalent behaviors can serve as the first step of the new integration algorithm. For instance, comparing program slices can be used for this purpose; the Sequence-Congruence Algorithm discussed in Chapter 4 can also be used as the first step of the new integration algorithm.

By using more exact equivalence-detection algorithms, the new integration algorithm can identify the changed and preserved computations more accurately; hence the integrated program produced by the new integration algorithm will be closer to what the programmers expect. The new integration algorithm is actually a *family* of algorithms, parameterized by the equivalence-detection algorithm used. This endows the new integration algorithm with potential and flexibility.

One advantage of the new integration algorithm is that it can easily exploit additional facts about program semantics. Many techniques used in compiler optimization [Allen72, Loveman77, Aho86], such as constant propagation, movement of invariant code, and common subexpression elimination, can be combined with the Sequence-Congruence Algorithm to detect large classes of program components with equivalent behavior. Alpern, Wegman, and Zadeck discussed two such extensions in [Alpern88]. Knowledge of semantics-preserving program transformations that have been applied to a program or certain parts of the program (either manually or by a transformation system) would also be helpful in detecting larger equivalence classes.

In the following discussion of the new integration algorithm, we do not assume any particular equivalence-detection algorithm. All we need is an algorithm that can detect program components with equivalent behaviors.

5.1.2. Classifying Components

There are two kinds of changes that can be introduced by a variant program: a change in a component's execution behavior, or a change in a component's text that does not affect its execution behavior. The new integration algorithm attempts to preserve both kinds of changes in the integrated program. The vertices in each of the PRGs 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 equivalence classes produced in Step (1) cannot always provide an answer, since one equivalence class may include several vertices from each program (*i.e.*, the partition does not define a one-to-one correspondence). As in Section 2.1.3, we assume that there is a unique-naming mechanism so that program components are identified consistently in all three versions—program components have tags.

First, we assume that components are tagged uniquely within a program. Tags may be provided by the editor[14] used to create *A* and *B* from *Base*, or may be generated by some other mechanism—the source of the tags is not relevant to the algorithm itself.

Next we define that two components c_1 and c_2 are *comparable* components if and only if all of the following hold:

(1) c_1 and c_2 have equivalent behaviors;

(2) c_1 and c_2 are the same kinds of vertices, *i.e.*, they both are *Entry, FinalUse, if* predicates, *while* predicates, assignment statements, ϕ_{if}, ϕ_{enter}, or ϕ_{exit} statements (this implies that c_1 and c_2 have the same number of incoming control dependence edges);

(3) corresponding incoming control dependence edges of c_1 and c_2 have the same *true* or *false* labels;

(4) analogous control predecessors of c_1 and c_2 are comparable components. (Analogous control predecessors are defined in Chapter 4.)

Note that all *Entry* vertices, which have no control predecessors, are always comparable components according to this definition.

Given tags for the components, the correspondence between components of the three programs is established as follows: Two components c_1 and c_2 *correspond* (or c_1 and c_2 are *corresponding* components) if and only if all of the following hold:

(1) c_1 and c_2 are comparable components;

(2) c_1 and c_2 have the same tag;

(3) if c_1 and c_2 are assignment statements, they assign to the same variable.[15]

Corresponding components are considered to be the same components in different programs. That is, we can assign to each component an *identity* so that two components correspond if and only if they have the same identity; hence corresponding components are considered to be the same component occurring in different versions of a program.

Note that corresponding vertices may have different text. For assignment statements that correspond, we only require that they assign to the same variables; the expression part may still differ.

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

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

(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 a context that has changed their execution behaviors.

(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.

[14] Since ϕ statements are not part of the source program, they cannot be tagged by the editor. Their tags can, however, be generated systematically from the tags of the associated predicates and the names of the variables that are assigned to at the ϕ statements.

[15] We require that corresponding components assign to the same variables in order to avoid certain undesirable interferences. This will be explained in Section 5.2.

(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*.

(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, or because the vertex assigns to a different variable 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 whose texts and behaviors are identical in all three programs.

Vertices in *B* are similarly classified into the sets New_B, $Modified_B$, $Modified_A$, *Unchanged*, and $Intermediate_B$. Vertices in *Base* are similarly 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 or whose behavior have been changed in both *A* and *B*.)

Note that it is possible for a vertex in New_A to have a corresponding vertex in *B* that is in New_B and for a vertex in $Modified_A$ to have a corresponding vertex in *B* that is in $Modified_B$. For instance, consider the following three programs (the tags are shown explicitly on the left of each component):

Program *Base*	Variant *A*	Variant *B*
<T1> $x := 0$	<T1> $x := 0$	<T1> $x := 0$
<T2> $y := x$	<T2> $y := 0$	<T2> $y := 0$
<T3> $z := x$	<T4> $x := 1$	<T4> $x := 1$
<T4> $x := 1$	<T3> $z := x$	<T3> $z := 1$

The assignment T3 in *A* is in New_A because the value assigned to *z* at T3 in *A* differs from that assigned to *z* at T3 in *Base*; Similarly, the assignment T3 in *B* is in New_B. However, the two assignment statements T3 in *A* and *B* correspond. The assignment T2 in *A* is in $Modified_A$ because the two assignment statements T2 in *A* and *Base* produce the same value, have the same tag, and they assign to the same variable *y* but their texts differ. Similarly, the assignment T2 in *B* is in $Modified_B$. The assignment T2 in *Base* is in both $Modified_A$ and $Modified_B$. The three assignment statements T2 in *A*, *B*, and *Base* correspond.

The classification process may discover that *A* and *B* interfere with respect to *Base* by identifying corresponding vertices v_A and v_B in *A* and *B*, respectively, such that the text of v_A differs from the text of v_B and, if there is a corresponding vertex v_{Base} in *Base*, the texts of v_A, v_B, and v_{Base} are pairwise unequal. Since a vertex in the merged graph can have only one text, it is not possible to preserve the changed text of this component from both *A* and *B*. This can occur either for a vertex in New_A (with a corresponding vertex in New_B), or for a vertex in $Modified_A$ (with a corresponding vertex in $Modified_B$). In the example given above, the fact that the two assignments tagged T3 in *A* and *B* are corresponding *New* vertices but have different text causes interference.

5.1.3. Computing Changed and Preserved Computations

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

Limited Slices

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

In contrast, when more exact equivalence-detection techniques are used, such as the Sequence-Congruence Algorithm, it is sometimes possible to identify behaviorally equivalent vertices that have unequal program slices. Therefore, an affected component's behavior can be retained in the integrated program without including its entire slice; only a part of the entire slice is needed. Limited slices provide the mechanism for extracting the necessary subgraph of the entire slice.

Definition. Let R be the program representation graph of *Base*, A, or B, and let S be a set of vertices in R. The *limited slice* of R with respect to S, denoted by $R//S$, is 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 $Intermediate_A$ 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. Also note that the limited slice is a subgraph of the entire slice.

Changed and Preserved Computations

The affected components of a variant are the components that have different text than the corresponding components of *Base*, or that have no corresponding component in *Base*. A graph that represents the changed computations of a variant is 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 each vertex u in the set *Unchanged*. Note that these limited slices may not be equal;[16] although u itself has identical text and behavior 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. In summary, for each vertex $u \in Unchanged$, the preserved limited slice with respect to u, $Preserved(u)$, is determined as follows:

---

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

Relationship of limited slices	*Preserved* (u)
$R_A//u = R_B//u$	$R_A//u$ (or $R_B//u$)
($R_A//u = R_{Base}//u$) and ($R_B//u \neq R_{Base}//u$)	$R_B//u$
($R_A//u \neq R_{Base}//u$) and ($R_B//u = R_{Base}//u$)	$R_A//u$
$R_{Base}//u$, $R_A//u$, and $R_B//u$ are pairwise unequal	interference

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

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

5.1.4. 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 graph 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 that have 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*. If vertices from the sets New_A and New_B are corresponding vertices, these vertices must have the *same* text, or else interference would have been reported during vertex classification; if vertices from the sets $Modified_A$ and $Modified_B$ are corresponding vertices, these vertices must have the *same* text, or else interference would have been reported during vertex classification; vertices from the set $Intermediate_A$ cannot have corresponding vertices from *B* (and similarly for vertices from $Intermediate_B$); vertices from the set *Unchanged* have the same text in both *A* and *B*; corresponding ϕ vertices must have the same text.

5.1.5. Removing Useless Pseudo-Assignment Statements

In a program representation graph, there is always a path from every ϕ vertex to a non-ϕ vertex. However, some "useless" ϕ vertices—ones without such paths—can occur in the merged graph created in the previous step due to deletion or modification of code in the variants. For instance, consider the following program fragments.

Program *Base*	Variant *A*	Variant *B*	merged fragment *M*
if P	**if** P	**if** P	**if** P
then $x := 0$	**then** $x := 0$	**then** $x := 0$	**then** $x := 0$
else $x := 1$	**else** $x := 1$	**else** $x := 1$	**else** $x := 1$
ϕ_{if}: $x = x$	ϕ_{if}: $x = x$	ϕ_{if}: $x = x$	ϕ_{if}: $x = x$
$y = x + 2$	$y = x + 2$	$z = x * 3$	
$z = x * 3$			

Variant *A* deleted the assignment to z and variant *B* deleted the assignment to y. Hence in the merged fragment *M*, there are no assignments to y or z. Since the ϕ_{if} statement is an *Unchanged* component, it will be included in *M*. Note the ϕ_{if} statement is useless because it has no flow successors.

If these useless ϕ vertices were not removed, the merged graph would not be the program representation graph of any program and the integration algorithm would report interference just because of these useless ϕ vertices. Since we want to avoid such insignificant interference, all useless ϕ vertices are removed before the integration algorithm attempts to reconstitute a program from the merged graph.

5.1.6. Reconstituting a Program From the Merged Graph

The final step of the new integration algorithm is to determine whether the merged graph is the program representation graph of some program, and if so, to produce the program. If the merged graph is not the program representation graph of any program, the new integration algorithm reports that the two variants interfere.

Determining whether a program dependence graph is feasible has been shown to be NP-complete [Horwitz88a]; a similar result can be shown for program representation graphs. The crux of the problem is to order each predicate's control successors. A backtracking algorithm that operates on program dependence graphs has been written and proved correct [Ball90]; that algorithm can be adapted to work on program representation graphs. Although the algorithm is, in the worst case, exponential in the number of pairs of assignments to the same variable, it is possible to reduce the search space using the techniques described in [Horwitz89, Ball90]. It is our belief that a backtracking method for solving the remaining search will be satisfactory in practice.

Example. Figure 5-1 illustrates the new integration algorithm using the first set of example programs from Figure 1-1, which is shown below.

Program *Base*		Variant *A*		Varian *B*	
	program		**program**		**program**
<T1>	$P := 3.14$	<T1>	$P := 3.14$	<T1>	$PI := 3.14$
<T2>	$rad := 2$	<T2>	$rad := 2$	<T3>	**if** *debug*
<T3>	**if** *debug*	<T3>	**if** *debug*	<T4>	**then** $rad := 4$
<T4>	**then** $rad := 4$	<T4>	**then** $rad := 4$	<T2>	**else** $rad := 2$
	fi		**fi**		**fi**
<T5>	$area := P * (rad ** 2)$	<T5>	$area := P * (rad ** 2)$	<T5>	$area := PI * (rad ** 2)$
<T6>	**end**(*area*)	<T7>	$height := 4$	<T6>	**end**(*area*)
		<T8>	$vol := height * area$		
		<T9>	**end**(*vol*,		
		<T6>	*area*)		

The equivalence classes of program components with equivalent behaviors are determined by the Sequence-Congruence Algorithm. This example shows both textual and behavioral changes made in the variants. The tags of components are shown explicitly on the left of the components. Variant *A* adds two statements to compute the volume of the cylinder. In variant *B*, variable *P* is renamed *PI* and the assignment *rad* := 2 is moved into the *if* statement. The classification of components is shown in Figure 5-1(a); Figure 5-1(b) shows the limited slices of $ChangedComps_A$ and $ChangedComps_B$, that is, the limited slices with respect to vertices in $Affected_A$ and $Affected_B$; Figure 5-1(c) shows the limited slices of *Preserved*. The merged graph in Figure

Classification of components.

	Components of *Base*	Components of *A*	Components of *B*
Unchanged	Entry	Entry	Entry
Unchanged	*debug* :=*InitSt* (*debug*)	*debug* :=*InitSt* (*debug*)	*debug* :=*InitSt* (*debug*)
New_B			*PI* :=3.14
$Intermediate_A$	*P* :=3.14	*P* :=3.14	
$Intermediate_A$	*rad* :=2	*rad* :=2	
Unchanged	**if** *debug*	**if** *debug*	**if** *debug*
Unchanged	*rad* :=4	*rad* :=4	*rad* :=4
New_B			*rad* :=2
Unchanged	ϕ_{if}: *rad* :=*rad*	ϕ_{if}: *rad* :=*rad*	ϕ_{if}: *rad* :=*rad*
$Modified_B$	*area* :=*P** (*rad*** 2)	*area* :=*P** (*rad*** 2)	*area* :=*PI** (*rad*** 2)
New_A		*height* :=4	
New_A		*vol* :=*height***area*	
New_A		*FinalUse* (*vol*)	
Unchanged	*FinalUse* (*area*)	*FinalUse* (*area*)	*FinalUse* (*area*)

Figure 5-1(a). An integration example. Part I. Classification of components. Components on the same row are corresponding vertices.

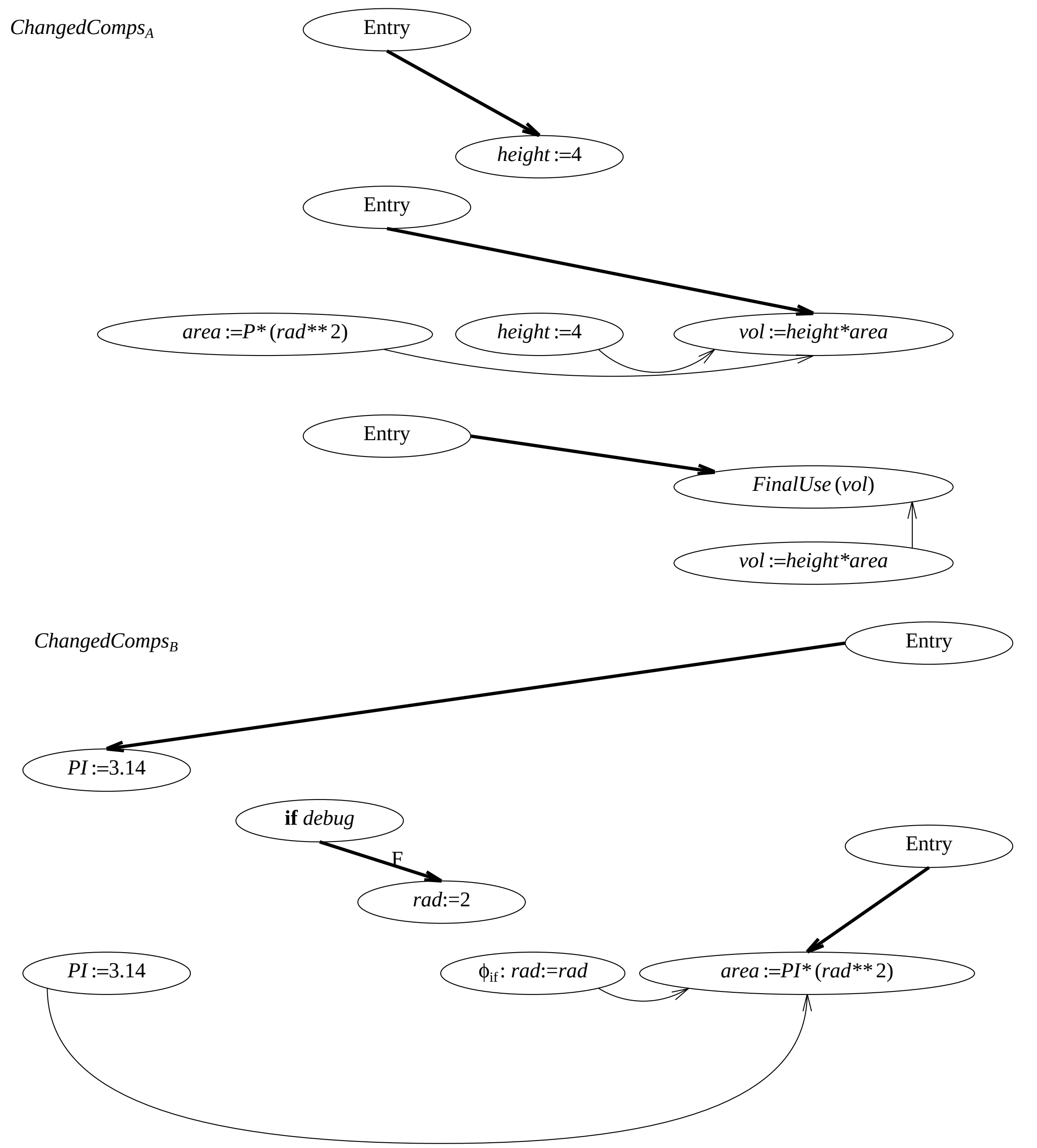


Figure 5-1(b). An integration example. Part II. *ChangedComps*$_A$ and *ChangedComps*$_B$.

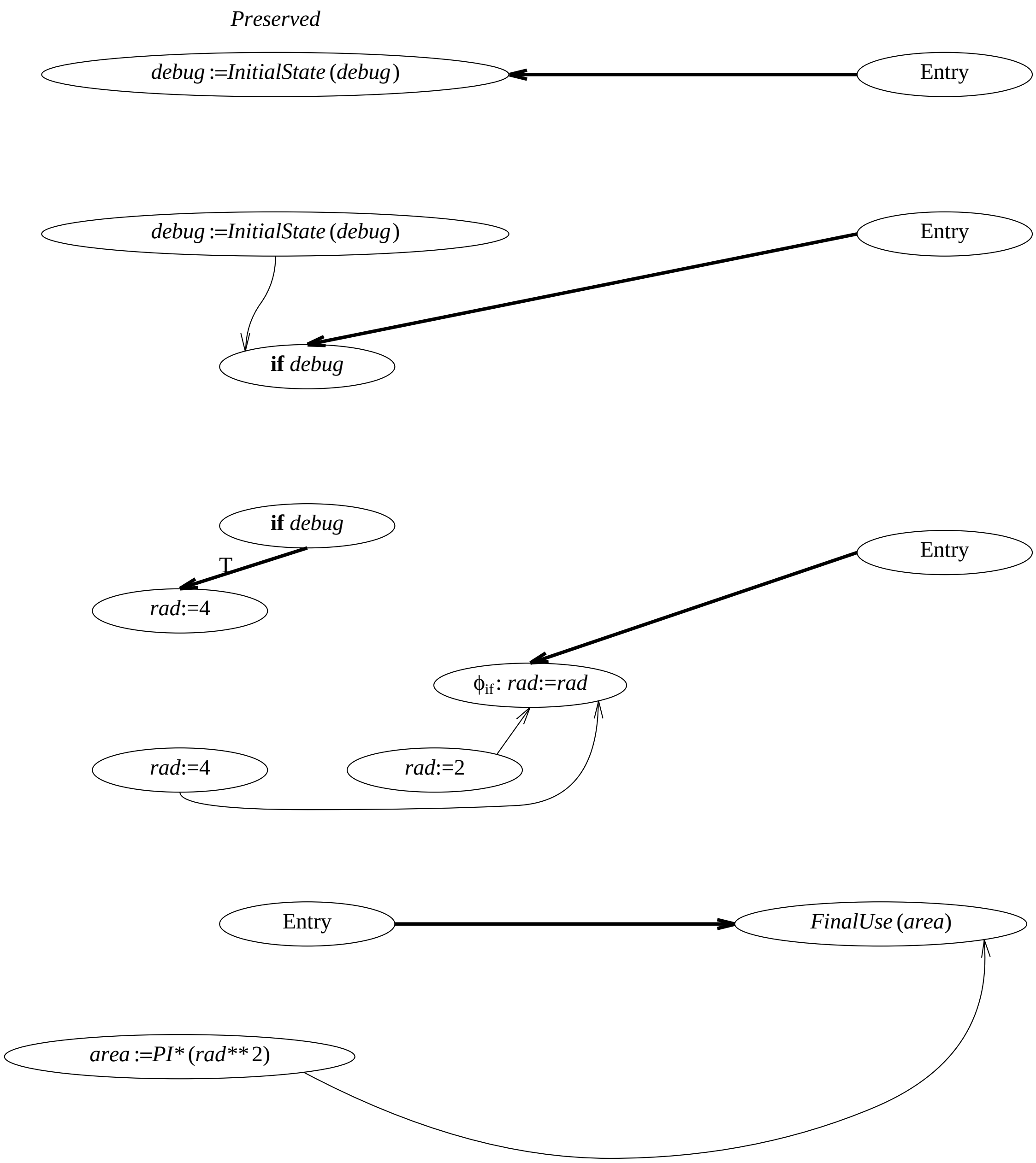
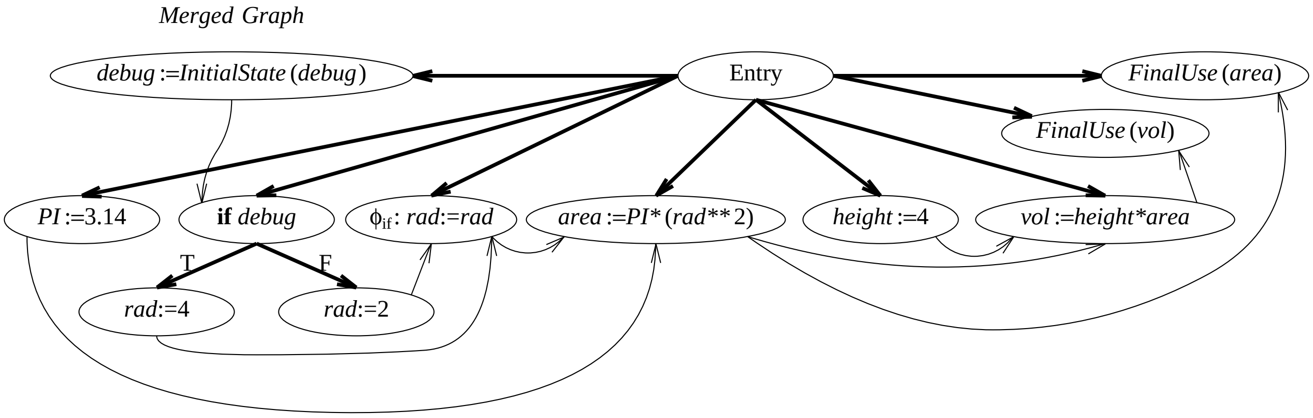


Figure 5-1(c). An integration example. Part III. *Preserved*.



Merged program

program
    PI := 3.14
    if *debug*
        then *rad* := 4
        else *rad* := 2
    fi
    area := *PI* * (*rad* ** 2)
    height := 4
    vol := *height* * *area*
end(*vol*, *area*)

Figure 5-1(d). An integration example. Part IV. The merged graph.

5-1(d) is the union of $ChangedComps_A$, $ChangedComps_B$, and *Preserved*. It is not difficult to see the merged graph is a feasible PRG. One of the programs that correspond to the merged graph is also shown in Figure 5-1(d).

5.2. Discussion of Classification of Vertices

In Section 5.1.2, 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 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> $x := 1$
end	<T2> $y := x + 1$	<T3> $z := u + 2$	<T2> $y := x + 1$	<T2> $y := x + 1$
	end	**end**	<T3> $z := u + 2$	<T1> $u := 1$
			end	<T3> $z := u + 2$
				end

If corresponding assignment statements could assign to different variables, the merged program would be as in $M1$. Note that in $M1$ there is a conflict in the name of the variable that should be used in the statement tagged T1. Because the new integration algorithm requires that corresponding assignment statements assign to the same variable, the merged program produced by the new integration algorithm is as in $M2$. This is because 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 conflict.

Chapter 6

Properties of the New Integration Algorithm

In this chapter, we prove some properties of the new integration algorithm. The properties are summarized in two theorems: the first theorem asserts that the new integration algorithm preserves textual changes of the variants; the second theorem asserts that the new integration algorithm satisfies the semantic criterion of program integration.

Before we state the theorems, we define one new term.

Definition. Two program components are *analogous* if they have the same tag.

Note that corresponding components must be analogous, but not *vice versa*. (It is useful to compare this definition with that of *analogous flow (control) predecessors*, which is given in Section 4.2, and *corresponding components*, which is given in Section 5.2.)

6.1. Preservation of Textual Changes

There are two kinds of changes that can be made in the variants: the behavior of a component may be changed or the text of the component may be changed. The new integration algorithm was designed so that both kinds of changes will be preserved in the merged program in a successful integration.

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 equivalence 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 $Modified_A$ 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 $v_{Base} \in Unchanged$ or $v_A \in New_A$ or $v_B \in New_B$ depending on whether v_{Base}, v_A, and v_B are put in the same equivalence classes in the first step of the new integration algorithm. If $v_{Base} \in Unchanged$, there is a component v in M that has the same text as v_{Base}. If $v_A \in New_A$, there is a component v in M that has the same text as v_A. If $v_B \in New_B$, 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. □

6.2. The Integration Theorem

As with the HPR algorithm, we can prove a theorem for the new integration algorithm about how the execution behavior of the merged program relates to the execution behaviors of the base and the variant programs. The Integration Theorem, which we will prove in this section, asserts that the new integration algorithm satisfies the semantic criterion of program integration discussed in Chapter 1.

Theorem. (Integration Theorem for the New Integration Algorithm). *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}.

Note that this theorem meets (and generalizes) the semantic integration criterion stated in Chapter 1. For example, if there is a variable x whose final value after executing A on σ differs from its final value after executing *Base* on σ, then (1) there is a final-use vertex for variable x in A, and (2) the (sequences of) values produced at the final-use vertices for x in M and in A are identical. Thus, x's final value after executing M on σ is equal to the final value of x after executing A on σ.

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 *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 cases 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 lemmas in this section.

Lemma 6.1. *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 6.2. *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 predecessor, 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 *Entry* to every vertex in the PRG and all *Entry* vertices always produce the same (sequence of) values. If there are many vertices satisfying the above conditions, choose u_M to be the "first" such component (in terms of appearance in program M).

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 two sequences of values are of different length and the shorter one is a prefix of the longer one. However, the latter case cannot happen because u_M's control predecessor, v_M, produces the same sequence of values as v_M's originating vertex and because u_M is the "first" such component in M satisfying the above conditions. Note that there cannot be a non-terminating computation w_M between v_M and u_M for otherwise w_M would have produced a different sequence of values than its originating vertex. If this were the case, because w_M occurs before u_M, we would have chosen 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, 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.

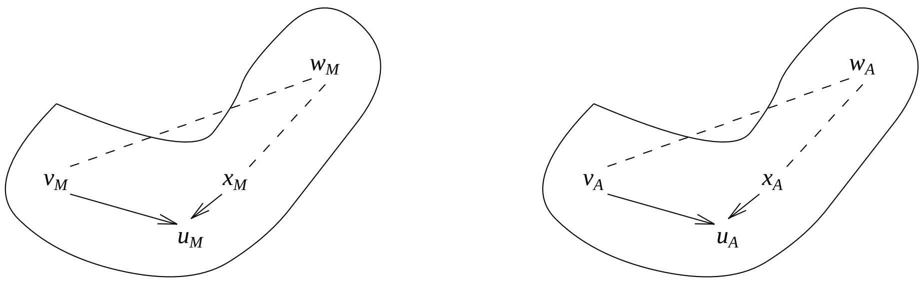
Let v_M be a flow predecessor of u_M. Without loss of generality, assume the flow edge $v_M \longrightarrow_f u_M$ is taken from A; that is, there is a corresponding flow edge $v_A \longrightarrow_f u_A$ in A and v_A and u_A are originating vertices of v_M and u_M, respectively. Let t_A be the moment 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, 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 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 6-1.



M A

Figure 6-1.

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 6.1, 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 6.1, x_M and x_A must have produced identical sequence of values by the times t_M and t_A, respectively.

- 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 6-2.

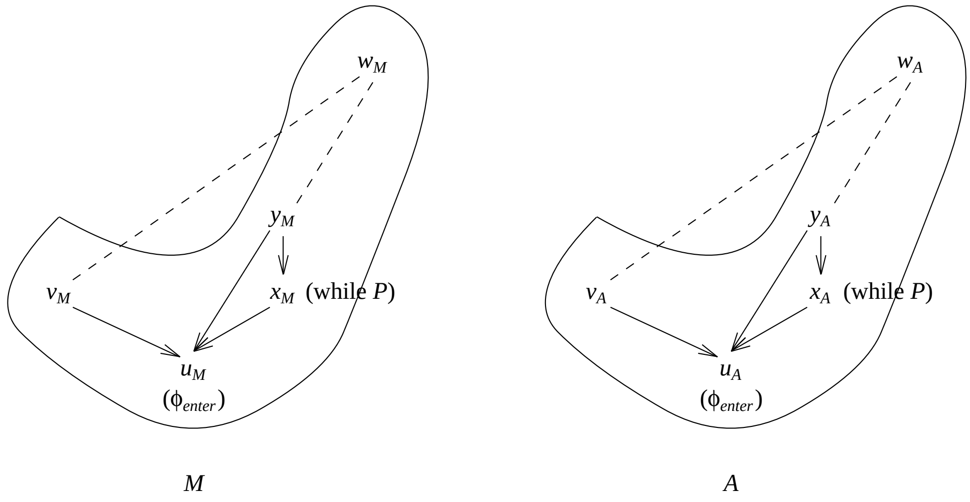


Figure 6-2.

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, 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 6-3.

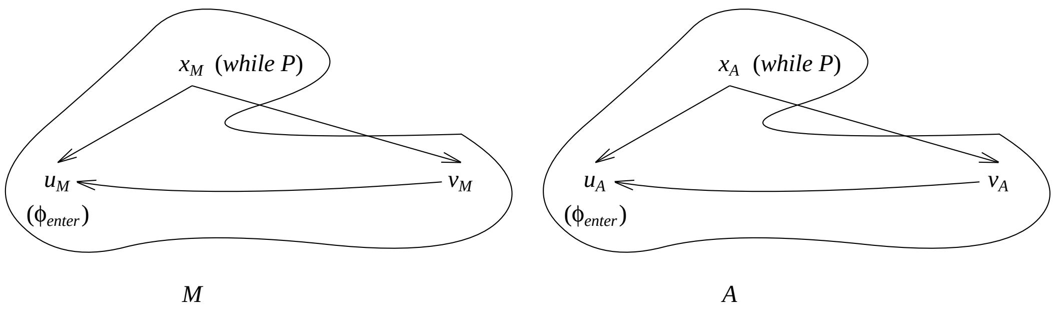


Figure 6-3.

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 6-4.

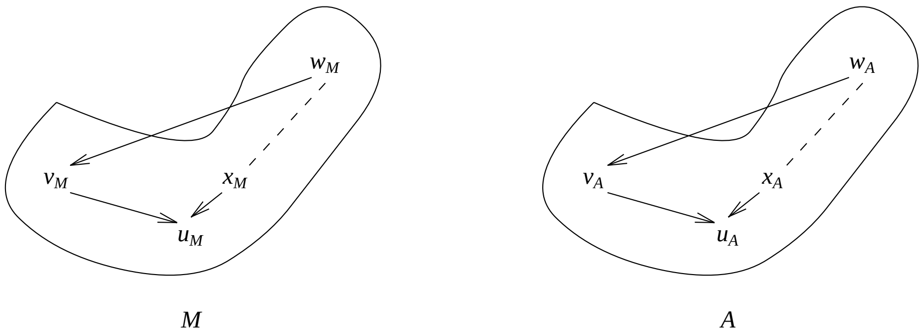


Figure 6-4.

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.

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 6-4. 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 6-5.

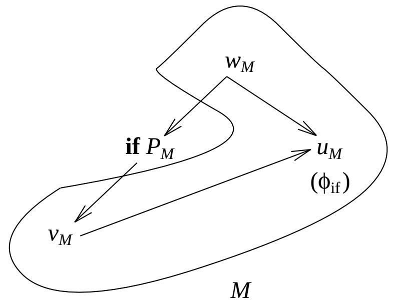


Figure 6-5.

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 corresponding vertices, P_A and P_B are comparable vertices (see the definition in Section 5.1.2). 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 6-6.

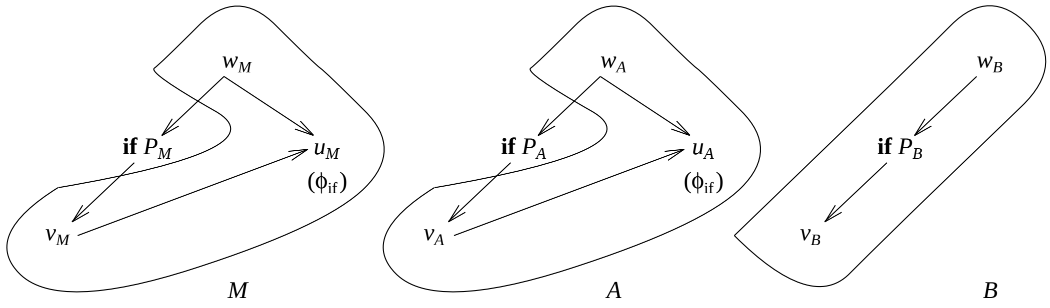


Figure 6-6.

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 comparable 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.

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 6-7.

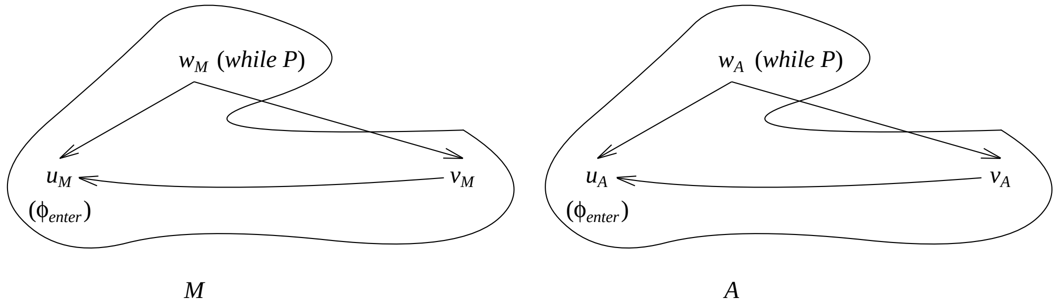


Figure 6-7.

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 6-8.

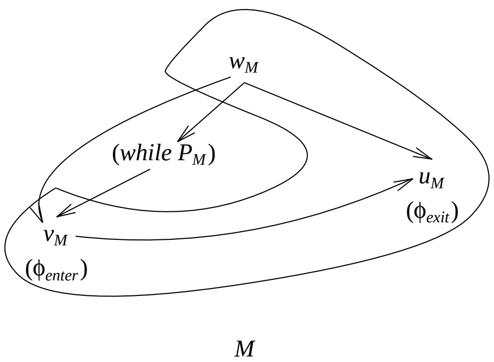


Figure 6-8.

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.

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 6-9.

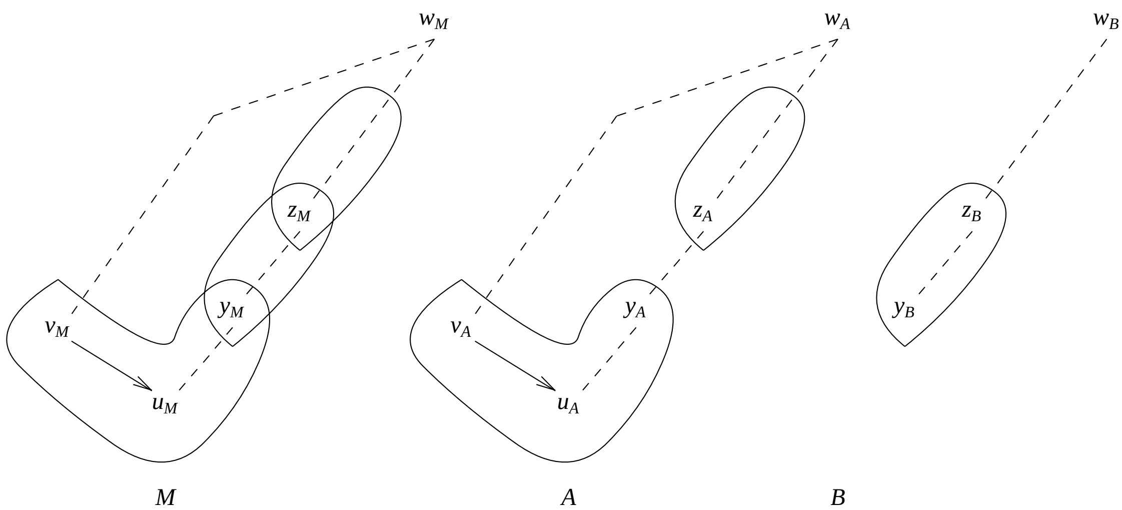


Figure 6-9.

In Figure 6-9, 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 6-9, 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 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 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 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 6.3. *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 6.2, 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 6.2, 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 6.4. *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 6.2, 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 equivalence 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 6.2, 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 6.2, 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 6.3 and 6.4.

Chapter 7

Comparison with the HPR Algorithm

It is interesting to compare the new program-integration algorithm with the HPR algorithm [Horwitz88, Horwitz89]. The new integration algorithm is actually a family of algorithms, parameterized by the equivalence-detection algorithms employed. To compare it with the HPR algorithm, we must settle on a fixed equivalence-detection algorithm. In this chapter, we show that the new integration algorithm combined with the Sequence-Congruence Algorithm improves on the HPR algorithm in the following sense:

(1) The new integration algorithm successfully integrates[17] the two variants with respect to the base program whenever the HPR algorithm succeeds.

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

However, we cannot compare the two integration algorithms directly since the HPR integration algorithm operates on program dependence graphs (PDGs) while the new integration algorithm operates on program representation graphs (PRGs)—we first need to modify the HPR algorithm to operate on PRGs. The new integration algorithm is, then, compared with the modified HPR algorithm.

In this chapter, we first modify the HPR algorithm to operate on PRGs. Because the HPR algorithm makes use of program slices, Section 7.1 demonstrates how slices can be extracted from PRGs and gives the Feasibility Lemma for program slices of PRGs. The modified HPR algorithm, presented in Section 7.2, is a straightforward translation of the original HPR algorithm; the difference is that it uses PRGs instead of PDGs. We show that the modified HPR algorithm is equivalent to the original HPR algorithm in that the modified HPR algorithm successfully integrates the two variants with respect to the base program if and only if the original HPR algorithm does, and when both algorithms succeed, they produce the same set of merged programs. In Section 7.3, we compare the new program-integration algorithm with the (modified) HPR algorithm. We are able to show that, given the same set of component tags, the new integration algorithm successfully integrates the two variants with respect to the base program whenever the HPR algorithm succeeds.

7.1. Feasibility Lemma for Program Representation Graphs

The HPR integration algorithm makes use of *slices* of program dependence graphs. In order to modify the HPR algorithm to work on program representation graphs, we first define slices of program representation graphs.

---

[17]The phrase "an integration algorithm *successfully integrates* the variants" means that no interference is detected and a merged program satisfying the semantic criterion is created.

A slice of a program representation graph can be extracted in the same way as a slice of a program dependence graph, except that we do not need to consider def-order edges. A slice of a PRG *R* with respect to a component *c* is the subgraph of *R* induced by all components that can reach *c* via a path of dependence edges in *R*.

Definition. A *slice* of a program representation graph *R* with respect to a set of (ϕ and non-ϕ) vertices *S*, denoted by *R/S*, is the subgraph of *R* induced by all vertices that can reach an element of *S* via a path of dependence edges.

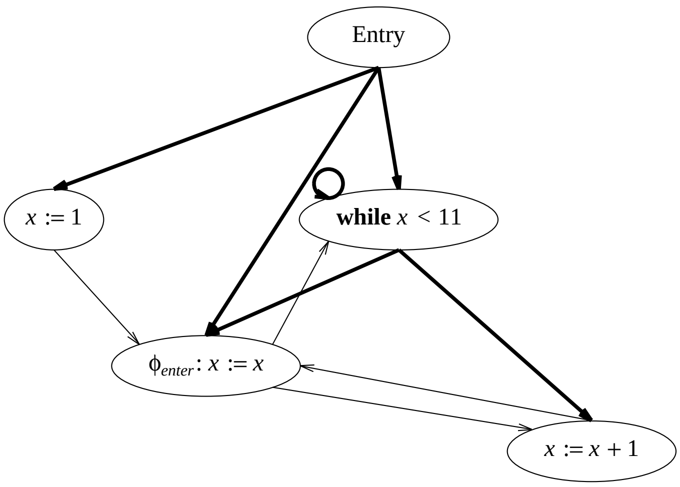
Note that a slice of *R* with respect to a vertex that does not appear in *R* is, by definition, the empty graph. A slice of the example PRG in Figure 3-2 is shown in Figure 7-1. The slice is taken with respect to the statement "$x := x + 1$."

We say a graph is a *feasible* PRG if it is isomorphic to the PRG of some program. It has been shown in Chapter 2 that slices of a feasible PDG are always feasible. For the same result to hold for PRGs, it is necessary to impose the further restriction that the slice be taken with respect to a set of non-ϕ vertices.

Lemma. (Decomposition Lemma). *Let R be a program representation graph and S_1 and S_2 be two sets of (ϕ or non-ϕ) vertices.* $R/(S_1 \cup S_2) = R/S_1 \cup R/S_2$.

Proof. Because any vertex or edge in $R/(S_1 \cup S_2)$ must be on a path from *Entry* to a vertex in S_1 or S_2, the vertex or edge must be in either R/S_1 or R/S_2. Therefore, $R/(S_1 \cup S_2)$ is a subgraph of $R/S_1 \cup R/S_2$. On the

(a)



(b)

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

Figure 7-1. (a) is a slice of the example program representation graph shown in Figure 3-2. The slice is taken with respect to the statement $x := x + 1$. (b) is a program whose PRG is isomorphic to the slice shown in (a).

other hand, both R/S_1 and R/S_2 are subgraphs of $R/(S_1 \cup S_2)$. Therefore, $R/S_1 \cup R/S_2$ is a subgraph of $R/(S_1 \cup S_2)$. We conclude that $R/(S_1 \cup S_2) = R/S_1 \cup R/S_2$. □

Lemma. (Feasibility Lemma for Program Representation Graphs). *For any program P, if R_Q is the slice of P's program representation graph with respect to a set of non-ϕ vertices, then R_Q is a feasible program representation graph.*

Proof. We prove this lemma by constructing a program from the program *P* and the slice R_Q, and by showing that the PRG of the constructed program is isomorphic to R_Q.

We construct a new program Q' from *P* and R_Q as follows: The components (statements and predicates) of Q' correspond to the vertices of R_Q (except the ϕ vertices); each component of Q' is subordinate to the same component that it is subordinate to in *P*; the components of Q' occur in the same relative order as they do in *P*. The variables that appear in the *end* statement of Q' are those variables whose *FinalUse* vertices are in R_Q. We use $R_{Q'}$ to denote the program representation graph of Q'. We want to show that R_Q and $R_{Q'}$ are isomorphic.

Because each component of Q' is subordinate to the same component that it is subordinate to in *P*, and because components of Q' occur in the same order as they occur in *P*, the control flow graph for program Q' is the projection of the control flow graph for program *P* (without the ϕ vertices) onto the components of Q'. That is, if *a* and *b* are components of Q', the projection of any path from *a* to *b* in the control flow graph of *P* onto the set of components of Q' is a path in the control flow graph of Q'. Furthermore, every path from *a* to *b* in the control flow graph of Q' is the projection of some path from *a* to *b* (and possibly several such paths) in the control flow graph of *P*.

We add ϕ_{if}, ϕ_{enter}, and ϕ_{exit} vertices to the control flow graph of Q' as explained in the definition of program representation graphs to create the augmented control flow graph of Q'. We now want to show that the ϕ vertices added to the control flow graph of Q' are exactly those appearing R_Q.

First we show that all ϕ vertices added to the control flow graph of Q' also exist in R_Q. If a vertex *c* labeled "ϕ_{if}: $x = x$" is added to the control flow graph of Q', then there must exist a non-ϕ vertex *a* inside the *if* statement that assigns a value to *x* and a non-ϕ vertex *b* that uses *x* and the definition[18] to *x* at *a* can reach the use of *x* at *b* via a path that passes through the end of the *if* statement, that is, there is an *x*-definition-free path from *a* to *b* that passes through the end of the *if* statement in the control flow graph of Q'. There must exist a corresponding *x*-definition-free path from *a* to *b* in the control flow graph of *P* because all definitions to *x* that can reach *b* in program *P* are included in Q'. Thus, vertex *c* also exists in R_P and there is an *x*-definition-free flow dependence path[19] $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P. Because *b* is in R_Q, *c* is in the slice R_Q as well.

If a pair of vertices *c* and *d* labeled "ϕ_{enter}: $x = x$" and "ϕ_{exit}: $x := x$", respectively, are added to the control flow graph of Q', then there must exist a non-ϕ vertex *a* inside the loop that assigns a value to *x* and a non-ϕ vertex *b* that uses *x* and the definition to *x* at *a* can reach the use of *x* at *b* via a path that passes through the end of the loop of *c*. That is, there is an *x*-definition-free path from *a* to *b* in the control flow graph of Q' that exits the loop of *c* at least once. There must exist a corresponding *x*-definition-free path from *a* to *b* in the con-

---

[18]A *definition* to a variable *x* is a non-ϕ assignment statement that assigns a value to *x*. A ϕ assignment is *not* considered a definition to a variable. (See Section 3.3.)

[19]An *x*-definition-free flow dependence path is a sequence of flow dependence edges $u_1 \longrightarrow_f u_2,\ u_2 \longrightarrow_f u_3,\ \ldots,\ u_{k-1} \longrightarrow_f u_k$ so that the variable *x* is assigned a value at u_1 and is used at u_k, and all vertices except u_1 and u_k are ϕ vertices. (See Section 3.3.)

trol flow graph of P because all definitions to x that can reach b in program P are included in Q'. Thus, vertices c and d also exist in R_P and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f d \longrightarrow_f^* b$ in R_P. Because b is in R_Q, c and d are in the slice R_Q as well.

If a vertex c labeled "ϕ_{enter}: $x = x$" is added to the control flow graph of Q' and there is no associated ϕ_{exit} vertex, then there must exist a non-ϕ vertex a inside the loop that assigns a value to x and a non-ϕ vertex b inside the loop that uses x and the definition to x at a can reach, via the predicate of the loop, the use of x at b. That is, there is an x-definition-free path from a, via the predicate of the loop, to b in the control flow graph of Q'. There must exist a corresponding x-definition-free path from a to b in the control flow graph of P because all definitions to x that can reach b in program P are included in Q'. Thus, vertex c also exists in R_P and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P. Because b is in R_Q, c is included in the slice R_Q as well.

Next, we show that all ϕ vertices in R_Q are added to the control flow graph of Q'. For any vertex c labeled "ϕ_{if}: $x := x$" in R_Q, there must be a non-ϕ vertex a inside the *if* statement that assigns a value to x and a non-ϕ vertex b that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_Q (and hence in R_P). Since a and b are in R_Q, a and b are included in the control flow graph of Q'. Because there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P, there is an x-definition-free path from a to b that passes through the end of the *if* statement in the control flow graph of P; hence there is a corresponding x-definition-free path from a to b that passes through the end of the *if* statement in the control flow graph of Q'. From the definition of program representation graphs, the vertex c is added to the control flow graph of Q'; hence c is in $R_{Q'}$.

For any pair of vertices c and d labeled "ϕ_{enter}: $x := x$" and "ϕ_{exit}: $x := x$", respectively, in R_Q, there must be a non-ϕ vertex a inside the *while* loop that assigns a value to x and a non-ϕ vertex b that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f d \longrightarrow_f^* b$ in R_Q (and hence in R_P). Since a and b are in R_Q, a and b are included in the control flow graph of Q'. Because there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f d \longrightarrow_f^* b$ in R_P, there is an x-definition-free path from a to b in the control flow graph of P that exits the loop of c at least once; hence there is a corresponding x-definition-free path from a to b in the control flow graph of Q'. From the definition of program representation graphs, the vertices c and d are added to the control flow graph of Q'; hence c and d are in $R_{Q'}$.

If a vertex c labeled "ϕ_{enter}: $x := x$" is in R_Q and there is no associated ϕ_{exit} vertex, then there must be a non-ϕ vertex a inside the *while* loop that assigns a value to x and a non-ϕ vertex b inside the the *while* loop that uses x and there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_Q (and hence in R_P). Since a and b are in R_Q, a and b are included in the control flow graph of Q'. Because there is an x-definition-free flow dependence path $a \longrightarrow_f^* c \longrightarrow_f^* b$ in R_P, there is an x-definition-free path from a via the predicate of the loop to b in the control flow graph of P; hence there is a corresponding x-definition-free path from a via the predicate of the loop to b in the control flow graph of Q'. From the definition of program representation graphs, the vertex c is added to the augmented control flow graph of Q'; hence c is in $R_{Q'}$.

Now we have shown that R_Q and $R_{Q'}$ have the same ϕ vertices. From the construction of Q', the only other possible differences between the vertex sets of R_Q and $R_{Q'}$ is in their *InitialState* vertices. By the definition of a slice, if there is an *InitialState*(x) vertex a for variable x in R_Q, there must be a flow edge $a \longrightarrow_f b$ in R_Q; hence the flow dependence edge $a \longrightarrow_f b$ is in R_P. This means that there is a path from the beginning of the control flow graph of P to b that is free of any definition to x. The projection of this path onto the components of Q' corresponds to a path in the control flow graph of Q'. The projected path runs from the beginning of the control flow graph of Q' to b and is free of any definition to x; consequently, there must be an *InitialState*(x)

vertex for x in $R_{Q'}$, which corresponds to vertex a in R_Q.

Arguing in the other direction, if there is an *InitialState*(x) vertex a for variable x in $R_{Q'}$, there must be a flow dependence edge $a \longrightarrow_f b$ in $R_{Q'}$, which corresponds to an x-definition-free path from the beginning of the program to b in the control flow graph of Q'. This means that there is a corresponding x-definition-free path from the beginning of the program to b in the control flow graph of P. By the construction of Q', because the vertex b is in Q', the vertex b must be in R_Q. Consequently, by the definition of a slice, the vertex a is in R_Q.

We have shown above that R_Q and $R_{Q'}$ have the same vertex sets, what remains to be shown is that R_Q and $R_{Q'}$ have the same edge sets.

Sub-proof 1. If the edge $a \longrightarrow b$ is in R_Q, then $a \longrightarrow b$ is in $R_{Q'}$.

(1) By the definition of a slice, if $a \longrightarrow_c b$ is a control dependence edge in R_Q, then $a \longrightarrow_c b$ is a control dependence edge in R_P, which means that b is subordinate to a in program P. Because a component in program Q' is subordinate to the same component that it is subordinate to in P, $a \longrightarrow_c b$ is in $R_{Q'}$ as well.

(2) By the definition of a slice, if $a \longrightarrow_f b$ is a flow dependence edge in R_Q, then $a \longrightarrow_f b$ is in R_P, which means that there is a path in the augmented control flow graph of P from a to b without any ϕ or non-ϕ assignments to the target variable of a. The projection of this path onto the components of Q' is a path in Q' that contains no redefinition to the target variable of a; thus, the flow dependence edge $a \longrightarrow_f b$ is in $R_{Q'}$.

Sub-proof 2. If the edge $a \longrightarrow b$ is in $R_{Q'}$, then $a \longrightarrow b$ is in R_Q.

(1) If $a \longrightarrow_c b$ is a control dependence edge in $R_{Q'}$, then b is subordinate to a in Q'; hence b is subordinate to a in P. Therefore, $a \longrightarrow_c b$ is a control dependence edge in R_P and, by the definition of a slice, $a \longrightarrow_c b$ is a control dependence edge in R_Q.

(2) Suppose $a \longrightarrow_f b$ is a flow dependence edge in $R_{Q'}$ but not in R_Q. Since the vertices a and b are in R_Q but the flow dependence edge $a \longrightarrow_f b$ is not in R_Q, by the definition of a slice, $a \longrightarrow_f b$ cannot be in R_P. Therefore, along every path from a to b in the augmented control flow graph of P there must be a ϕ or non-ϕ assignment of the target variable of a.

    Along each such path p, let c_p be the last (ϕ or non-ϕ) assignment of the target variable of a. Since the vertex b is in R_Q, the flow dependence edge $c_p \longrightarrow_f b$ is in R_P (and hence is in R_Q); the vertex c_p itself must be in R_Q and hence is in $R_{Q'}$. Because every path from a to b in the control flow graph of Q' is a projection of a path p from a to b in the control flow graph of P, there must be a ϕ or non-ϕ assignment to the target variable of a on each path from a to b in the augmented control flow graph of Q'. This means that the flow dependence edge $a \longrightarrow_f b$ can not be in $R_{Q'}$, which is a contradiction; therefore, the flow dependence edge $a \longrightarrow_f b$ is in R_Q.

We have shown that R_Q is isomorphic to $R_{Q'}$. Therefore, a slice of a feasible program representation graph with respect to a set of non-ϕ vertices is always a feasible program representation graph. □

7.2. The Modified HPR Integration Algorithm

The modified HPR integration algorithm is a straightforward translation of the original HPR algorithm discussed in Chapter 2. It takes as input a base program *Base*, and two variant programs A and B. Whenever the changes made to *Base* to create A and B do not “interfere” (as defined below), the modified algorithm produces a merged program M that incorporates the changed computations of A and B as well as the preserved computations common to all three. In this section, we show that the modified integration algorithm is equivalent to the HPR algorithm in the sense that, given the same *Base*, *A*, and *B* programs, they produce equivalent merged programs or they both report interference.

7.2.1. The Modified HPR Algorithm

There are three steps in the modified HPR algorithm. The first step determines slices that represent safe approximations to the changed computations of *A* and *B* and the computations of *Base* preserved in both *A* and *B*; the second step combines these slices to form the merged graph R_M; the third step tests R_M for interference.

Step 1: Determining changed and preserved computations

If the slice of variant R_A at vertex *v* differs from the slice of R_{Base} at *v*, then R_A and R_{Base} may compute different values at *v*. In other words, vertex *v* is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of R_A with respect to R_{Base}, denoted by $AR_{A,\,Base}$, to be the subset of non-ϕ vertices of R_A whose slices in R_{Base} and R_A differ $AR_{A,\,Base} = \{\, v \mid v \textit{ is a non}-\phi \textit{ vertex in } R_A \wedge\ (R_{Base} / v) \neq (R_A / v) \,\}$. We define $AR_{B,\,Base}$ similarly. It follows that the slices $R_A / AR_{A,\,Base}$ and $R_B / AR_{B,\,Base}$ capture the respective computations of *A* and *B* that differ from *Base*.

If the slice of R_{Base} with respect to vertex *v* is identical to the slices of R_A and R_B with respect to *v*, then all three programs compute the same sequence of values at *v*. We define the *preserved points* $PR_{Base,\,A,\,B}$ as the subset of non-ϕ vertices of R_{Base} with identical slices in R_{Base}, R_A, and R_A: $PR_{Base,\,A,\,B} = \{\, v \mid v \textit{ is a non}-\phi \textit{ vertex in } R_{Base} \wedge\ (R_{Base} / v) = (R_A / v) = (R_B / v) \,\}$. Thus, the slice $R_{Base} / PR_{Base,\,A,\,B}$ captures the computations of *Base* that are preserved in both *A* and *B*.

Step 2: Forming the merged graph

The merged graph, R_M, is formed by unioning the three slices that represent the changed and preserved computations:

$R_M = (R_A / AR_{A,\,Base}) \cup (R_B / AR_{B,\,Base}) \cup (R_{Base} / PR_{Base,\,A,\,B})$.

Step 3: Testing for interference

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

The second interference criterion arises because the merged graph may not be feasible; if the graph is infeasible, *A* and *B* interfere.

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

7.2.2. The Modified HPR Algorithm is Equivalent to the Original HPR Algorithm

Having modified the original HPR integration algorithm, we need to show that the modified HPR algorithm, which works on program representation graphs, is equivalent to the original algorithm, which works on program dependence graphs. To be more specific, we need to show that, for a base program *Base* and two variants *A* and *B*, the original algorithm successfully integrates *A*, *B*, and *Base* (and produces a merged program whose program dependence graph is G_{old}) if and only if the modified algorithm successfully integrates *A*, *B*, and *Base* (and produces a merged program whose program representation graph is R_{old}), and G_{old} and R_{old} represent "equivalent" merged programs.

By the Equivalence Theorem for PDGs, which is proved in Chapter 2, a PDG *G* represents a set of strongly equivalent programs whose PDGs are isomorphic to *G*. Similarly, by the Equivalence Theorem for PRGs, which is proved in Section 3.4, a PRG *R* represents a set of strongly equivalent programs whose PRGs are isomorphic to *R*. Furthermore, as we have shown in Chapter 3, the PDGs of two programs are isomorphic if and only if their PRGs are isomorphic. Thus, either *G* and *R* represent the same set of programs or they do not represent any common program. This leads to the following definition.

Definition. Let *G* be a program dependence graph and *R* be a program representation graph. *G* and *R* *represent the same set of programs*, denoted by $G \equiv R$, if the set of programs represented by *G* is the same as that represented by *R*.

It is immediate from the above discussion that, for any program *P* with PDG G_P and PRG R_P, $G_P \equiv R_P$.

Because the original HPR algorithm (the modified HPR algorithm) may produce any one of the programs that G_{old} (R_{old}, respectively) represents, it is more reasonable to show that the merged graphs G_{old} and R_{old} represent the same set of programs rather than to show that the two algorithms produce exactly the same program. That is, to show that the modified HPR algorithm is equivalent to the original HPR algorithm, we really want to show that $G_{old} \equiv R_{old}$.

In this subsection, we use a series of lemmas to prove that the modified HPR algorithm is equivalent to the original HPR algorithm. We use G_A, G_B, G_{Base}, R_A, R_B, and R_{Base} to denote the PDGs and PRGs of the three programs *A*, *B*, and *Base*, respectively. We use G_{old} and R_{old} to denote the respective merged graphs produced by the original HPR algorithm and by the modified HPR algorithm. Note that we do not assume that either of the two integration algorithms succeeds unless it is stated so explicitly.

In the following proofs, $\overline{AP}_{A,\,Base}$ denotes the complement of $AP_{A,\,Base}$: $\overline{AP}_{A,\,Base} = \{\ vertices\ in\ G_A\ \} - AP_{A,\,Base}$. We define $\overline{AP}_{B,\,Base}$ similarly. $\overline{AR}_{A,\,Base}$ denotes the complement of $AR_{A,\,Base}$: $\overline{AR}_{A,\,Base} = \{\ non-\phi\ vertices\ in\ R_A\ \} - AR_{A,\,Base}$. We define $\overline{AR}_{B,\,Base}$ similarly. Note that $PP_{Base,\,A,\,B} = \overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}$ and $PR_{Base,\,A,\,B} = \overline{AR}_{A,\,Base} \cap \overline{AR}_{B,\,Base}$.

Lemma 7.1. *Let G be a* PDG *and R be a* PRG. *If there is a program P with* PDG G_P *and* PRG R_P *such that* $G = G_P$ *and* $R = R_P$, *then* $G \equiv R$.

Proof. Suppose that there is a program *P* with PDG G_P and PRG R_P such that $G = G_P$ and $R = R_P$. Since $G_P \equiv R_P$, we conclude that $G \equiv R$. □

Lemma 7.2. *Let G be a feasible* PDG *and R be a feasible* PRG *such that* $G \equiv R$. *For any non-*ϕ *vertex u*, $G/u \equiv R/u$.

Proof. Because $G \equiv R$, there is a program *P* with PDG G_P and PRG R_P such that $G = G_P$ and $R = R_P$. Therefore, $G/u = G_P/u$ and $R/u = R_P/u$. As shown in the proof of the Feasibility Lemma for PDGs, G_P/u represents the program P' obtained by restricting the abstract syntax tree of *P* to just the statements and predicates included in G_P/u. As shown in the proof of the Feasibility Lemma for PRGs, R_P/u represents the program P'' obtained by restricting the abstract syntax tree of P to just the statements and predicates included in R_P/u. Note that G_P/u and R_P/u have the same set of non-ϕ vertices. Therefore, the two programs P' and P'' are identical. Because there is a program P' (or, equivalently, P'') that is represented by both G_P/u and R_P/u, by Lemma 7.1, $G_P/u \equiv R_P/u$. Because $G/u = G_P/u \equiv R_P/u = R/u$, we conclude that $G/u \equiv R/u$. □

Lemma 7.3. $\overline{AR}_{A,\ Base} = \overline{AP}_{A,\ Base}$. $AR_{A,\ Base} = AP_{A,\ Base}$. *Similarly,* $\overline{AR}_{B,\ Base} = \overline{AP}_{B,\ Base}$. $AR_{B,\ Base} = AP_{B,\ Base}$. $PR_{Base,\ A,\ B} = PP_{Base,\ A,\ B}$.

Proof. Let $u \in \overline{AP}_{A,\ Base}$. By the definition of $\overline{AP}_{A,\ Base}$, $G_A/u = G_{Base}/u$. By Lemma 7.2, $G_A/u \equiv R_A/u$ and $G_{Base}/u \equiv R_{Base}/u$; therefore, $R_A/u = R_{Base}/u$. So $u \in \overline{AR}_{A,\ Base}$; hence $\overline{AP}_{A,\ Base} \subseteq \overline{AR}_{A,\ Base}$.

On the other hand, suppose $u \in \overline{AR}_{A,\ Base}$. By the definition of $\overline{AR}_{A,\ Base}$, $R_A/u = R_{Base}/u$. By Lemma 7.2, $G_A/u \equiv R_A/u$ and $G_{Base}/u \equiv R_{Base}/u$; therefore, $G_A/u = G_{Base}/u$. So $u \in \overline{AP}_{A,\ Base}$; hence $\overline{AR}_{A,\ Base} \subseteq \overline{AP}_{A,\ Base}$.

Because $\overline{AP}_{A,\ Base} \subseteq \overline{AR}_{A,\ Base}$ and $\overline{AR}_{A,\ Base} \subseteq \overline{AP}_{A,\ Base}$, $\overline{AR}_{A,\ Base} = \overline{AP}_{A,\ Base}$. Since $\overline{AR}_{A,\ Base} = \overline{AP}_{A,\ Base}$ and R_A and G_A have the same non-ϕ vertices, their complements are equal, that is, $AR_{A,\ Base} = AP_{A,\ Base}$.

By the same argument, we know that $AR_{B,\ Base} = AP_{B,\ Base}$ and $\overline{AR}_{B,\ Base} = \overline{AP}_{B,\ Base}$.

Since $PR_{Base,\ A,\ B} = (\overline{AR}_{A,\ Base} \cap \overline{AR}_{B,\ Base})$ and $PP_{Base,\ A,\ B} = (\overline{AP}_{A,\ Base} \cap \overline{AP}_{B,\ Base})$, $PR_{Base,\ A,\ B} = PP_{Base,\ A,\ B}$. □

Lemma 7.4. *G_{old} and R_{old} always contain the same set of non-ϕ vertices.*

Proof. Note that $G_{old} = (G_A/AP_{A,\ Base}) \cup (G_B/AP_{B,\ Base}) \cup (G_{Base}/PP_{Base,\ A,\ B})$ and $R_{old} = (R_A/AR_{A,\ Base}) \cup (R_B/AR_{B,\ Base}) \cup (R_{Base}/PR_{Base,\ A,\ B})$. By Lemma 7.3, $AR_{A,\ Base} = AP_{A,\ Base}$. Note that for each $u \in AR_{A,\ Base}$ (or, equivalently, $u \in AP_{A,\ Base}$), R_A/u and G_A/u have the same set of non-ϕ vertices. Because (1) $R_A/AR_{A,\ Base} = \bigcup_{u \in AR_{A,\ Base}} R_A/u$, (2) $G_A/AP_{A,\ Base} = \bigcup_{u \in AP_{A,\ Base}} G_A/u$, and (3) $AR_{A,\ Base} = AP_{A,\ Base}$, $R_A/AR_{A,\ Base}$ and $G_A/AP_{A,\ Base}$ have the same set of non-ϕ vertices. By the same arguments, we know that $R_B/AR_{B,\ Base}$ and $G_B/AP_{B,\ Base}$ have the same set of non-ϕ vertices and $R_{Base}/PR_{Base,\ A,\ B}$ and $G_{Base}/PP_{Base,\ A,\ B}$ also have the same set of non-ϕ vertices. Since $G_{old} = (G_A/AP_{A,\ Base}) \cup (G_B/AP_{B,\ Base}) \cup (G_{Base}/PP_{Base,\ A,\ B})$ and $R_{old} = (R_A/AR_{A,\ Base}) \cup (R_B/AR_{B,\ Base}) \cup (R_{Base}/PR_{Base,\ A,\ B})$, we know that R_{old} and G_{old} have the same set of non-ϕ vertices. □

From the construction of G_{old} in the original HPR algorithm, every vertex must be taken from G_A or G_B, or both. That is, $V(G_{old}) \subseteq V(G_A) \cup V(G_B)$. This can be rewritten as

$$
\begin{aligned}
V(G_{old}) &\subseteq AP_{A,\ Base} \cup \overline{AP}_{A,\ Base} \cup AP_{B,\ Base} \cup \overline{AP}_{B,\ Base} \\
&= (AP_{A,\ Base} \cup (\overline{AP}_{A,\ Base} - AP_{B,\ Base})) \cup (AP_{B,\ Base} \cup (\overline{AP}_{B,\ Base} - AP_{A,\ Base})).
\end{aligned}
$$

In the proofs given below, it turns out to be convenient to use this for case analysis on the vertices of G_{old} (*i.e.*, for every vertex u in G_{old}, either $u \in AP_{A,\ Base} \cup (\overline{AP}_{A,\ Base} - AP_{B,\ Base})$ or $u \in AP_{B,\ Base} \cup (\overline{AP}_{B,\ Base} - AP_{A,\ Base})$). Similarly, with the modified HPR algorithm, for every non-ϕ vertex u in R_{old}, either $u \in AR_{A,\ Base} \cup (\overline{AR}_{A,\ Base} - AR_{B,\ Base})$ or $u \in AR_{B,\ Base} \cup (\overline{AR}_{B,\ Base} - AR_{A,\ Base})$.

Lemma 7.5. *Suppose the original HPR algorithm successfully integrates G_A, G_B, and G_{Base} and produces G_{old}. Let u be any (non-ϕ) vertex in G_{old}. If $u \in AP_{A,\ Base} \cup (\overline{AP}_{A,\ Base} - AP_{B,\ Base})$, then $G_{old}/u = G_A/u$. If $u \in AP_{B,\ Base} \cup (\overline{AP}_{B,\ Base} - AP_{A,\ Base})$, then $G_{old}/u = G_B/u$.*

Proof. Suppose $u \in AP_{A,\,Base}$. Then $G_A/u = G_{old}/u$ for otherwise there will be interference. On the other hand, if $u \in (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$, then either $u \in PP_{Base,\,A,\,B}$ or $u \in (\overline{AP}_{A,\,Base} - AP_{B,\,Base} - \overline{AP}_{B,\,Base})$. If $u \in PP_{Base,\,A,\,B}$, then $G_{old}/u = G_{Base}/u = G_A/u$. If $u \in (\overline{AP}_{A,\,Base} - AP_{B,\,Base} - \overline{AP}_{B,\,Base})$, then there must be a vertex $v \in AP_{A,\,Base}$ such that u is a vertex in G_A/v. Since $G_A/v = G_{old}/v$ (from the interference criterion of the original HPR algorithm), $G_A/u = (G_A/v)/u = (G_{old}/v)/u = G_{old}/u$.

By the same argument, if $u \in AP_{B,\,Base} \cup (\overline{AP}_{B,\,Base} - AP_{A,\,Base})$, then $G_{old}/u = G_B/u$. □

Lemma 7.6. *Suppose the modified HPR algorithm successfully integrates R_A, R_B, and R_{Base} and produces R_{old}. Let u be any non-ϕ vertex in R_{old}. If $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$, then $R_{old}/u = R_A/u$. If $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$, then $R_{old}/u = R_B/u$.*

Proof. Suppose $u \in AR_{A,\,Base}$. Then $R_A/u = R_{old}/u$ for otherwise there will be interference. On the other hand, if $u \in (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$, then either $u \in PR_{Base,\,A,\,B}$ or $u \in (\overline{AR}_{A,\,Base} - AR_{B,\,Base} - \overline{AR}_{B,\,Base})$. If $u \in PR_{Base,\,A,\,B}$ then $R_{old}/u = R_{Base}/u = R_A/u$. If $u \in (\overline{AR}_{A,\,Base} - AR_{B,\,Base} - \overline{AR}_{B,\,Base})$ then there must be a vertex $v \in AR_{A,\,Base}$ such that u is a vertex in R_A/v. Since $R_A/v = R_{old}/v$ (from the interference criterion of the modified HPR algorithm), $R_A/u = (R_A/v)/u = (R_{old}/v)/u = R_{old}/u$.

By the same argument, if $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$, then $R_{old}/u = R_B/u$. □

Lemma 7.7. *Suppose the original HPR algorithm successfully integrates G_A, G_B, and G_{Base} and produces G_{old}. Let R_{old} be the merged graph produced by the modified HPR algorithm on input R_A, R_B, and R_{Base}, and let u be any (non-ϕ) vertex in G_{old}. If $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$, then $R_A/u = R_{old}/u$. If $u \in AP_{B,\,Base} \cup (\overline{AP}_{B,\,Base} - AP_{A,\,Base})$, then $R_B/u = R_{old}/u$.*

Note that we do not assume R_{old} is feasible in this lemma.

Proof. Let u be any (non-ϕ) vertex in G_{old} such that $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$. By Lemma 7.5, $G_A/u = G_{old}/u$. Since $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$, one of the following three conditions must be true: (1) $u \in PP_{Base,\,A,\,B} = (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$, (2) $u \in AP_{A,\,Base}$, or (3) $u \in \overline{AP}_{A,\,Base} - AP_{B,\,Base} - \overline{AP}_{B,\,Base}$. We consider each in turn.

(1) Suppose $u \in PP_{Base,\,A,\,B}$. Then $G_A/u = G_B/u$. By Lemma 7.2, $G_A/u \equiv R_A/u$ and $G_B/u \equiv R_B/u$. Thus, $R_A/u = R_B/u$. By the construction of R_{old} and the fact that $u \in PP_{Base,\,A,\,B} = PR_{Base,\,A,\,B}$, R_{old}/u is a subgraph of $(R_A/u \cup R_B/u)$; furthermore, since $R_A/u = R_B/u$, R_{old}/u is a subgraph of R_A/u. On the other hand, since $u \in PP_{Base,\,A,\,B} = PR_{Base,\,A,\,B}$, $R_A/u = R_{Base}/u$, which, by the construction of R_{old}, is a subgraph of R_{old}/u. Thus, $R_A/u = R_{old}/u$.

(2) Suppose $u \in AP_{A,\,Base}$. If, for all vertices $u' \in AR_{B,\,Base}$ and all (ϕ and non-ϕ) vertices w in $(R_A/u \cap R_B/u')$, $R_A/w = R_B/w$, then $R_A/u = R_{old}/u$. So assume there is a vertex $u' \in AR_{B,\,Base}$ and a vertex w in $(R_A/u \cap R_B/u')$ such that $R_A/w \neq R_B/w$. Let u' and w be any vertices such that $u' \in AR_{B,\,Base}$, w is a vertex in $(R_A/u \cap R_B/u')$, and $R_A/w \neq R_B/w$.

    We claim that w must be a ϕ vertex. Suppose w is a non-ϕ vertex. Since $R_A/w \neq R_B/w$, $G_A/w \neq G_B/w$. Because $G_A/w \subseteq G_A/u \subseteq G_{old}$[20] and $G_B/w \subseteq G_B/u' \subseteq G_{old}$, $(G_A/w \cup G_B/w) \subseteq G_{old}/w$. Therefore, $G_A/w \neq G_{old}/w$ and $G_B/w \neq G_{old}/w$. Because $G_A/w \neq G_B/w$, either $w \in AP_{A,\,Base}$ or $w \in AP_{B,\,Base}$. If $w \in AP_{A,\,Base}$ then $G_A/w = G_{old}/w$; if $w \in AP_{B,\,Base}$ then $G_B/w = G_{old}/w$. Either case leads to a contrad-

[20]For two graphs G and H, $G \subseteq H$ denotes that G is a subgraph of H.

iction. Therefore, w must be a ϕ vertex.

Next we want to show that all definition sites[21] for w in R_A are definition sites for w in R_B and *vice versa*. Note that w is a ϕ vertex. Let x be the variable that is assigned to at w. Let v be a definition site for w in R_A. So there is an x-definition-free flow dependence path $v \longrightarrow_f^* w$ in R_A/u.

Since G_{old} is a feasible program dependence graph, let R be the program representation graph such that $R \equiv G_{old}$. Since $G_{old}/u = G_A/u$, $R/u = R_A/u$. Since $G_{old}/u' = G_B/u'$, $R/u' = R_B/u'$.

Because there is an x-definition-free flow dependence path $v \longrightarrow_f^* w$ in R_A/u and because $R_A/u = R/u$, the flow dependence path $v \longrightarrow_f^* w$ is in R/u and hence in R. Since w is a vertex in R_B/u' and $R_B/u' = R/u'$, w is a vertex in R/u'. Therefore, the flow dependence path $v \longrightarrow_f^* w$ is in R/u'. Because $R_B/u' = R/u'$, the flow dependence path $v \longrightarrow_f^* w$ is in R_B/u'. Therefore, v is a definition site for w in R_B. By the same argument, for any definition site v' for w in R_B, v' is a definition site for w in R_A. Therefore, all definition sites for w in R_A are definition sites for w in R_B and *vice versa*.

Let v be any definition site for w in R_A (or, equivalently, R_B). Note that v is a non-ϕ vertex in $R_A/u \cap R_B/u'$. As we have shown above, the assumption $R_A/v \neq R_B/v$ leads to a contradiction. Therefore, it must be that $R_A/v = R_B/v$. Because $R_A/w \neq R_B/w$ and, for all definition sites v for w, $R_A/v = R_B/v$, there must exist two definition sites v_1 and v_2 for w such that there is a def-order edge $v_1 \longrightarrow_{do} v_2$ in G_A/u and a def-order edge $v_2 \longrightarrow_{do} v_1$ in G_B/u'. Because $G_{old}/u = G_A/u$ and $G_{old}/u' = G_B/u'$, both def-order edges $v_1 \longrightarrow_{do} v_2$ and $v_2 \longrightarrow_{do} v_1$ are included in G_{old}, which makes G_{old} infeasible. This is a contradiction.

We conclude that there cannot be two vertices u' and w such that $u' \in AR_{B,\,Base}$, w is a vertex in $(R_A/u \cap R_B/u')$, and $R_A/w \neq R_B/w$. Thus, $R_A/u = R_{old}/u$.

(3) Suppose $u \in \overline{AP}_{A,\,Base} - AP_{B,\,Base} - \overline{AP}_{B,\,Base}$. Then there must be a vertex $v \in AP_{A,\,Base}$ such that u is a vertex in G_A/v. Because v is a vertex in G_{old} and $v \in AP_{A,\,Base}$, we have already proved in (2) above that $R_A/v = R_{old}/v$. Thus, $R_A/u = R_{old}/u$.

This completes the proof that for any (non-ϕ) vertex u in G_{old}, if $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$, then $R_A/u = R_{old}/u$. By the same argument, for any (non-ϕ) vertex u in G_{old}, if $u \in AP_{B,\,Base} \cup (\overline{AP}_{B,\,Base} - AP_{A,\,Base})$, then $R_B/u = R_{old}/u$. □

Lemma 7.8. Suppose the modified HPR algorithm successfully integrates R_A, R_B, and R_{Base} and produces R_{old}. Let G_{old} be the merged graph produced by the original HPR algorithm on input G_A, G_B, and G_{Base}, and let u be any non-ϕ vertex in R_{old}. If $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$, then $G_A/u = G_{old}/u$. If $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$, then $G_B/u = G_{old}/u$.

Note that we do not assume that G_{old} is feasible in this lemma.

Proof. Let u be any non-ϕ vertex in R_{old} such that $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$. By Lemma 7.6, $R_A/u = R_{old}/u$. Since $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$, one of the following three conditions must be true: (1) $u \in PR_{Base,\,A,\,B}$, (2) $u \in AR_{A,\,Base}$, or (3) $u \in \overline{AR}_{A,\,Base} - AR_{B,\,Base} - \overline{AR}_{B,\,Base}$. We consider each in turn.

---

[21]A *definition site* for a vertex w is a non-ϕ vertex v that assigned a value to a variable x and x may be used at w before being redefined. In a program representation graph, a non-ϕ vertex v is a definition site for w if and only if there is an x-definition-free flow dependence path $v \longrightarrow_f^* w$, where x is the variable that is assigned a value at v. In a program dependence graph, v is a definition site for w if and only if there is a flow dependence edge $v \longrightarrow_f w$.

(1) Suppose $u \in PR_{Base,\,A,\,B}$. Then $R_A/u = R_B/u$. By Lemma 7.2, $G_A/u \equiv R_A/u$ and $G_B/u \equiv R_B/u$. Thus, $G_A/u = G_B/u$. Since $u \in PR_{Base,\,A,\,B}$, G_{old}/u is a subgraph of $(G_A/u \cup G_B/u)$; furthermore, since $G_A/u = G_B/u$, G_{old}/u is a subgraph of G_A/u. On the other hand, since $u \in PR_{Base,\,A,\,B} = PP_{Base,\,A,\,B}$, $G_A/u = G_{Base}/u$, which is a subgraph of G_{old}/u. Thus, $G_A/u = G_{old}/u$.

(2) Suppose $u \in AR_{A,\,Base}$. If, for all vertices $u' \in AP_{B,\,Base}$ and all vertices w in $(G_A/u \cap G_B/u')$, $G_A/w = G_B/w$, then $G_A/u = G_{old}/u$. So assume there is a vertex $u' \in AP_{B,\,Base}$ and a vertex w in $(G_A/u \cap G_B/u')$ such that $G_A/w \neq G_B/w$. Let u' and w be any vertices such that $u' \in AP_{B,\,Base}$, w is a vertex in $(G_A/u \cap G_B/u')$, and $G_A/w \neq G_B/w$.

We claim that there cannot be such vertices as u' and w. Since $G_A/w \neq G_B/w$, $R_A/w \neq R_B/w$. Because $R_A/w \subseteq R_A/u \subseteq R_{old}$ and $R_B/w \subseteq R_B/u' \subseteq R_{old}$, $(R_A/w \cup R_B/w) \subseteq R_{old}/w$. Therefore, $R_A/w \neq R_{old}/w$ and $R_B/w \neq R_{old}/w$. Because $R_A/w \neq R_B/w$, either $w \in AR_{A,\,Base}$ or $w \in AR_{B,\,Base}$. If $w \in AR_{A,\,Base}$ then $R_A/w = R_{old}/w$; if $w \in AR_{B,\,Base}$ then $R_B/w = R_{old}/w$. Either case leads to a contradiction.

We conclude that there cannot be two vertices u' and w such that $u' \in AP_{B,\,Base}$, w is a vertex in $(G_A/u \cap G_B/u')$, and $G_A/w \neq G_B/w$. Thus, $G_A/u = G_{old}/u$.

(3) Suppose $u \in \overline{AR}_{A,\,Base} - AR_{B,\,Base} - \overline{AR}_{B,\,Base}$. Then there must be a (non-ϕ) vertex $v \in AR_{A,\,Base}$ such that u is a vertex in R_A/v. Because v is a non-ϕ vertex in R_{old} and $v \in AR_{A,\,Base}$, we have already proved in (2) above that $G_A/v = G_{old}/v$. Thus, $G_A/u = G_{old}/u$.

This completes the proof that for any non-ϕ vertex u in R_{old}, if $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$, then $G_A/u = G_{old}/u$. By the same argument, for any non-ϕ vertex u in R_{old}, if $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$, then $G_B/u = G_{old}/u$. □

Lemma 7.9. *Suppose (1) G is a feasible program dependence graph, (2) R is a (not necessarily feasible) program representation graph, (3) G and R have the same set of non-ϕ vertices, and (4)* $R = \bigcup_{u\,:\,non-\phi\ vertices} R/u$. *If, for all non-$\phi$ vertices u,* $G/u \equiv R/u$, *then* $G \equiv R$ *and hence R is also a feasible program representation graph.*

Proof. Since G is feasible, let P be any program represented by G and let G_P and R_P be the program dependence graph and the program representation graph of P, respectively. Thus, $G = G_P$. We want to show that $R = R_P$.

By the definition of $\equiv$, $G_P \equiv R_P$. Note that G, R, G_P and R_P have the same set of non-ϕ vertices. Let u be any non-ϕ vertex. By Lemma 7.2, $G_P/u \equiv R_P/u$. Since $G = G_P$, $G/u = G_P/u$. Because $R/u \equiv G/u = G_P/u \equiv R_P/u$, we know $R/u = R_P/u$. Furthermore, $R = \bigcup_{u\,:\,non-\phi\ vertices} R/u = \bigcup_{u\,:\,non-\phi\ vertices} R_P/u$. From the definition of PRGs, we know $\bigcup_{u\,:\,non-\phi\ vertices} R_P/u = R_P$. Thus, $R = R_P$.

Since $G = G_P \equiv R_P = R$, we conclude that $G \equiv R$. □

Lemma 7.10. *Suppose (1) R is a feasible program representation graph, (2) G is a (not necessarily feasible) program dependence graph, and (3) G and R have the same set of non-ϕ vertices. If, for all non-ϕ vertices u,* $G/u \equiv R/u$, *then* $G \equiv R$ *and hence G is also a feasible program dependence graph.*

Proof. Since R is feasible, let P be any program represented by R and let G_P and R_P be the program dependence graph and the program representation graph for P, respectively. Thus, $R = R_P$. We want to show that $G = G_P$.

By the definition of $\equiv$, $G_P \equiv R_P$. Note that G, R, G_P and R_P have the same set of non-ϕ vertices. Let u be any non-ϕ vertex. By Lemma 7.2, $G_P/u \equiv R_P/u$. Since $R = R_P$, $R/u = R_P/u$. Because

$G/u \equiv R/u = R_P/u \equiv G_P/u$, we know $G/u = G_P/u$. Furthermore, $G = \bigcup_{u\,:\,non-\phi\ vertices} G/u = \bigcup_{u\,:\,non-\phi\ vertices} G_P/u$. From the definition of PDGs, we know $\bigcup_{u\,:\,non-\phi\ vertices} G_P/u = G_P$. Thus, $G = G_P$.

Since $G = G_P \equiv R_P = R$, we conclude that $G \equiv R$. □

Theorem. The original HPR algorithm successfully integrates G_A, G_B, and G_{Base} (and produces G_{old}) if and only if the modified HPR algorithm successfully integrates R_A, R_B, and R_{Base} (and produces R_{old}). Furthermore, when the integrations succeed, $G_{old} \equiv R_{old}$.

Proof. First suppose the original HPR algorithm successfully integrates G_A, G_B, and G_{Base}. For any (non-ϕ) vertex u in G_{old}, either $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$ or $u \in AP_{B,\,Base} \cup (\overline{AP}_{B,\,Base} - AP_{A,\,Base})$. Suppose $u \in AP_{A,\,Base} \cup (\overline{AP}_{A,\,Base} - AP_{B,\,Base})$. By Lemma 7.5, $G_A/u = G_{old}/u$. By Lemma 7.7, $R_A/u = R_{old}/u$. By the definition of $\equiv$, $G_A \equiv R_A$. By Lemma 7.2, $G_A/u \equiv R_A/u$. Thus, $G_{old}/u = G_A/u \equiv R_A/u = R_{old}/u$. By the same argument, if $u \in AP_{B,\,Base} \cup (\overline{AP}_{B,\,Base} - AP_{A,\,Base})$ then $G_{old}/u = G_B/u \equiv R_B/u = R_{old}/u$. Therefore, for any (non-ϕ) vertex u in G_{old}, $G_{old}/u \equiv R_{old}/u$. By Lemma 7.4, G_{old} and R_{old} contain the same set of non-ϕ vertices. Because $R_{old} = (R_A/AR_{A,\,Base}) \cup (R_B/AR_{B,\,Base}) \cup (R_{Base}/PR_{Base,\,A,\,B})$, for every ϕ vertex v in R_{old}, there must be a non-ϕ vertex u in R_{old} such that v is in R_{old}/u. Therefore, $R_{old} = \bigcup_{u\,:\,non-\phi\ vertices} R_{old}/u$. By Lemma 7.9, $G_{old} \equiv R_{old}$.

We have already shown that if $u \in AR_{A,\,Base}$ then $R_A/u = R_{old}/u$ and if $u \in AR_{B,\,Base}$ then $R_B/u = R_{old}/u$. Therefore, R_M preserves the two slices $R_A/AR_{A,\,Base}$ and $R_B/AR_{B,\,Base}$. This, together with the previous result that R_{old} is feasible, proves that the modified HPR algorithm also succeeds.

Next suppose the modified HPR algorithm successfully integrates R_A, R_B, and R_{Base}. For any non-ϕ vertex u in R_{old}, either $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$ or $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$. Suppose $u \in AR_{A,\,Base} \cup (\overline{AR}_{A,\,Base} - AR_{B,\,Base})$. By Lemma 7.6, $R_A/u = R_{old}/u$. By Lemma 7.8, $G_A/u = G_{old}/u$. By the definition of $\equiv$, $G_A \equiv R_A$. By Lemma 7.2, $G_A/u \equiv R_A/u$. Thus, $G_{old}/u = G_A/u \equiv R_A/u = R_{old}/u$. By the same argument, if $u \in AR_{B,\,Base} \cup (\overline{AR}_{B,\,Base} - AR_{A,\,Base})$ then $G_{old}/u = G_B/u \equiv R_B/u = R_{old}/u$. Therefore, for any non-ϕ vertex u in R_{old}, $G_{old}/u \equiv R_{old}/u$. By Lemma 7.4, G_{old} and R_{old} contain the same set of non-ϕ vertices. By Lemma 7.10, $G_{old} \equiv R_{old}$.

We have already shown that if $u \in AP_{A,\,Base}$ then $G_A/u = G_{old}/u$ and if $u \in AP_{B,\,Base}$ then $G_B/u = G_{old}/u$. Therefore, G_M preserves the two slices $G_A/AP_{A,\,Base}$ and $G_B/AP_{B,\,Base}$. This, together with the previous result that G_{old} is feasible, proves that the original HPR algorithm also succeeds. □

7.3. Comparison Theorem

In this section, we compare the new integration algorithm with the HPR algorithm. Since the new integration algorithm is a family of integration algorithms, each parameterized by the equivalence-detection algorithm used, the "new integration algorithm" in this section refers to the one that uses the Sequence-Congruence Algorithm for detecting program components with equivalent behavior. In addition, the HPR algorithm actually refers to the modified HPR algorithm discussed in the previous section since the original HPR algorithm, which works on PDGs, is not directly comparable with the new integration algorithm.

As discussed in Chapter 2, when the HPR algorithm successfully integrates a base program and a set of variants, the execution behavior of the integrated program can be characterized in terms of the behaviors of the base program and the variants. As discussed below, given the same set of component tags, the new integration algorithm also succeeds, and produces a program whose execution behavior has the same characterization.

However, it is important to understand that for the same base and variant programs it is possible for the two algorithms to produce *different* sets of integrated programs. This situation is illustrated by the following integration example.

Program *Base*	Variant *A*	Variant *B*	Program $M1$	Program $M2$
program	**program**	**program**	**program**	**program**
$x := 1$	$x := 1$	$x := 1$	$x := 1$	$x := 1$
$y := x + 2$	$w := x + 2$	$y := x + 2$	$w := x + 2$	$w := x + 2$
	$y := w$		$y := w$	$y := w$
$z := y + 3$	$z := y + 3$		$z := y + 3$	
end(x, y)	**end**(x, y)	**end**(x, y)	**end**(x, y)	**end**(x, y)

Program $M1$ is the integrated program produced by the HPR algorithm; Program $M2$ is the integrated program produced by the new integration algorithm. The discrepancy between the two integrated programs is due to the assignment to z in variant A. The assignment to z in A is considered to be an affected component by the HPR algorithm because the slice with respect to this assignment in A is not equal to its counterpart in *Base*. Therefore, the assignment is included in the integrated program by the HPR algorithm. However, the Sequence-Congruence Algorithm discovers that the execution behaviors of the respective assignments to z in A and *Base* are, in fact, equivalent. This assignment is, therefore, *not* considered to be an affected component of A. Furthermore, this assignment statement is not a preserved component because it has been deleted in B. Since no affected components depend on this assignment to z in A, this assignment is not included in the integrated program produced by the new integration algorithm.

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

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

In this section, we use R_A, R_B, and R_{Base} to denote the respective program representation graphs of A, B, and *Base*. We use R_{old} and R_{new} to denote the respective merged graphs produced by the modified HPR algorithm and by the new integration algorithm.

The HPR algorithm assumes that there is a tagging mechanism that can consistently identify program components in different versions of the program. The tagging mechanism guarantees that (1) tags are unique within a given version and (2) if components in different versions of a program have the same tag, then they also have the same texts. Since the two integration algorithms should be compared under the same conditions, both conditions will also be assumed in our discussion of the new integration algorithm in this section. In particular, vertices with the same tag will always have the same text; hence the sets $Modified_A$ and $Modified_B$ in the new integration algorithm are always empty. From now on, issues about the text in a vertex will be ignored.

The two integration algorithms use different methods for establishing a correspondence among program components. In particular, components that have the same tag but are not sequence-congruent are corresponding components under the HPR algorithm, but they do *not* correspond under the new integration algorithm. In order to clarify this difference, we first prove Lemma 7.11, which shows that when *A*, *B*, and *Base* can be integrated by the HPR algorithm, there can be at most one vertex in R_{new} with a given tag. Again under the assumption that *A*, *B*, and *Base* can be integrated by the HPR algorithm, Lemma 7.12 shows that R_{new} is a subgraph of R_{old} and Lemma 7.13 shows that R_{new} is a slice of R_{old}. The proof of the Comparison Theorem follows from Lemma 7.13 and the Feasibility Lemma for PRGs.

Lemma 7.11. *Suppose A, B, and Base can be integrated by the HPR algorithm. Then there is at most one vertex with a given tag in* R_{new}.

Proof. First we have to show that when *A*, *B*, and *Base* can be integrated by the HPR algorithm, the new integration algorithm will produce a merged graph. That is, the new integration algorithm will *not* report interference in step (2) or in step (3) (see Chapter 5).

Interference in step (2) is due to conflicting text in corresponding components. However, we have already assumed, for the purposes of this section, that components with the same tag always have the same text. Thus, interference due to conflicting text will not happen.

Interference in step (3) can happen only when there is a component $u \in Unchanged$ such that $R_A//u$, $R_B//u$, and $R_{Base}//u$ are pairwise unequal. However, if $R_A//u$, $R_B//u$, and $R_{Base}//u$ are pairwise unequal, then R_A/u, R_B/u, and R_{Base}/u are pairwise unequal. Thus, the HPR algorithm will also report interference, which contradicts the assumption that *A*, *B*, and *Base* can be integrated by the HPR algorithm. Thus, interference in step (3) cannot happen.

Since no interference can occur in either step (2) or step (3), the new integration algorithm will produce a merged graph R_{new}.

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

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

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

If v_1 is a non-ϕ vertex then $v_1 \in AP_{A,\,Base}$. Because the HPR algorithm successfully integrates *A*, *B*, and *Base*, it must be that $R_A/v_1 = R_{old}/v_1$. On the other hand, if v_1 is a ϕ vertex then there must be a non-ϕ vertex $v_1' \in AP_{A,\,Base}$ such that v_1 is in the slice R_A/v_1'. Since $v_1' \in AP_{A,\,Base}$, $R_A/v_1' = R_{old}/v_1'$ and therefore, $R_A/v_1 = R_{old}/v_1$. Thus, regardless of whether v_1 is a ϕ vertex or a non-ϕ vertex, we have $R_A/v_1 = R_{old}/v_1$. By the same argument, $R_B/v_2 = R_{old}/v_2$.

Note that the HPR algorithm considers v_1 and v_2 to be the "same vertex" in performing graph union because they have the same tag. Thus, $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, However, since we assume v_1 and v_2 are not sequence-congruent, $R_A/v_1 \neq R_B/v_2$. This is a contradiction. Therefore, there cannot be two non-sequence-congruent vertices v_1 and v_2 in R_{new} with the same tag.

Next assume that there is a vertex in R_{Base} that has the same tag as v_1 and v_2. Let v_{Base} be such a vertex in R_{Base}. Because v_1 and v_2 are not sequence-congruent, $R_A/v_1 \neq R_B/v_2$. Hence, $R_A/v_1 \neq R_{Base}/v_{Base}$ or

$R_B/v_2 \neq R_{Base}/v_{Base}$. Without loss of generality, we may assume that $R_A/v_1 \neq R_{Base}/v_{Base}$.

Because $R_A/v_1 \neq R_{Base}/v_{Base}$, by the same argument as above, $R_A/v_1 = R_{old}/v_1$. There are two cases depending on whether or not $R_B/v_2 = R_{Base}/v_{Base}$.

Case 1. Suppose $R_B/v_2 \neq R_{Base}/v_{Base}$. Because $R_B/v_2 \neq R_{Base}/v_{Base}$, by the same arguments as above, the slice R_B/v_2 must be included in R_{old} and $R_B/v_2 = R_{old}/v_1$ for otherwise the HPR algorithm would report interference. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the assumption that v_1 and v_2 are not sequence-congruent.

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

(1) There is a vertex $w_B \in Affected_B$ such that v_2 is included in $R_B//w_B$. Since $w_B \in Affected_B$ and $Modified_B$ is an empty set, $w_B \in New_B$; hence $w_B \in AP_{B,\, Base}$ in the HPR algorithm. Because the HPR algorithm successfully integrates A, B, and $Base$, $R_{old}/w_B = R_B/w_B$. Therefore, $R_{old}/v_2 = R_B/v_2$. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the assumption that v_1 and v_2 are not sequence-congruent.

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

    If $R_B//w \neq R_{Base}//w$, $R_B/w \neq R_{Base}/w$; hence $w \in AP_{B,\, Base}$. Because (1) the HPR algorithm successfully integrates A, B, and $Base$ and (2) $w \in AP_{B,\, Base}$, $R_{old}/w = R_B/w$. Because v_2 is a vertex in $R_B//w$, $R_{old}/v_2 = R_B/v_2$. We conclude that $R_A/v_1 = R_{old}/v_1 = R_B/v_2$, but this contradicts the assumption that v_1 and v_2 are not sequence-congruent.

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

There is a contradiction in either case. Therefore, there cannot be two vertices v_1 and v_2 in R_{new} with the same tag. □

Lemma 7.12. *Suppose A, B, and Base can be integrated by the HPR algorithm. Then R_{new} is a subgraph of R_{old}.*

Proof. By Lemma 7.11, if A, B, and $Base$ can be integrated by the HPR algorithm, there is at most one vertex with a given tag in R_{new}. Thus, tags provide a means for identifying vertices of R_{new}.

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

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

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

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

$Preserved(u)$ is a subgraph of R_{old}.

Case 2. Suppose $u \in \overline{AP}_{A,\,Base}$ and $u \in AP_{B,\,Base}$. Because $R_A/u = R_{Base}/u$, $R_A//u = R_{Base}//u$. So $Preserved(u) = R_B//u$. Because $R_B//u$ is a subgraph of R_B/u and R_B/u is a subgraph of $R_B/AP_{B,\,Base}$, which, in turn, is a subgraph of R_{old}, $Preserved(u)$ is a subgraph of R_{old}.

Case 3. Suppose $u \in AP_{A,\,Base}$ and $u \in \overline{AP}_{B,\,Base}$. This case is similar to *Case 2*.

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

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

Proposition 2. $ChangedComps_A$ is a subgraph of R_{old}.

$ChangedComps_A$ is the union of $R_A//w_A$ for all vertices $w_A \in Affected_A$. It suffices to show that $R_A//w_A$ is a subgraph of R_{old} for each vertex $w_A \in Affected_A$. Let w_A be a vertex in $Affected_A$. Because $w_A \in Affected_A$ and $Modified_A$ is an empty set, $w_A \in New_A$. If there is no vertex w_{Base} in R_{Base} that has the same tag as w_A, by definition, $w_A \in AP_{A,\,Base}$. On the other hand, if there is a vertex w_{Base} in R_{Base} that has the same tag as w_A, we have $R_A/w_A \neq R_{Base}/w_{Base}$ since $w_A \in New_A$. Therefore, $w_A \in AP_{A,\,Base}$.

In either case, $w_A \in AP_{A,\,Base}$. Because $R_A//w_A$ is a subgraph of R_A/w_A and R_A/w_A is a subgraph of $R_A/AP_{A,\,Base}$ and $R_A/AP_{A,\,Base}$ is a subgraph of R_{old}, $R_A//w_A$ is a subgraph of R_{old}. Therefore, $ChangedComps_A$ is a subgraph of R_{old}.

Proposition 3. $ChangedComps_B$ is a subgraph of R_{old}.

This proposition is similar to Proposition 2.

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

Lemma 7.13. *Suppose A, B, and Base can be integrated by the HPR algorithm. Then R_{new} is a slice of R_{old}.*

Proof. Since the HPR algorithm successfully integrates *A*, *B*, and *Base*, R_{old} is a feasible PRG. Note that every vertex in a feasible PRG has a fixed number of incoming edges of a given type. We prove Lemma 7.13 by considering the incoming edges of each vertex in R_{new}.

By Lemma 7.12, R_{new} is a subgraph of R_{old}. The proposition that R_{new} is a slice of R_{old} is equivalent to the following proposition: if v is a vertex in R_{new} and there is a dependence edge $u \longrightarrow v$ in R_{old}, then the edge $u \longrightarrow v$ is in R_{new} as well.

Let v be a vertex in R_{new}. Suppose there is a dependence edge $u \longrightarrow v$ in R_{old}. We want to show that the edge $u \longrightarrow v$ is in R_{new} as well. Depending on the classification of v, there are four cases to consider:

Case 1. Suppose $v \in Affected_A$. Because $v \in Affected_A$, the limited slice $R_A//v$ is included in R_{new}. Note that every vertex in a PRG has a fixed number of edges of a given type. From the definition of limited slices, the limited slice $R_A//v$ must have included for v the correct number of incoming edges of each type. Therefore, R_{new} must have included the correct number of incoming edges for vertex v. Since R_{new} is a subgraph of R_{old} (by Lemma 7.12), every incoming edge of v in R_{new} is also in R_{old}. If the edge $u \longrightarrow v$ is in R_{old} but not in R_{new}, then v has an extra incoming edge in R_{old}, which makes R_{old} infeasible. This contradicts the observation that R_{old} is feasible. Therefore, the edge $u \longrightarrow v$ must also be in R_{new}.

Case 2. Suppose $v \in Affected_B$. This case is similar to *Case 1*.

Case 3. Suppose $v \in Unchanged$. Because $v \in Unchanged$, either $R_A // v$ is included in R_{new} or $R_B // v$ is included in R_{new}. In either case, by the same argument as in Case 1, the edge $u \longrightarrow v$ must also be in R_{new}.

Case 4. Suppose $v \in Intermediate_A$ or $v \in Intermediate_B$. Because $R_{new} = (R_A // Affected_A) \cup (R_B // Affected_B) \cup (\bigcup_{u \in Unchanged} Preserved(u))$, v is included in the limited slice $R_A // w_A$ for some $w_A \in (Affected_A \cup Unchanged)$ or the limited slice $R_B // w_B$ for some $w_B \in (Affected_B \cup Unchanged)$. In either case, by the same argument as in Case 1, the edge $u \longrightarrow v$ must also be in R_{new}.

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

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

Proof. Because the HPR algorithm successfully integrates *A*, *B*, and *Base*, R_{old} is a feasible PRG. From Lemma 7.13, R_{new} is a slice of R_{old}. Because all the useless ϕ statements are removed from the merged graph in the fifth step of the new integration algorithm, by the Feasibility Lemma for PRGs, R_{new} is a feasible slice of R_{old}. We conclude that the new integration algorithm also produces a feasible merged program representation graph. □

Extra components included in the integrated program by the HPR algorithm are the result of that algorithm's less precise computation of affected components; the fact that the Integration Theorem holds for the new integration algorithm (see Chapter 6) assures us that the programs produced by the new integration algorithm are reasonable ones.

It is interesting to consider the kinds of changes that cause the HPR algorithm to report interference, while the new integration algorithm succeeds in producing an integrated program. The first example in Figure 1-1 illustrates that the new integration algorithm allows variables to be renamed and certain statements to be moved into or out of conditional statements. Furthermore, the new integration algorithm also allows insertion or deletion of trivial assignments (*i.e.* assignments of the form $x := y$) in the program.

There is another class of integration problems on which the HPR algorithm reports interference while the new integration algorithm succeeds. These are problems in which both variants change a component's execution behavior (in different ways). In this case, the HPR algorithm reports interference because its definition of corresponding vertices relies only on tags; there can be only one copy of the changed component in the integrated program, and it cannot simultaneously have both changed behaviors. In contrast, the new integration algorithm considers a component of a variant to be a *new* component whenever its execution behavior has been changed. Thus, even if there is a component in the other variant that has the same tag this does not cause any interference since the two components are considered distinct new components by the new integration algorithm. The programs shown below illustrate this situation.

Program *Base*	Variant *A*	Variant *B*	Program *M*
program	**program**	**program**	**program**
<T1> $x := 1$	<T1> $x := 1$	<T1> $x := 1$	<T1> $x := 1$
<T2> $x := 2$	<T4> $y := x + 4$	<T2> $x := 2$	<T4> $y := x + 4$
<T3> $x := 3$	<T2> $x := 2$	<T4> $y := x + 4$	<T2> $x := 2$
<T4> $y := x + 4$	<T3> $x := 3$	<T3> $x := 3$	<T4> $y := x + 4$
<T5> **end**(x)	<T5> **end**(x)	<T5> **end**(x)	<T3> $x := 3$
			<T5> **end**(x)

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

The four kinds of changes are not the only capabilities of the new integration algorithm. Since the new integration algorithm is actually a family of algorithms, given more powerful equivalence-detection techniques, the new integration algorithm can accommodate more classes of semantics-preserving transformations.

The new integration algorithm allows different stages of a computation to be modified in different variants and is still able to merge these changes. For instance, consider the second example in Figure 1-1. That example consists of three stages: one to compute the sum and product of the integers from 1 to 10, the other to compute *ratio*, and the final to compute *percentage*. The first stage has been changed in variant *B* (but the same values are computed for the variables *sum* and *prod*) whereas the second stage has been changed in variant *A* (the same value for *ratio* is computed anyway), The third stage (the assignment to *percentage*) consists of code added by *B*. Given an equivalence-detection algorithm that can discover these semantics-preserving transformation in the first two stages, the new integration algorithm can produce a merged program as shown in Figure 1-1.

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

A final point of comparison with the HPR algorithm is that the algebraic properties of the HPR algorithm have been characterized using Brouwerian algebra [Reps89a]. Unfortunately, the new integration algorithm does not seem to fit this model; some progress towards an algebraic characterization of the new integration algorithm has been made by G. Ramalingam [Ramalingam90].

Chapter 8

Conclusions

8.1. Work Accomplished

In this thesis, we have provided semantic foundations for the HPR program-integration algorithm, discussed the semantic criterion for program integration, presented a new data structure (program representation graphs) for representing programs, described the Sequence-Congruence Algorithm for detecting program components that have equivalent execution behavior, and proposed a new semantics-based program-integration algorithm that can accommodate semantics-preserving transformations. The advantages of the new program-integration algorithm and a comparison with the HPR integration algorithm have also been presented.

Given a base program and two variants, the HPR integration algorithm creates an integrated program by merging certain program slices. The Feasibility Lemma and the Slicing Theorem provide the necessary syntactic and semantic justifications for extracting slices from a dependence graph. The Termination Theorem, on the other hand, states a sufficient condition for a composite program (*e.g.*, the merged program) to terminate normally. Using these results, we showed in Chapter 2 that the HPR integration algorithm satisfies the following semantic criterion for program integration:

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

(1) *M* terminates normally on σ.

(2) For every variable *x* that is defined in the final state of *A*, if *x*'s final values after executing *Base* and *A* are different, then *x*'s final value in *M* is the same as in *A* (that is, *M* agrees with *A* on *x*).

(3) For every variable *y* that is defined in the final state of *B*, if *y*'s final values after executing *Base* and *B* are different, then *y*'s final value in *M* is the same as in *B* (that is, *M* agrees with *B* on *y*).

(4) For every variable *z* that is defined in the final state of *Base*, if *z*'s final values after executing *Base*, *A*, and *B* are the same, then *z*'s final value in *M* is the same as in all three (that is, *M* agrees with all three on *z*).

Any program that satisfies (1)-(4) can be viewed as the integrated version of the two variants with respect to the base program. If no such program exists, the changes made in the variants interfere. Note that this criterion is not decidable; consequently, any integration *algorithm* will fail on some examples for which a program satisfying (1)-(4) exists.

The HPR integration algorithm represents a fundamental advance over text-based integration algorithms in that the HPR algorithm attempts to merge the changed and preserved *computations* of the variants. This property distinguishes the HPR integration algorithm from text-based integration algorithms. However, there is room for improvements.

We developed a new integration algorithm that extends the capabilities of the HPR integration algorithm in two respects: (1) it is flexible and extendible in that additional techniques for detecting program components with equivalent behaviors can be easily incorporated in the new integration algorithm and (2) it can accommodate semantics-preserving transformations.

To determine whether a component's behavior has been changed is a fundamental problem for a program-integration tool. Although any program-integration algorithm must conservatively determine the set of affected components (because program equivalence is an undecidable problem), when more exact equivalence-detection techniques are employed, it is possible to avoid certain cases of spurious interference due to inexact determination of the set of affected components.

For this purpose, we developed the Sequence-Congruence Algorithm that partitions program components of several programs into disjoint equivalence classes so that two components are in the same class only if they have equivalent execution behavior. The Sequence-Congruence Algorithm can be employed by the new integration algorithm.

The new program-integration algorithm uses program representation graphs as the intermediate form for programs. Program representation graphs are equivalent to the program dependence graphs used in the HPR algorithm. A nice property of PRGs (and PDGs) is that programs with isomorphic PRGs (or, equivalently, PDGs) have equivalent execution behavior. This property is essential (both for the new integration algorithm as well as the HPR algorithm) because the result of integration may be any program whose PRG (or PDG, respectively) is isomorphic to a merged graph. This property is also essential to every semantics-based program-integration algorithm because the execution behavior of a program must be completely captured by whatever intermediate form is used by the integration systems.

The new integration algorithm is actually a family of algorithms, parameterized by the the equivalence-detection techniques used. Any equivalence-detection techniques can be employed as the first step of the new integration algorithm. Many techniques used in compiler optimization, such as constant propagation, invariant code movement, and common subexpression elimination, can be combined with the Sequence-Congruence Algorithm to detect larger classes of program components with equivalent behavior. Alpern, Wegman, and Zadeck [Alpern88] discussed two such extensions. It should also be possible to use knowledge of semantics-preserving transformations that have been applied to a program or certain parts of the program to detect larger classes of program components with equivalent behaviors. This knowledge can either be recovered by an equivalence-detection algorithm or supplied directly by a program transformation system.

One of the main characteristics of the new integration algorithm is concerned with the ability to accommodate semantics-preserving transformations. Given a sufficiently powerful algorithm to detect components with equivalent behaviors, the new integration algorithm allows semantics-preserving transformations to be applied to different stages of a computation in different variants and is still able to merge these changes. Since the new integration algorithm employs limited slicing to extract the affected and the preserved computations, such semantics-preserving transformations can be easily accommodated in the new integration algorithm.

8.2. Future Work

In this work, we studied program integration in a much simplified setting. It is our plan to extend our work to realistic languages and to apply it to solve real world problems. Some related research topics are outlined below.

Detecting components with equivalent behaviors

One of the fundamental problems in program integration is to detect program components with equivalent behaviors. Although this is an undecidable problem, research in this direction is well worth pursuing because the results obtained from research in this direction can also be applied in many other areas such as compilers.

On the other hand, many existing techniques used in compiler optimization, such as constant propagation, could detect equality of variables at certain moments during program execution. Adapting these existing techniques to detect components with equivalent behaviors requires further research.

Algebraic properties of operators are another source of information that an equivalence-detection algorithm could exploit. In the Sequence-Congruence Algorithm, all operators are treated as uninterpreted function symbols. This limits the capability of the equivalence-detection algorithm. By making use of properties of the operators, larger equivalence classes could be found. For instance, the Sequence-Congruence Algorithm can find larger equivalence classes when it knows some operators, such as addition, are commutative. The technique works as follows: the two incoming flow dependence edges of a vertex that contains a commutative operator are assigned the same type, rather than distinct types. The Sequence-Congruence Algorithm proceeds as before, except that the basic partitioning algorithm needs to be replaced by the naive partitioning algorithm.[22] (Alpern et al. in [Alpern88] uses hyperedges to handle commutative operators, which is very similar to the above approach.) Other algebraic properties, such as associativity and distributivity, might also be useful in detecting equivalent components.

It is obvious that the equivalence-detection algorithm could also benefit from additional information from the programmers. Such information could be supplied as pragmas or could be obtained through interaction with the programmers. For instance, if it is asserted that two components have equivalent behavior, a technique called *congruence closure* [Downey80, Nelson80] can merge the equivalence classes which the two components belong to and propagate the effects to combine other equivalence classes.

Extending to realistic languages

Our ultimate goal is to build a practical integration system that assists programmers to develop and maintain software. To achieve this goal, it is necessary to extend the integration algorithm to realistic languages. Additional language constructs involve new problems. For instance, in the presence of pointer variables, detection of components with equivalent behaviors is harder than it is for the simplified language we have worked with. Although some progress has been made toward extending the Sequence-Congruence Algorithm to handle certain kinds of generalized loops and procedures, it requires further study to extend the integration algorithm to handle these and other language constructs.

Relaxing the requirements of program integration

We feel that some requirements of program integration stated in Chapter 1 might be relaxed in order to make integration succeed more often. For example, if the integration system is allowed to rename variables or to make up new statements and predicates, an infeasible merged graph can sometimes be converted to a feasible one allowing integration to succeed. It would be useful to investigate alternative integration requirements.

---

[22]By the "naive partitioning algorithm," we mean the worklist algorithm that maintains a worklist of potentially partitionable classes; it repeated selects a class from the worklist, partitions the class, and augment the worklist (if necessary) until the worklist is empty.

Glossary

This glossary contains definitions of some important terms used in the thesis. The chapters in the parentheses denote the places where the terms are defined.

Analogous components. Two components are *analogous* if they have the same tag (Chapter 6).

Analogous flow (or control) predecessors. The analogous flow (or control) predecessors of two vertices u_1 and u_2 are two vertices v_1 and v_2 such that the flow (or control, respectively) dependence edges $u_1 \longrightarrow v_1$ and $u_2 \longrightarrow v_2$ have the same type (Chapter 4).

Analogous operands. If v_1 and v_2 are the analogous flow predecessors of u_1 and u_2 and if v_1 and v_2 assign a value to variables a_1 and a_2, then we say variables a_1 and a_2 are *analogous operands* of vertices u_1 and u_2, respectively (Chapter 4).

Comparable components. Two components c_1 and c_2 are *comparable* components if and only if (1) c_1 and c_2 have equivalent behaviors, (2) c_1 and c_2 are the same kinds of vertices, (3) corresponding incoming control dependence edges of c_1 and c_2 have the same *true* or *false* labels, and (4) analogous control predecessors of c_1 and c_2 are comparable components (Chapter 5).

Corresponding components. In the new integration algorithm, two components c_1 and c_2 are *corresponding* components if and only if (1) c_1 and c_2 are comparable components, (2) c_1 and c_2 have the same tag, and (3) if c_1 and c_2 are assignment statements, they assign to the same variable (Chapter 5). In the HPR algorithm, two components are corresponding components when they have the same tag (Chapter 2).

Definition. A *definition* to a variable x is a non-ϕ assignment statement that assigns a value to x. A ϕ assignment is *not* considered a definition to a variable (Chapter 3).

Definition-free flow dependence path. An *x-definition-free flow dependence path* is a sequence of flow dependence edges $a_1 \longrightarrow_f a_2$, $a_2 \longrightarrow_f a_3$, $\ldots$, $a_{k-1} \longrightarrow_f a_k$ so that the variable x is assigned a value at a_1 and is used at a_k, and all vertices except a_1 and a_k are ϕ vertices. An x-definition-free flow dependence path in the PRG of program P corresponds to an x-definition-free path in the control flow graph of P, which, in turn, corresponds to a flow dependence edge in the PDG of P (Chapter 3).

Originating vertex. In the new integration algorithm, 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 (Chapter 6).

Bibliography

Aho74.

Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).

Aho86.

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

Allen70.

Allen, F.E., "Control flow analysis," *Proc. ACM Symposium on Compiler Optimization* (July, 1970), *ACM SIGPLAN Notices* **5**(7) pp. 67-85 (July 1970).

Allen72.

Allen, F.E. and J.A. Cocke, "A catalogue of optimizing transformations," pp. 1-30 in *Design and Optimization of Compilers,* ed. R. Rustin,Prentice-Hall, Englewood Cliffs, N.J. (1972).

Allen83.

Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Dept. of Math. Sciences, Rice Univ., Houston, TX. (April 1983).

Alpern88.

Alpern, B., M.N. Wegman, and F.K. Zadeck, "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, Jan. 13-15, 1988), ACM, New York (January 1988).

Ball90.

Ball, T., S. Horwitz, and T. Reps, "Correctness of an algorithm for reconstituting a program from a dependence graph," TR-947, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (July 1990).

Baxter89.

Baxter, W. and H.R. Bauer, III, "The program dependence graph and vectorization," pp. 1-11 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York (January 1989).

Berzins86.

Berzins, V., "On merging software extensions," *Acta Informatica* **23** pp. 607-619 (1986).

Burstall77.

Burstall, R.M. and J. Darlington, "A transformation system for developing recursive programs," *J. ACM* **24**(1) pp. 44-67 (January 1977).

Cheatham84.

Cheatham, T.E. Jr., "Reusability through program transformations," *IEEE Trans. Software Engineering* **SE-10**(5) pp. 589-594 (September 1984).

Cytron89.

Cytron, R., J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck, "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York (January 1989).

Cytron86.

Cytron, R., A. Lowry, and K. Zadeck, “Code motion of control structures in high-level languages,” pp. 70-85 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages,* (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York (January 86).

Darlington76.

Darlington, J. and R. Burstall, “A system which automatically improves programs,” *Acta Informatica* **6**(1) pp. 41-60 (1976).

Downey80.

Downey, P.J., R. Sethi, and R.E. Tarjan, “Variations on the common subexpression problem,” *J. ACM* **27**(4) pp. 758-771 (1980).

Feather82.

Feather, M.S., “A system for assisting program transformation,” *ACM Trans. Programming Languages and Systems* **4**(1) pp. 1-20 (January 1982).

Ferrante87.

Ferrante, J., K.J. Ottenstein, and J.D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

Gerhart75.

Gerhart, S.L., “Correctness-preserving program transformations,” pp. 54-66 in *Conference Record of the Second ACM Symposium on Principles of Programming Languages,* (Palo Alto, CA, Jan. 20-22, 1975), ACM, New York (1975).

Hecht77.

Hecht, M.S., *Flow Analysis of Computer Programs,* North Holland, Amsterdam (1977).

Hopcroft71.

Hopcroft, J.E., “An *n log n* algorithm for minimizing states in a finite automaton,” pp. 189-196 in *The Theory of Machines and Computations,* ed. Z. Kohavi and A. Paz,Academic Press, New York (1971).

Horwitz87.

Horwitz, S., J. Prins, and T. Reps, “Integrating non-interfering versions of programs,” TR-690, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (March 1987).

Horwitz87a.

Horwitz, S., J. Prins, and T. Reps, “On the adequacy of program dependence graphs for representing programs,” TR-699, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (June 1987).

Horwitz88.

Horwitz, S., J. Prins, and T. Reps, “Integrating non-interfering versions of programs,” pp. 133-145 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, Jan. 13-15, 1988), ACM, New York (January 1988).

Horwitz88a.

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

Horwitz88b.

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

Horwitz89.

Horwitz, S., J. Prins, and T. Reps, “Integrating non-interfering versions of programs,” *ACM Trans. Programming*

Languages and Systems **11**(3) pp. 345-387 (July 1989).

Horwitz90.

Horwitz, S., "Identifying the semantic and textual differences between two versions of a program," *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation,* (White Plain, New York, June 20-22, 1990), *ACM SIGPLAN Notices* **25**(6) pp. 234-245 ACM, (June 1990).

Huet78.

Huet, G. and B. Lang, "Proving and applying program transformations expressed with second-order patterns," *Acta Informatica* **11** pp. 31-55 (1978).

Kennedy78.

Kennedy, K., "Use-definition chains with applications," *Computer Languages* **3** pp. 163-179 Pergamon Press, (1978).

Kuck72.

Kuck, D.J., Y. Muraoka, and S.C. Chen, "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. Computers* **C-21**(12) pp. 1293-1310 (December 1972).

Kuck81.

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

Loveman77.

Loveman, D.B., "Program improvement by source-to-source transformation," *J. ACM* **24**(1) pp. 121-145 (January 1977).

Nelson80.

Nelson, G. and D.C. Oppen, "Fast decision procedures based on congruence closure," *J. ACM* **27**(2) pp. 356-364 (April 1980).

Ottenstein78.

Ottenstein, K.J., "Data-flow graphs as an intermediate program form," Ph.D. Thesis, Purdue Univ., Indiana (August 1978).

Ottenstein84.

Ottenstein, K.J. and L.M. Ottenstein, "The program dependence graph in a software development environment," *Proceedings 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).

Padua86.

Padua, D.A. and M.J. Wolfe, "Advanced compiler optimizations for supercomputers," *Comm. ACM* **29**(12) pp. 1184-1201 (December 1986).

Paige83.

Paige, R., "Transformational programming—Applications to algorithms and systems," pp. 73-87 in *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 24-26, 1983), ACM, New York (January 1983).

Partsch83.

Partsch, H. and R. Steinbrüggen, "Program transformation systems," *ACM Computing Surveys* **15**(3) pp. 199-236 (September 1983).

Ramalingam90.

Ramalingam, G., Personal communication, July, 1990.

Reif82.

Reif, J.H. and Robert E. Tarjan, “Symbolic program analysis in almost linear time,” *SIAM J. of Computing* **11**(1) pp. 81-93 (February 1982).

Reps84.

Reps, T. and T. Teitelbaum, “The synthesizer generator,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 41-48 (May 1984).

Reps88.

Reps, T. and W. Yang, “The semantics of program slicing,” TR-777, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (June 1988).

Reps89.

Reps, T. and W. Yang, “The semantics of program slicing and program integration,” *Proceedings of the International Joint Conference on Theory and Practice of Software Development (Colloquium on Current Issues in Programming Languages),* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science* **352** pp. 360-374 Springer-Verlag, (1989).

Reps89a.

Reps, T., “On the algebraic properties of program integration,” TR-856, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (June 1989).

Rosen88.

Rosen, B.K., M.N. Wegman, and F.K. Zadeck, “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, Jan. 13-15, 1988), ACM, New York (January 1988).

Shapiro69.

Shapiro, R.M. and H. Saint, “The representation of algorithms,” RADC-TR-69-313, Rome Air Development Center, New York (September 1969).

Wegman85.

Wegman, M.N. and F.K. Zadeck, “Constant Propagation with Conditional Branches,” pp. 291-299 in *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages,* (New Orleans, LA, Jan. 14-16, 1985), ACM, New York (January 1985).

Weiser84.

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

Yang89.

Yang, W., S. Horwitz, and T. Reps, “Detecting program components with equivalent behaviors,” TR-840, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (April 1989).

Yang89a.

Yang, W., S. Horwitz, and T. Reps, “A new program integration algorithm,” TR-899, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (December 1989).

A NEW ALGORITHM

FOR

SEMANTICS-BASED PROGRAM INTEGRATION

by

WUU YANG

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

1990

to my Parents
Jyng-Pyng and Yuh-Wha
They gave me everything but a pair of shoes;
they fear that I will wander farther and farther away.

to my Wife
Pin-Chia
for her encouragement, inspiration, love, and sacrifice

Acknowledgements

I am much indebted to Professor Thomas Reps and Professor Susan Horwitz, my advisors and committee chairs, for their superb guidance and immense patience in all aspects during my study as well as their encouragement and support. I wish to express my deepest gratitude to Professor Charles N. Fischer for his extremely helpful comments and suggestions and to Professor Marvin H. Solomon and Professor Terrence S. Millar for serving on my examination committee.

Thanks are also due to Dr. Bowen Alpern for his participation in the early stage of this research and due to Professor Robert Paige who pointed out the possibility of using congruence closure to find larger equivalence classes.

I wish to express my thanks to Felix S.-T. Wu who taught me many lessons during my study in Madison. I also wish to thank Thomas Ball, Dave Binkley, Phil Pfeiffer and G. Ramalingam with whom I had a lot of beneficial discussions and Anthony Rich whose fantastic ideas interested me very much.

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 and CCR-8958530, 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, the Cray Research Foundation, Siemens, Hewlett-Packard, and 3M.

Table of Contents

List of Figures