DETECTING PROGRAM COMPONENTS WITH EQUIVALENT BEHAVIORS

Wuu Yang
Susan Horwitz
Thomas Reps

Computer Sciences Technical Report #840

April 1989

Detecting Program Components With Equivalent Behaviors

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

The execution behavior of a program component is defined as the sequence of values produced at the component during program execution. This paper presents an efficient algorithm for detecting program components – in *one or more* programs – that exhibit identical execution behaviors. The algorithm operates on a new graph representation for programs that combines features of static-single-assignment forms and program dependence graphs. The result provides insight into the relationship between execution behaviors and (control and flow) dependences in the program. The algorithm, called the *Sequence-Congruence Algorithm*, is applicable to programs written in a language that includes scalar variables and constants, assignment statements, conditional statements, and while-loops. The Sequence-Congruence Algorithm can be used as the basis for an algorithm for integrating program variants.

Categories and Subject Descriptors: D.3.4 **[Programming Languages]**: Processors – *compilers, interpreters, optimization*; E.1 **[Data Structures]** *graphs*

General Terms: Algorithms, Theory

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

1. INTRODUCTION

This paper defines a graph structure for representing programs and presents an efficient algorithm for detecting program components (assignment statements and predicates) – in *one or more* programs – that exhibit identical execution behaviors. By the execution behavior of a program component, we mean the sequence of values produced at the component during program execution. For an assignment statement, this means the sequence of values assigned to the target variable; for a predicate, this means the sequence of boolean values produced by the successive evaluations of the predicate.

Although the problem of deciding whether two components have the same execution behaviors is, in general, undecidable, a *safe* approximation can be achieved with reasonable effort. An algorithm for deciding whether two components have the same behaviors is safe if the equivalence classes of components determined by the algorithm are a refinement of the actual equivalence classes of components with the same behavior. For example, in [Reps88, Reps89], it has been shown that two components must have

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 DCR-8603356, 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, and Xerox.

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

the same execution behavior if the program slices[1] taken with respect to the two components are isomorphic. Even if the two components happen to be in *different* programs, the result still holds, as long as the programs are run on identical – or actually just sufficiently similar – initial states.

This paper presents a new algorithm, called the *Sequence-Congruence Algorithm*, for detecting program components that exhibit identical execution behaviors. As with the equivalence-detection algorithm based on comparing slices, the Sequence-Congruence Algorithm can detect components with identical execution behaviors even if the components are in *different* programs. Two such components that are in the same equivalence class have identical execution behaviors whenever the two programs are run on identical -- or sufficiently similar – initial states. The Sequence-Congruence Algorithm is strictly stronger than the method based on comparing slices in that all pairs of components with isomorphic slices are found by the Sequence-Congruence Algorithm as well, but not *vice versa*.

The Sequence-Congruence Algorithm operates on a graph representation of programs. These graphs are called *program representation graphs* in this paper to distinguish them from other related representations. As explained in Section 3, program representation graphs combine features of static-single-assignment forms [Shapiro70, Alpern88, Cytron89, Rosen88] and program dependence graphs [Kuck81, Ferrante87, Horwitz88].

The Sequence-Congruence Algorithm is based on an idea of [Alpern88] for finding equivalence classes of program components by first optimistically grouping possibly equivalent components in an initial partition and then finding the coarsest partition consistent with the initial partition. However, in refining the initial partition, the algorithm of [Alpern88] considers only flow dependences among program components. It is shown in [Alpern88] that components in the same partition produce the same values at *certain moments* during execution. In contrast, the Sequence-Congruence Algorithm given in this paper considers control dependences as well as flow dependences and can detect components whose *execution behaviors* are identical.

A further point of contrast between our work and that of [Alpern88] concerns the idea of applying partitioning to more than one program simultaneously. This idea was not studied in [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 from [Alpern88] is essentially the first phase of our Sequence-Congruence Algorithm, and our first result (the Data-Congruence Lemma) establishes a semantic property of vertices 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 that have identical execution behaviors even though they occur in different programs.

The reason for our interest in detecting components of different programs that have identical execution behaviors is that such information is fundamental to our algorithm for automatic program integration [Horwitz88]. Given a program *Base* and two variants A and B, each created by editing separate copies of *Base*, our program integration algorithm determines whether the changes made to *Base* to produce A and B interfere; if there is no interference, the algorithm produces a merged program M that incorporates the changed behavior of A with respect to *Base*, the changed behavior of B with respect to *Base*, and the unchanged behavior common to *Base*, A, and B.

One of the key issues in program integration is how to determine whether a component of a variant has the same execution behavior as the corresponding component of *Base*. The integration algorithm of

[1]The slice of a program with respect to a program component c is (roughly) all the statements and predicates in the program that can potentially affect the values produced at c during program execution.

[Horwitz88] compares program slices as a safe way of determining whether two program components have identical execution behaviors. The Sequence-Congruence Algorithm provides an alternative method for determining whether two components have the same execution behaviors. It can be shown that the Sequence-Congruence Algorithm finds larger equivalence classes than the method based on comparing program slices. A new algorithm for automatic program integration that makes use of the Sequence-Congruence Algorithm is described in [Yang89].

The remainder of this paper is organized into four sections, as follows: Section 2 describes the programming language under consideration in this paper. Section 3 defines program representation graphs. Section 4 presents the Sequence-Congruence Algorithm and justifies the algorithm. Section 5 discusses how the work reported here relates to previous work.

2. THE PROGRAMMING LANGUAGE UNDER CONSIDERATION

We are concerned with a programming language with the following characteristics: expressions contain only scalar variables and constants; statements are either assignment statements, conditional statements, while-loops, or *end* statements. An *end* statement, which can only appear at the end of a program, names zero or more of the variables used in the program. An example program is shown in the upper left-hand corner of Figure 1.

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; the statements and predicates of a program are executed in the order specified by the program's control flow graph; at any moment there is a single locus of control; the execution of each assignment statement or predicate passes control to a single successor; the execution of each assignment statement changes a global execution state. An execution of the program on an initial state 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 c consists of the value computed when c 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, the final state is defined on only those variables in the *end* statement.

3. PROGRAM REPRESENTATION GRAPH

Program representation graphs (PRGs) combine features of static-single-assignment forms (SSA forms) [Shapiro70, Alpern88, Cytron89, Rosen88] and program dependence graphs [Kuck81, Ferrante87, Horwitz88]. In the SSA form of a program, special assignment statements (ϕ assignments) are inserted so that exactly one assignment to a variable x, either an assignment from the original program or a ϕ assignment, can reach a use of x from the original program. The ϕ statements assign the value of a variable to itself; at most two assignments to a variable x can reach the use of x in a ϕ statement. For instance, consider the following example program fragments:

```
L1    x := 1
      if p then
L2        x := 2
      fi
L4    y := x + 3
```

```
L1    x := 1
      if p then
L2        x := 2
      fi
L3    x := φif(x)
L4    y := x + 3
```

In the source program (on the left), both assignments to x at L_1 and L_2 can reach the use of x at L_3; after the insertion of "$x := \phi_{if}(x)$" at L_3 (on the right), only the ϕ assignment to x can reach the use of x at L_4.

Both assignments to x at L_1 and L_2 can reach the use of x at L_3.

Different definitions of program dependence graphs have been given, depending on the intended application; nevertheless, they are all variations on a theme introduced in [Kuck72], and share the common feature of having an explicit representation of data dependences. The program dependence graph defined in [Ferrante87] introduced the additional feature of an explicit representation for control dependences. The program representation graph, defined below, has edges that represent control dependences and one kind of data dependence, called flow dependence.

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. An example program, its augmented control flow graph, and its program representation graph are shown in Figure 1.

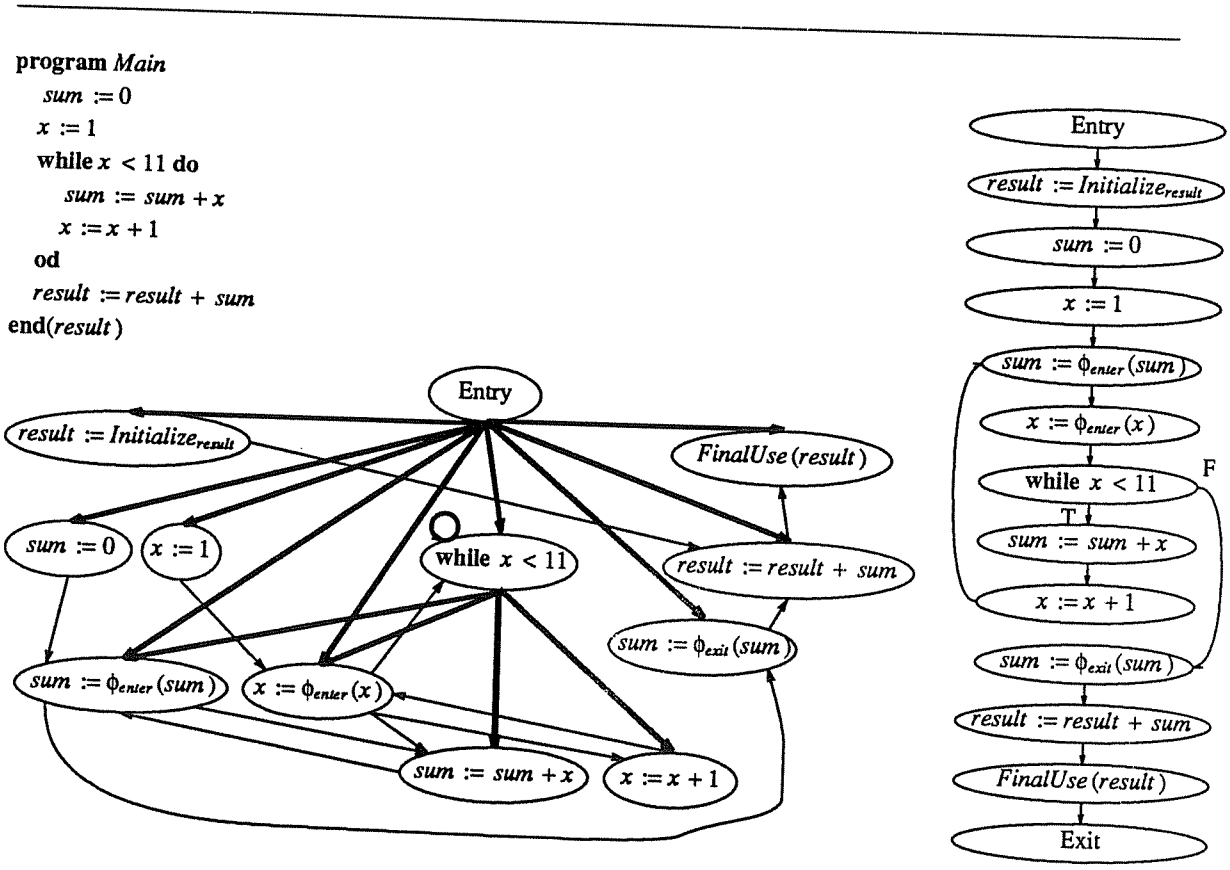

Figure 1. An example program is shown on the top left. This example sums the integers 1 to 10 and adds the sum to the variable *result*. On the right is the augmented control flow graph for the program. Note the absence of *Initialize* and *FinalUse* vertices for *sum* and x and of a ϕ_{exit} vertex for x. On the bottom left is the program representation graph for the program. 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.

Step 1:

The control flow graph[2] of program P is augmented by adding *Initialize*, *FinalUse*, ϕ_{if}, ϕ_{enter}, and ϕ_{exit} vertices, as follows:

(1) A vertex labeled "$x := Initialize_x$" 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 *Initialize* 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 "$x := \phi_{if}(x)$" 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 "$x := \phi_{enter}(x)$" 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.

(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 "$x := \phi_{exit}(x)$" 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, but ϕ_{exit} vertices are placed outside of loops.

Step 2:

Next, the program representation graph is constructed from the augmented control flow graph. The vertices of the program representation graph are those in the augmented control flow graph (except the *Exit* vertex). Edges are of two kinds: control dependence edges and flow dependence edges.

A control dependence edge from a vertex u to a vertex v, denoted by $u \longrightarrow_c v$, means that, during execution, whenever the predicate represented by u is evaluated and its value matches the label – *true* or *false* – on the edge to v, then the program component represented by v will eventually be executed if the program terminates. The source of a control dependence edge is the *Entry* vertex or a predicate vertex.

- 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.

---

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

Note that there is a control dependence edge from a *while* predicate to itself.[3] 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: Except for the extra control dependence edge incident on a ϕ_{enter} vertex, the control dependence edges merely reflect the nesting structure of the program (see [Horwitz88].)

A flow dependence edge from a vertex u to a vertex v, denoted by $u \rightarrow_f v$, means that the value produced at u may be used at v. There is a flow dependence edge $u \rightarrow_f v$ if there is a variable x that is assigned a value at u and used at v, and there is an x-definition-free path from u to v in the augmented control flow graph. The flow dependence edges of a program representation graph are computed using data-flow analysis. For the restricted language considered in this paper, the necessary computations can be defined in a syntax-directed manner (see [Horwitz87]).

The *imported variables* of a program P, denoted by Imp_P, are the variables that might be used before being defined in P, *i.e.*, the variables for which there are *Initialize* vertices in the PRG of P.

Textually different programs may have isomorphic program representation graphs. However, we have shown that if two programs have isomorphic program representation graphs, then the programs are semantically equivalent [Yang89a]:

THEOREM. (EQUIVALENCE THEOREM FOR PROGRAM REPRESENTATION GRAPHS). *Suppose P and Q are programs for which R_P is isomorphic to R_Q. If σ is a state on which P halts, then for any state σ' that agrees with σ on the imported variables of P, (1) Q halts on σ', (2) P and Q compute the same sequence of values at each corresponding program component, and (3) the final states of P and Q agree on all variables for which there are final-use vertices in R_P and R_Q.*

4. THE SEQUENCE-CONGRUENCE ALGORITHM AND THE SEQUENCE-CONGRUENCE THEOREM

In this section we present the Sequence-Congruence Algorithm and a theorem about the equivalence classes of vertices it produces. The Sequence-Congruence Algorithm divides vertices of one or more program representation graphs into disjoint equivalence classes. The theorem states that program components designated by vertices in the same class produce the same sequence of values when the programs are run on sufficiently similar initial states.

This section is divided into three subsections: Section 4.1 presents the Sequence-Congruence Algorithm. Section 4.2 proves the Sequence-Congruence Theorem. Section 4.3 describes three simple enhancements to the Sequence-Congruence Algorithm.

4.1. The Sequence-Congruence Algorithm

The execution behavior of a program component is, by definition, the sequence of values produced at the component during program execution. A component's execution behavior depends on three factors: the operator in the component, the operands available when the operator is applied, and the predicates that control the execution of the operation. It is not unreasonable to expect that vertices with the same operators, equivalent operands, and equivalent controlling predicates will have the same execution behaviors. This expectation is confirmed by the Sequence-Congruence Theorem, which is proved in Section 4.2.

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

The Sequence-Congruence Algorithm consists of two passes. The initial partition puts vertices with the same operators into the same classes. Flow dependence edges (and some additional edges) are used in the first pass to refine the initial partition; in the second pass, control dependence edges are used to further refine the partition obtained from the first pass. Both passes make use of the same partitioning algorithm to refine the partition of the graph's vertices; only the starting partition and the edges considered in the two passes are different.

The operator in a statement or a predicate vertex is determined from the expression part of the vertex. For instance, a statement like "$x := a + b * c$" has the same operator as a statement like "$y := d + e * f$" but a different operator than a statement like "$z := g * h$"; that is, the structure of the expression in the vertex defines the operator. An expression like "$a + b * c$" is viewed as an operator that takes three arguments a, b, and c, and returns the value of "$a + b * c$".

A predicate is *simple* if it consists of a single boolean variable; an assignment statement is simple if its right-hand-side expression consists of a single variable. Both vertices that represent simple predicates and vertices that represent simple assignments are referred to as simple vertices. The operator in a simple vertex is the *identity* operator, that is, an operator that takes one argument and returns the value of the argument. An assignment or a predicate vertex is a *constant vertex* if its expression consists of a single constant. The operator in a constant vertex is the *constant* operator that takes no argument and always returns the value of the constant.

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 in the PRGs. Thus, we can speak of the "corresponding" flow (or control) predecessors of the two vertices. To be more specific, we assign *types* to edges in the PRGs; the notion of corresponding flow (or control) predecessors of two vertices is then defined in terms of the types of edges. (Note that the numbers for the edge types, specified below, are chosen arbitrarily; these numbers are used only to distinguish different types of edges.)

Due to the presence of ϕ vertices in PRGs, each use of a variable in a non-ϕ vertex is reached by exactly one definition (either one of the original assignment statements or one of the ϕ assignments). 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 $1, 2, \ldots, n$, one for each operand. Edge-type numbers for other kinds of edges in a PRG start at $m + 1$, where m is the greatest number of flow edges incident on some non-ϕ vertex. In what follows, we will assume that $m = 3$, and start numbering other edges at 4.

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

```
L1      x := 1
        if p then
L2              x := 2
        else
                skip
        fi
L3      x := φif(x)
```

The definition at L_1 reaches L_3 around the *false* branch of the *if* statement, so the flow dependence edge from L_1 to L_3 has type 5. The definition at L_2 reaches L_3 from the *true* branch, so the flow dependence

edge from L_2 to L_3 has type 4.

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

A vertex u labeled "$x := \phi_{exit}(x)$" 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 8.

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 9. The incoming control dependence edge of a ϕ_{enter} vertex u whose source is *not* the associated *while* predicate for u is assigned type 10 or 11 depending on whether the label on the control dependence edge is *true* or *false*. All other control dependence edges are assigned type 12 or 13 depending on whether the label on the control dependence edge is *true* or *false*.

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

The partitioning algorithm in Figure 2 is adapted from [Alpern88, Aho74], which is based on an algorithm of [Hopcroft71] for minimizing a finite state machine. The m-successors of a vertex u are the vertices v such that there is an edge $u \longrightarrow v$ of type m. The partitioning algorithm finds the coarsest partition of a graph that is consistent with a given initial partition of the graph's vertices; it guarantees that two vertices v and v' are in the same class after partitioning if and only if they are in the same class before partitioning and for any predecessor u of v there is a corresponding predecessor u' of v' such that u and u' are in the same class after partitioning.

Figure 3 presents the Sequence-Congruence Algorithm, which operates on one or more program representation graphs. When the algorithm operates on more than one program's 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 14 and 15, respectively. The initial partition is based on the operators in the vertices. Initially, there is a class for all the non-ϕ vertices that have the same operators. There is a class for all the *Entry* vertices; for each variable x there is a class for all the $Initialize_x$ vertices; for each nesting level of *while* loops, there is a class for all the ϕ_{enter} vertices at this nesting level; there is a class for all the ϕ_{exit} vertices; there is a class for all the ϕ_{if} vertices. The initial partition is refined by the 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 14 and 15 – are discarded at the end of the first pass.)

Pass 2:

The second pass considers only control dependence edges, and applies the partitioning algorithm again to refine the partition obtained from the first pass.

Definition.[4] Vertices are *data-congruent* if they are in the same class after the first pass of partitioning.

```
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 type m of edge 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 2. The partitioning algorithm. This algorithm, which is adapted from [Alpern88, Aho74], finds the coarsest partition of a graph that is consistent with a given initial partition of the graph's vertices. The algorithm guarantees that two vertices v and v' are in the same class after partitioning if and only if they are in the same class before partitioning and for any predecessor u of v there is a corresponding predecessor u' of v' such that u and u' are in the same class after partitioning.

Vertices are *sequence-congruent* if they are in the same class after the second pass of partitioning.

In the worst case, the data-congruence classes can be determined in $O(E_1 \log E_1)$ where E_1 is the number of flow dependence edges plus the number of ϕ_{if} and ϕ_{exit} vertices. The sequence-congruence classes can be determined in $O(E_1 \log E_1 + E_2 \log E_2)$ where E_1 is as above and E_2 is the number of control dependence edges in the graph.

Example. Figure 4 shows an example of partitioning. The initial partition is $<A\,0, B\,0>$, $<A\,1, B\,1>$, $<A\,2, B\,2>$, $<A\,3, B\,3>$, and $<A\,4, B\,4>$. This partition remains unchanged after the first pass of partitioning. After the second pass of partitioning, the final partition is <A0, B0>, <A1>, <B1>, <A2, B2>, <A3, B3>, and <A4, B4>. Note that A1 and B1 are no longer in the same class; thus, $A\,1$ and $B\,1$ are data-congruent but not sequence-congruent. Note also that vertices A4 and B4 are sequence-congruent even

[4]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].

Pass 1:	Add an *if*-edge from every *if* predicate to each associated ϕ_{if} vertex. Add a *while*-edge from every *while* predicate to each associated ϕ_{exit} vertex. The starting partition is based on the operators in the vertices as stated in the text. Apply the partitioning algorithm to refine the initial partition, ignoring all control dependence edges.
Pass 2:	The starting partition is the partition obtained from the first pass. Apply the partitioning algorithm, using only control dependence edges, to further refine the partition.

Figure 3. The Sequence-Congruence Algorithm. The Sequence-Congruence Algorithm consists of two passes. Both passes make use of the partitioning algorithm presented in Figure 2; only the starting partition and the edges considered in the two passes are different.

though their slices are not isomorphic (in this example, the slices with respect to A4 and B4 consist of the respective program fragments in their entirety).

4.2. The Sequence-Congruence Theorem

The Sequence-Congruence Algorithm operates on program representation graphs; the Sequence-Congruence Theorem relates the partitioning operation to the execution behaviors of program components. The Theorem asserts that components designated by sequence-congruent vertices produce the same sequences of values when the programs are run on sufficiently similar initial states. (Since a vertex in a PRG designates a statement or a predicate in the program, in what follows, "a vertex" is used as a synonym for "a statement or a predicate.")

THEOREM. (SEQUENCE-CONGRUENCE THEOREM). *Let P_1 and P_2 be two (not necessarily distinct) 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 halt 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 halts on σ_1 but P_2 does not halt on σ_2, then the sequence of values produced at x_2 is an initial segment of the sequence of values produced at x_1.
(3) If P_1 does not halt on σ_1 but P_2 halts on σ_2, then the sequence of values produced at x_1 is an initial segment of the sequence of values produced at x_2.
(4) If neither P_1 nor P_2 halts 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 an initial segment of the sequence of values produced at x_2, or vice versa.

Since the theorem concerns execution behaviors of program components, we define explicitly the notion of a moment immediately before (or after) an execution step. Statements and predicates of a program P are executed in the order specified by the augmented control flow graph of P. ϕ assignments are considered as statements during program execution. A *moment* immediately before (or after) the execution of a vertex 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:

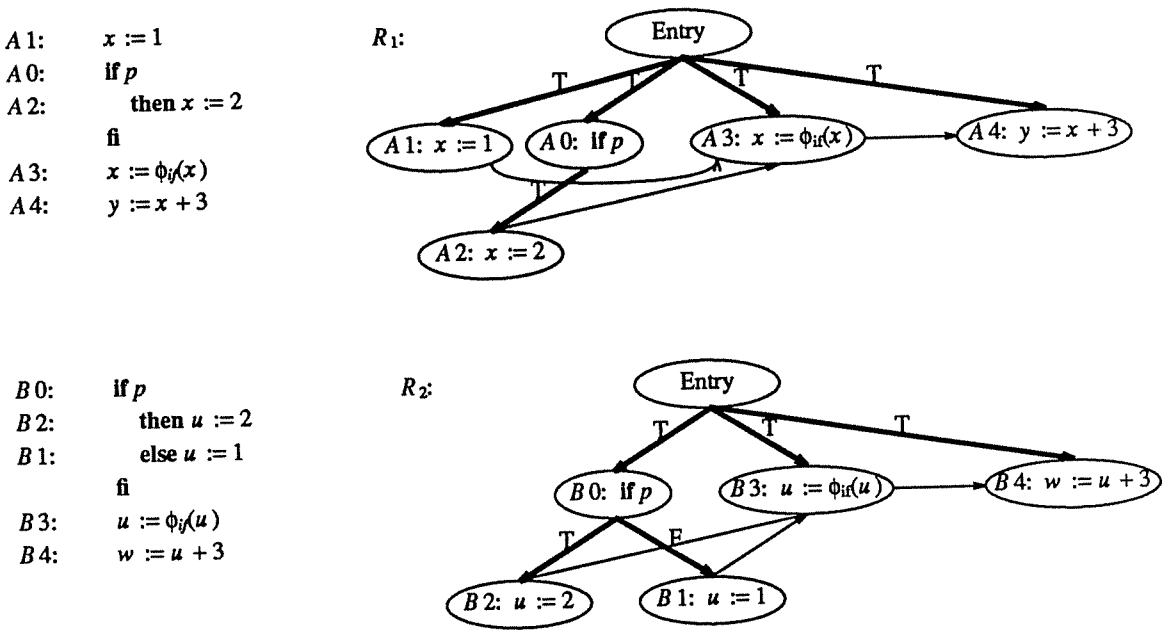

Figure 4. R_1 and R_2 are two fragments of program representation graphs. The incoming flow dependence edges of the two *if* predicates are omitted. The labels on the vertices are included for ease of reference only. The initial partition is $\langle A0, B0\rangle$, $\langle A1, B1\rangle$, $\langle A2, B2\rangle$, $\langle A3, B3\rangle$, and $\langle A4, B4\rangle$. This partition remains unchanged after the first pass of partitioning. After the second pass of partitioning, the final partition is <A0, B0>, <A1>, <B1>, <A2, B2>, <A3, B3>, and <A4, B4>. Note that A1 and B1 are no longer in the same class; thus, $A1$ and $B1$ are data-congruent but not sequence-congruent.

L_1 $x := 0$
L_2 **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 L_1, whereas the locus of control moves *inside* the loop at the moment immediately before the execution of L_2. Note also that the locus of control moves outside the loop at the moment immediately after the *while* predicate L_2 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 at t; hence CLP(t) is the predicate of the associated loop.

A vertex u is an *ancestor* of another vertex v if there is a control-dependence path from u to v in the PRG; thus, *while* predicates are ancestors of the associated ϕ_{enter} vertices. A loop *encloses* a vertex v (or, equivalently, v *is enclosed in* the loop) if the predicate of the loop is an ancestor of 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, there may be one or more *iterations*. There is at least one iteration during an execution of a loop:[5] 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* (defined below) 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 (not necessarily distinct) 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.

Note that *if* predicates are ignored in the above definition. For instance, in Figure 4, if t_1 is the moment immediately after $A1$ executes and t_2 is the moment immediately after $B1$ 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[6] at concurrent moments when the programs run on sufficiently similar initial states.

LEMMA. (DATA-CONGRUENCE LEMMA). *Let P_1 and P_2 be two (not necessarily distinct) 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. 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

[5]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.

[6]The value of a vertex at a moment is the most recent value produced at the vertex before that moment.

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 control and 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 corresponding 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.

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. Because corresponding operands of x_1 and x_2 have the same values at t_1 and t_2, respectively, 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 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. 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 z_1 and z_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. 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. Since the value of x_1 at t_1 is the value of y_1 at t_1' and the value of x_2 at t_2 is the value 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 corresponding 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 corresponding operands of z_1 and z_2 are data-congruent and are active at s_1 and s_2, respectively, corresponding 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 corresponding 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 r_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 r_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 r_1 and r_2, respectively. Therefore, r_1 and r_2 are concurrent. Since y_1 and y_2 are data-congruent, y_1 is active at r_1, y_2 is active at r_2, and r_1 and r_2 are concurrent, y_1 and y_2 must have the same values at r_1 and r_2, respectively, for otherwise we would have chosen r_1, r_2, y_1, and y_2 instead of t_1, t_2, x_1, and x_2. Because the value of x_1 at t_1 is the same as that of y_1 at r_1 and the value of x_2 at t_2 is the same as that of y_2 at r_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 *Initialize* vertex. Since x_1 and x_2 are data-congruent, they must be the *Initialize* 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 Sequence-Congruence Theorem states that sequence-congruent vertices produce the same sequence of values when their programs are run on sufficiently similar initial states.

THEOREM. (SEQUENCE-CONGRUENCE THEOREM). *Let P_1 and P_2 be two (not necessarily distinct) 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 halt 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 halts on σ_1 but P_2 does not halt on σ_2, then the sequence of values produced at x_2 is an initial segment of the sequence of values produced at x_1.
(3) If P_1 does not halt on σ_1 but P_2 halts on σ_2, then the sequence of values produced at x_1 is an initial segment of the sequence of values produced at x_2.
(4) If neither P_1 nor P_2 halts 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 an initial segment of the sequence of values produced at x_2, or vice versa.

PROOF. We prove the theorem by contradiction. We first show that if the theorem is not correct, then the following Proposition must hold:

Proposition. There are two sequence-congruent vertices x_1 and x_2 in P_1 and P_2, respectively, and a constant k such that the k^{th} value produced at x_1 is different from the k^{th} value produced at x_2.

(The Proposition implies that if the Sequence-Congruence Theorem is not correct, then a counterexample occurs in a finite number of steps.) There are four cases to consider; each case corresponds to a clause in the Theorem. In each case, we will show that if the theorem is not correct, then the Proposition holds.

Case 1. Suppose P_1 and P_2 halt on σ_1 and σ_2, respectively. If the theorem is not correct, then the sequences of values produced at x_1 and x_2, respectively, are different. There are two ways in which the sequences of values produced at x_1 and x_2 could be different.

(1) There is a constant k such that the k^{th} value produced at x_1 is different from the k^{th} value produced at x_2. Thus, the Proposition holds.

(2) The sequences of values produced at x_1 and x_2, respectively, are of different lengths and the shorter sequence is an initial segment of the longer one. We may examine the sequences of values produced at corresponding control ancestors of x_1 and x_2. Since the sequences of values produced at x_1 and x_2 are of different lengths and the shorter sequence is an initial segment 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' differs from the k^{th} value produced at x_2'. Thus, the Proposition holds.

Case 2. Suppose P_1 halts on σ_1 but P_2 does not halt on σ_2. If the theorem is not correct, then the sequence of values produced at x_2 is not an initial segment of the sequence of values produced at x_1. There are two ways in which the sequence of values produced at x_2 might not be an initial segment of the sequence of values produced at x_1.

(1) There is a constant k such that the k^{th} value produced at x_1 is different from the k^{th} value produced at x_2. Thus, the Proposition holds.

(2) The sequence of values produced at x_1 is a proper initial segment of the sequence of values produced at x_2. We may examine the sequences of values produced at corresponding control ancestors of x_1 and x_2. Since the sequence of values produced at x_1 is a proper initial segment 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' differs from the k^{th} value produced at x_2'. Thus, the Proposition holds.

Case 3. Suppose P_1 does not halt on σ_1 but P_2 halts on σ_2. This case is similar to *Case 2.*

Case 4. Suppose neither P_1 nor and P_2 halts on σ_1 and σ_2, respectively. There are two cases to consider depending on whether the sequences of values produced at x_1 and x_2 are infinite.

(1) Suppose the sequences of values produced at x_1 and x_2, respectively, are infinite. If the theorem is not correct, the two infinite sequences are not identical. Thus, there must be a constant k such that the k^{th} value produced at x_1 differs from the k^{th} value produced at x_2. Thus, the Proposition holds.

(2) Suppose at least one of the sequences of values produced at x_1 and x_2 is finite. If the theorem is not correct, the sequence of values produced at x_1 is not an initial segment of the sequence produced at x_2, or *vice versa.* Thus, there must be a constant k such that the k^{th} value produced at x_1 differs from the k^{th} value produced at x_2. Thus, the Proposition holds.

We have shown that if the theorem is not correct, then the Proposition holds. Thus, to prove the theorem, it is sufficient to show that the Proposition leads to a contradiction.

Suppose the theorem is not correct. Then, by the argument given above, the Proposition holds. We can find two sequence-congruent vertices, x_1 and x_2, in P_1 and P_2, respectively, and a constant k such that the k^{th} value produced at x_1 is different from the k^{th} value produced at x_2. Let t_1 be the moment immediately after the k^{th} value of x_1 is produced and t_2 be the moment immediately after the k^{th} value of x_2 is produced. There may be many x_1, x_2, and k that satisfy the Proposition. In this case, the ones with the earliest

t_1 and t_2 are chosen.

Because x_1 and x_2 are sequence-congruent, either both x_1 and x_2 are *while* predicates or neither is a *while* predicate. (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.) There are two cases to consider. We will derive a contradiction in each case.

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) are at the same loop nesting levels. Because x_1 and x_2 are sequence-congruent, CLP(t_1) and CLP(t_2) and each pair of corresponding control ancestors of CLP(t_1) and CLP(t_2) are sequence-congruent and hence data-congruent. Because t_1 and t_2 are the earliest moments when the theorem fails, CLP(t_1) and CLP(t_2) and each pair of corresponding control ancestors of CLP(t_1) and CLP(t_2) 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 x_1 and x_2 have different values at t_1 and t_2, respectively.

Case 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, x_1 and x_2 must be at the same loop nesting levels. Because x_1 and x_2 are sequence-congruent, each pair of corresponding control ancestors of x_1 and x_2 are sequence-congruent and hence data-congruent. 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 corresponding 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. Because corresponding flow predecessors of x_1 and x_2 have the same values at t_1' and t_2', respectively, x_1 and x_2 must evaluate to the same values at t_1 and t_2, respectively, which contradicts the assumption that x_1 and x_2 have different values at t_1 and t_2, respectively.

In both cases we have shown a contradiction. Because the Proposition leads to a contradiction, the theorem is proved. □

4.3. Enhancements to the Sequence-Congruence Algorithm

In this subsection 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 $v \rightarrow x$ with an edge $u \rightarrow x$, remove edge $u \rightarrow v$, 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:

```
L1    a := 1                L3    c := 1
L2    b := a+2              L4    d := c
                            L5    e := d+2
```

If we merge the simple assignment statement L_4 with its flow predecessor L_3 before performing the first pass of partitioning, we can discover that L_2 and L_5 are both data-congruent and sequence-congruent. The proofs in the previous section can be directly adapted to account for this change by extending the notion of "corresponding" flow predecessors of two vertices to take simple vertices into account.

In the Sequence-Congruence Algorithm, we assume that a statement like "$x := a + b * c$" has the same operator as a statement like "$y := d + e * f$" but a different operator than a statement like "$z := g * h$"; that is, the structure of the right-hand-side expression defines the operator. An expression like "$a + b * c$" is viewed as an operator that takes three arguments a, b, and c, and returns the value of "$a + b * c$". Thus, in the following program fragment, L_1 and L_2 are not sequence-congruent because they have different operators.

```
L1    x := a + b * c

      z := b * c
L2    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; L_3 and L_4 are found to be sequence-congruent by the Sequence-Congruence Algorithm.

```
      temp := b * c
L3    x := a + temp

      z := b * c
L4    y := a + z
```

Similarly, a constant inside an expression is tightly coupled with the operator. An expression like "$a + 1$" is viewed as a unary operator that takes an argument a and returns the value of "$a + 1$". Therefore, in the following program fragment, L_5 and L_6 are not sequence-congruent because they have different operators (and different number of incoming flow dependence edges).

```
L5    x := a + 1

      z := 1
L6    y := a + z
```

As before, a simple transformation can improve the result of partitioning: (1) for each constant c that appears in the program, 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 partitioning algorithm—the one given in Figure 2—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:

L_1	$a := 1$	L_5	$c := 2$
L_2	$b := 2$	L_6	$d := 1$
L_3	$x := a + b$	L_7	$u := c + d$
L_4	$y := x * 3$	L_8	$v := u * 3$

With the present algorithm for Pass 1, L_3 and L_7 are eventually placed in separate data-congruence classes, and hence L_4 and L_8 are also placed in separate data-congruence classes. However, because addition is commutative, L_3 and L_7 could be placed in a single equivalence class, which then also makes it possible for L_4 and L_8 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, L_3 and L_7 would be members of one sequence-congruence class, and L_4 and L_8 would be members of another.

5. RELATION TO PREVIOUS WORK

Program representation graphs combine features of both program dependence graphs [Kuck81, Ferrante87, Horwitz88] and static-single-assignment forms [Alpern88, Rosen88, Cytron89] (especially the value graph associated with an SSA form [Alpern88]). Although program dependence graphs contain no ϕ vertices they do contain additional data-dependence edges not found in program representation graphs. For example, the program dependence graphs used in [Horwitz88] contain def-order dependence edges. We have shown elsewhere that PRGs and the program dependence graphs of [Horwitz88] are equivalent program representations in the sense that two programs have isomorphic PRGs if and only if their program dependence graphs are isomorphic [Yang89a]. In essence, the ϕ vertices of program representation graphs introduce extra flow dependence edges that substitute for def-order dependence edges.

There are two main differences between our PRGs and (the value graphs of) SSA forms:

(1) PRGs contain control dependence edges, whereas SSA forms do not. Control dependence edges were added so that the Sequence-Congruence Algorithm could take control dependences into account during partitioning.

(2) The ϕ statements in PRGs are slightly different from those in SSA forms. In the SSA 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 program 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.[7]

Another difference between the ϕ statements in PRGs and those in SSA forms is that PRGs include only ϕ statements whose left-hand side variable is live (*i.e.* every ϕ statement has a flow successor). For instance, consider the following program fragment (which is augmented with ϕ statements for the ϕ vertices of its PRG):

```
        a := 0
L1      x := 1
        if p then
                x := 2
                a := x
        fi
        a := φif(a)
L2      x := 3 + a
```

If the PRG were to include non-live ϕ statements, there would be a ϕ statement "$x := \phi_{if}(x)$" immediately after the *if* statement; however, this ϕ_{if} statement is not included in the PRG for this program because the variable x is defined before being used after the *if* statement. The reasons why we excluded these extra ϕ statements from PRGs are not directly relevant to the questions addressed in this paper; however, to summarize briefly, our decisions were motivated by the following concerns:

(1) The exclusion of non-live ϕ statements permits larger sets of semantically equivalent programs to have the same PRG. For example, the same PRG represents not only the program shown above, but also a version of the program in which L_1 comes *after* the *if* statement (but before L_2). As shown elsewhere [Yang89a], PRGs and the program dependence graphs of [Horwitz88] are equivalent program representations in the sense that two programs have isomorphic PRGs if and only if their program dependence graphs 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) A useful operation on PRGs is that of *slicing*: The slice of a PRG R with respect to a set of non-ϕ vertices S is the subgraph of R induced by all vertices from which there is a path to an element of S via control and/or flow dependence edges in R. We wished to have the property that any slice of a PRG would be (isomorphic to) the PRG of some program. (This corresponds to a similar property that holds for slices of program dependence graphs — the Feasibility Lemma of [Reps88].) For instance, if non-live ϕ vertices were required in PRGs, the example given above would be

```
        a := 0
L1      x := 1
        if p then
                x := 2
                a := x
        fi
        x := φif(x)
        a := φif(a)
L2      x := 3 + a
```

Its slice with respect to L_2 would correspond to the fragment

[7] It has been recognized by others that variable renaming is not necessary for all uses of SSA forms [Alpern88a].

```
        a := 0
        if p then
                x := 2
                a := x
        fi
        a := φif(a)
L2      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 $x := \phi_{if}(x)$ just after the *if* statement, it does *not* correspond to the program

```
        a := 0
        if p then
                x := 2
                a := x
        fi
L2      x := 3 + a
```

By excluding non-live ϕ statements from PRGs, infeasible slices do not arise; every slice of a PRG is (isomorphic to) the PRG of some program [Yang89a].

The Sequence-Congruence Algorithm is based on an idea of [Alpern88] for finding equivalence classes of program components by first optimistically grouping possibly equivalent components in an initial partition (of an SSA form's value graph) and then finding the coarsest partition of the graph's vertices that is consistent with the initial partition. The algorithm of [Alpern88] considers only flow dependences among program components. The property established in [Alpern88] shows that components of a *single* program in the same partition produce the same values at *certain moments* during execution. Stated using our terminology: "If two data-congruent components are both active at some moment, they have produced the same values."

In contrast to [Alpern88], our Sequence-Congruence Algorithm has two important properties: (1) By considering control dependences as well as data dependences, it is able to detect components with equivalent *execution behaviors*, and (2) it is able to do so even if the components are in *different* programs.

The algorithm of [Alpern88] is essentially the first pass of our Sequence-Congruence Algorithm. An important difference is that our algorithm can be applied to *one or more* programs. Our Data-Congruence Lemma establishes a semantic property for components that are in the same partition after the first pass: "If two data-congruent components are active at concurrent moments when the programs are run on sufficiently similar initial states, they have produced the same values." It is the concept of concurrent moments that makes it possible to compare the executions of more than one program.

The second pass of our Sequence-Congruence Algorithm uses control dependences to refine the partitioning produced by the first pass. Our Sequence-Congruence Theorem establishes a semantic property about the overall execution behaviors of congruent components (rather than about their behaviors at certain moments): "If two components are sequence-congruent, they have identical execution behaviors whenever the programs they are in are run on sufficiently similar initial states."

Our work on program representation graphs and the Sequence-Congruence Algorithm has been motivated by the desire to improve our techniques for automatic program integration [Horwitz88]. The goal of such work is to create a tool that can automatically determine whether changes that have been made in several variants of a base program interfere; if there is *no* interference, the tool should produce a new program that combines the changed behaviors of the variants as well as the behaviors common to the base program and all variants. A program integration tool is needed, for example, when a number of collabora-

tors are collectively producing updates in a large programming project.

One of the key issues of program integration is to determine whether the execution behavior of a program component in one of the variants differs from that of the corresponding component in the base program. In the integration algorithm of [Horwitz88], this is done by finding corresponding program components whose slices are not isomorphic. The justification for this approach is found in [Reps88] where (for the same language considered in this paper) it is shown that program components with isomorphic slices have identical execution behaviors.

The Sequence-Congruence Algorithm given in this paper provides a different method for determining whether program components in two programs have different execution behaviors. Rather than comparing slices, the Sequence-Congruence Algorithm starts from an optimistic assumption about which program components may exhibit identical execution behaviors and then refines this assumption by considering flow and control dependences. The justification for this approach is provided by the Sequence-Congruence Theorem, which shows that sequence-congruent components have identical execution behaviors. It is easy to see that the equivalence classes found by the Sequence-Congruence Algorithm are strictly larger than those found by comparing slices.

We have devised a new algorithm for program integration, described in [Yang89], that makes use of the Sequence-Congruence Algorithm. This algorithm has several advantages when compared with the integration algorithm described in [Horwitz88]. One advantage concerns the ability of users to rename variables: because the algorithm from [Horwitz88] detects changed execution behavior by comparing program slices, all program elements that depend on a variable whose name has been changed will be considered to have different execution behavior. This increases the likelihood that the integration algorithm will report that two users' modifications are in conflict. In contrast, the Sequence-Congruence Algorithm will determine that such program elements will not have different execution behaviors. Thus, the integration algorithm that uses the Sequence-Congruence Algorithm is able to avoid reporting some of the spurious conflicts that arise when a variable's name is changed.

Variable renaming is an example of one kind of meaning-preserving modification — one that does not introduce any structural changes to a program. In addition to being able to detect that two components have the same execution behavior in spite of variable renaming, the Sequence-Congruence Algorithm is also able to detect that two components have the same execution behavior for a limited class of meaning-preserving structural changes. Thus, a second advantage of the new integration algorithm is that it can accommodate a limited class of meaning-preserving structural changes and not report some of the spurious conflicts that would be reported by the algorithm from [Horwitz88].

ACKNOWLEDGEMENTS.

Thanks are due to Bowen Alpern for his participation in early discussions that led to the results reported in this paper.

REFERENCES

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).

Alpern88a.
Alpern, B., Personal communication, September 1988.

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, January 13-15, 1988), ACM, New York (January 1988).

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).

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).

Hopcroft71.
Hopcroft, J.E., "An *n log n* algorithm for minimizing the states of a finite automaton," *The Theory of Machines and Computations*, pp. 189-196 (1971).

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).

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, January 13-15, 1988), ACM, New York (January 1988).

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, January 26-28, 1981), ACM, New York (1981).

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," pp. 360-374 in *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*, Vol. 352, Springer-Verlag, New York, NY (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, January 13-15, 1988), ACM, New York (January 1988).

Shapiro70.
Shapiro, R.M. and H. Saint, "The representation of algorithms," Technical Report CA-7002-1432, Massachusetts Computer Associates (February 1970). as cited in [Alpern88, Rosen88]

Yang89.
Yang, W., S. Horwitz, and T. Reps, "A new program integration algorithm," Technical Report in preparation, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (Spring 1989).

Yang89a.
Yang, W., S. Horwitz, and T. Reps, "Program representation graphs: Semantic properties and relationship to program dependence graphs," Technical Report in preparation, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI (Summer 1989).