INTEGRATING NON-INTERFERING VERSIONS OF PROGRAMS

by

Susan Horwitz
Jan Prins
Thomas Reps

Computer Sciences Technical Report #690

March 1987

Integrating Non-Interfering Versions of Programs

SUSAN HORWITZ, JAN PRINS, and THOMAS REPS
University of Wisconsin – Madison

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. Anyone who has had to reconcile divergent lines of development will recognize the problem and identify with the need for automatic assistance. This paper concerns the design of a tool for automatically integrating program versions. The main contribution of the paper is an algorithm, called *Integrate*, that takes as input three programs A, B, and *Base*, where A and B are two variants of *Base*. Whenever the changes made to *Base* to create A and B do not "interfere" (in a sense defined in the paper), Integrate produces a program M that integrates A and B.

The method is based on the assumption that any change in the *behavior*, rather than the *text*, of *Base*'s variants is significant and must be preserved in M. Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with *Base*. To determine this information, we employ a program representation that is similar, but not identical, to the *program dependence graphs* that have been used previously in vectorizing compilers.

To the best of our knowledge, the code-integration problem has not been previously formalized. It should be noted, however, that the integration problem examined here is a greatly simplified one; in particular, we assume that expressions contain only scalar variables and constants, and that the only statements used in programs are assignment statements, conditional statements, and while-loops.

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

General Terms: Algorithms, Design

Additional Key Words and Phrases: program integration, non-interfering versions, propagating enhancements, program dependence graph, program slicing, data-flow analysis, control dependency, data dependency

---

1. INTRODUCTION

Programmers are often faced with the task of integrating several related, but slightly different variants of a system. One of the ways in which this situation arises is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating several related versions with slightly different features. If one wishes to create a new version that incorporates several of the enhancements simultaneously, one has to check for conflicts in the implementations of the different versions and then merge them to create an integrated version that combines their separate features.

This work was supported in part by the National Science Foundation under grants DCR-8552602 and DCR-8603356 as well as by grants from IBM, DEC, Siemens, and Xerox.

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

The task of integrating different versions of programs also arises as systems are being created. Program development is usually a cooperative activity that involves multiple programmers. If a task can be decomposed into independent pieces, the different aspects of the task can be developed and tested independently by different programmers. However, if such a decomposition is not possible, the members of the programming team must work with multiple, separate copies of the source files, and the different versions of the files have to be merged into a common version.

The program-integration problem also arises in a slightly different guise when a family of related versions of a program has been created (for example, to support different machines or different operating systems), and the goal is to make the same enhancement or bug-fix to all of them. Such a change cannot be developed for one version and blindly applied to all other versions since the differences among the versions might alter the effects of the change.

Anyone who has had to reconcile divergent lines of development will recognize these situations and identify with the need for automatic assistance. Unfortunately, at present, the only available tools for integration are variants of differential file comparators, such as the Unix utility *diff*. The problem with such tools is that they implement an operation for merging files as strings of text. For instance, *diff* implements a heuristic algorithm for file comparison based on the longest common subsequence problem [Hunt & McIlroy].

This approach has the advantage that the current tools are as applicable to merging documents, data files, and other text objects as they are to merging programs. Unfortunately, these tools are necessarily of limited utility for integrating programs because the manner in which two programs are merged is not *safe*. One has no guarantees about the way the program that results from a purely *textual* merge behaves in relation to the behavior of either of the two programs that are the arguments to the merge. The merged program must, therefore, be checked carefully for conflicts that might have been introduced by the merge.

This paper describes a technique that could serve as the basis for building an automatic program integration tool. We present an algorithm *Integrate* that takes as input three programs A, B, and *Base*, where A and B are two variants of *Base*. Integrate either determines that the changes made to *Base* to produce A and B interfere, or it produces a new program M that integrates A and B with respect to *Base*.

The method is based on the assumption that any change in the *behavior*, rather than the *text*, of *Base*'s variants is significant and must be preserved in M. Although it is undecidable to determine whether a program modification actually leads to a change in behavior, it is possible to determine a safe approximation by comparing each of the variants with *Base*. To determine this information, we adopt (and adapt) the *program dependence graphs* that have been used previously for representing programs in vectorizing compilers [Kuck et al. 1981, Allen & Kennedy 1982, 1984, Ferrante et al. 1987].

To the best of our knowledge, the program-integration problem has not been previously formalized. It should be noted, however, that the integration problem examined here is a greatly simplified one; in particular, the algorithm Integrate operates under the simplifying assumptions that expressions contain only scalar variables and constants, and that the statements used in programs consist of assignment statements, conditional statements, while-loops, and no other control constructs.

The paper is organized into seven sections, as follows: Section 2 discusses criteria for integratability and interference. Section 3 illustrates some of the problems that can arise when programs are integrated using textual comparison and merging operations.

Sections 4.1 through 4.5 describe the five steps of the algorithm Integrate. The first step, described in Section 4.1, builds the program dependence graphs that represent the programs *Base*, A, and B. The program dependence graph that represents program P is denoted by G_P. The second step, discussed in Sec-

tion 4.2, determines sets of *affected program points* of G_A and G_B as computed with respect to G_{Base}. Each set captures the essential differences between *Base* and the variant program. The third step, described in Section 4.3, combines G_A and G_B to create a merged dependence graph G_M, making use of the sets of affected program points that were computed by the second step. The fourth step uses G_A, G_B, the affected points of G_A and G_B, and G_M to determine whether A and B interfere with respect to *Base*; interference is defined and discussed in Section 4.4. The fifth step, which is carried out only if A and B do not interfere, determines whether G_M corresponds to some program and, if it does, creates an appropriate program from G_M. Although, as we show in Appendix C, the problem of determining whether a program dependence graph corresponds to some program is NP-complete, we conjecture that the backtracking algorithm given for this step in Section 4.5 will behave satisfactorily on actual programs. The algorithm Integrate is summarized in Section 4.6.

Section 5 discusses how Integrate may be applied to the problem of propagating enhancements and bug-fixes through related program versions. Section 6 describes some technical differences between the kind of program dependence graphs we employ and the program dependence representations that have been defined by others. Section 7 discusses some areas for future work and possible extensions to our method to permit it to be applied to more realistic programming languages.

2. CRITERIA FOR INTEGRATABILITY AND INTERFERENCE

Two versions A and B of a common *Base* may, in general, be arbitrarily different. To describe the integrated version M we could say that the developers of A and B each have in mind their own *specification*, and that M should be constructed so as to satisfy *both* specifications. For example, following the view of specifications as pairs of pre- and post-condition predicates [Hoare 1969, Dijkstra 1976], given programs A and B that satisfy $\{P_A\}\ A\ \{Q_A\}$ and $\{P_B\}\ B\ \{Q_B\}$, respectively, A and B are integratable if there exists a program M such that $\{P_A\}\ M\ \{Q_A\}$ and $\{P_B\}\ M\ \{Q_B\}$.

Under certain circumstances, it is not possible to integrate two programs; we say that such programs *interfere*. One source of interference for the integration criterion given above can be illustrated by restating the criterion as follows: M integrates A and B if M satisfies the three triples $\{P_A \wedge P_B\}\ M\ \{Q_A \wedge Q_B\}$, $\{P_A \wedge \neg P_B\}\ M\ \{Q_A\}$, and $\{P_B \wedge \neg P_A\}\ M\ \{Q_B\}$. A and B interfere if the formula $P_A \wedge P_B$ is satisfiable but $Q_A \wedge Q_B$ is unsatisfiable; under this circumstance, it is impossible to find an M that satisfies the specification $\{P_A \wedge P_B\}\ M\ \{Q_A \wedge Q_B\}$.

An integration criterion based on program specifications leaves a great deal of freedom for constructing a suitable M, but would be plagued by the familiar undecidable problems of automated program derivation. Moreover, the requirement that programs be annotated with specifications would make such an approach unusable with the methods of system development currently in use. Consequently, we investigate a considerably restricted definition of the program-integration problem and devise appropriate interference criteria.

Our basic assumption is that the integrated version M must be composed of exactly the statements and control structures that appear as components of A and B. In conjunction with this assumption, we assume that the editor used to create *Base*, A, and B provides a unique-naming capability so that statements and predicates are identified consistently in all three versions. Each component that occurs in a program is an object whose identity is recorded by a unique identifier that is guaranteed to persist across different editing sessions and machines. For example, a component that is moved from its original position in *Base* to a new position in A retains its identity from *Base*.

As for the interference criterion, since we do not know the specification of any of the versions of the program, we assume that any change to version A or B that could lead to a different behavior than the behavior of *Base* is significant and must be preserved in M. Although it is undecidable to determine

whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with *Base*. To determine this information, we make use of *program dependence graphs* [Kuck et al. 1981, Allen & Kennedy 1982,1984, Ferrante et al. 1987]; we also make use of the notion of a *program slice* [Weiser 1984, Ottenstein & Ottenstein 1984] to determine just those statements of a program that determine the values of (potentially) affected variables. (In both cases, these ideas have been adapted to the particular needs of the program-integration problem).

Although we restrict our attention in this paper to a programming language with only the most basic control constructs, we are hopeful that our approach to program integration can be extended to more realistic programming languages. Our assumptions about the language have been made to simplify the program-integration problem to a manageable level. When the program-integration problem is extended beyond our simplified setting, certain details of the integration method will become more complicated. For instance, the data-flow problems whose solutions are necessary to support our approach may become more complex; however, similar problems have already been addressed in [Weiser 1984, Ferrante et al. 1987], and they are not insurmountable.

An additional goal for an integration tool, although one of secondary importance, is ensuring that the program M that results from integrating A and B resembles A and B as much as possible. While on the subject of our method's limitations, it is only fair to point out that that it is notably weak in this area. For example, when the final step of the integration algorithm finally determines the order of statements in M, it does not make direct use of the order in which statements occur in A or B. Consequently, it may not preserve original statement order, even in portions of the programs that are unaffected by the changes made to the base program to create A and B. Our integration method *does* preserve the original variable names used in A, B, and *Base*; however, as discussed briefly in Section 4.5, it may be necessary to abandon this property and permit the final step of the integration algorithm to perform a limited amount of variable renaming.

3. THE PERILS OF TEXT-BASED INTEGRATION

Integrating programs via textual comparison and merging operations is accompanied by numerous hazards. This section describes some of the problems that can arise, and underscores them with an example that baffles the Unix program *diff3*. (*Diff3* is a relative of *diff* that can be used to create a merged file when supplied a base file and two variants).

One problem is that character- or line-oriented textual operations do not preserve syntactic structure; consequently, a processor like *diff3* can easily produce something that is syntactically incorrect. Even if the problem of syntactically erroneous output were overcome, there would still be severe drawbacks to integration by textual merging because text operations do not account for program semantics. This has two undesirable consequences:

a) If the variants of the base program do interfere (under a semantic criteria), *diff3* still goes ahead and produces an "integrated" program.

b) Even when the variants do not interfere (under semantic criteria), the integrated program created using *diff3* is not necessarily an acceptable integration.

The latter problem is illustrated by the example given below. In this example, *diff3* creates an unacceptable integrated program despite the fact that it is only necessary to reorder (whole) lines to produce an acceptable one. The example concerns the following base program and two variants:

Base program

```
if P then x := 0 fi
if Q then x := 1 fi
y := x
if R then w := 3 fi
if S then w := 4 fi
z := w
```

Variant *A*

```
if Q then x := 1 fi
if P then x := 0 fi
y := x
if R then w := 3 fi
if S then w := 4 fi
z := w
```

Variant *B*

```
if S then w := 4 fi
if R then w := 3 fi
z := w
if P then x := 0 fi
if Q then x := 1 fi
y := x
```

In variant *A*, the conditional statements that have *P* and *Q* as their conditions are reversed from the order in which they appear in *Base*. In variant *B*, the order of the *P*–*Q* pair remains the same as in *Base*, but the order of the *R*–*S* pair is reversed; in addition, the order of the first and second groups of three statements have been interchanged.

Under Unix, a program that (purportedly) integrates *Base*, *A*, and *B* can be created by the following operations:

```
diff3 –e A Base B > script
(cat script; echo '1,$p') | ed – A
```

The first command invokes the three-way file comparator *diff3*; the –e flag of *diff3* causes it to create an editor script as its output. This script can be used to incorporate in one of the variants (in this case, *A*) changes between the base program (*Base*) and the second variant (*B*). The second command invokes the editor to apply the script to variant *A*.

The program that results from these operations is:

```
if S then w := 4 fi
if R then w := 3 fi
z := w
if P then x := 0 fi
if Q then x := 1 fi
y := x
```

This program is exactly the same as the one given as variant *B*. Because it does not account for the differences in behavior between *Base* and variant *A*, this can hardly be considered an acceptable integration of *Base*, *A*, and *B*.

We now try a different tactic and exchange the positions of *A* and *B* in the argument list passed to *diff3*, thereby treating *B* as the "primary" variant and *A* as the "secondary" variant (*diff3* is not symmetric in its first and third arguments). The program that results is:

```
if Q then x := 1 fi
if P then x := 0 fi
y := x
```

Clearly this program is unacceptable as the integration of *Base*, *A*, and *B*.

This example illustrates the use of *diff3* to create an editing script that merges three documents whether or not there are "conflicts." It is also possible to have *diff3* produce an editing script that annotates the merged document at places where conflicts occur. At such places, the script inserts both versions of the

text, and brackets the region of the conflict by "<<<<<<" and ">>>>>>." For instance, the outcome for the second case discussed above is:

```
<<<<<<< B
if S then w := 4 fi
if R then w := 3 fi
z := w
if P then x := 0 fi
=======
>>>>>>> A
if Q then x := 1 fi
if P then x := 0 fi
y := x
```

When we apply the program-integration method that is described in this paper to this same example, there are several programs it might create, including the following three:

```
if S then w := 4 fi          if Q then x := 1 fi          if Q then x := 1 fi
if R then w := 3 fi          if P then x := 0 fi          if P then x := 0 fi
z := w                       y := x                       if S then w := 4 fi
if Q then x := 1 fi          if S then w := 4 fi          if R then w := 3 fi
if P then x := 0 fi          if R then w := 3 fi          y := x
y := x                       z := w                       z := w
```

In contrast with the programs that result from text-based integration, any of the algorithm's possible products is a satisfactory outcome for integrating *Base*, *A*, and *B*. We return to this example in Section 5, and use it to illustrate the operation of the program-integration algorithm.

4. AN ALGORITHM FOR INTEGRATING NON-INTERFERING VERSIONS OF PROGRAMS

4.1. The Program Dependence Graph

The information used by our program integration method is encapsulated by a program representation called a *program dependence graph*. Program dependence graphs were introduced by Kuck [Kuck et al. 1972, Towle 1976, Kuck 1978, Kuck et al. 1981] and a number of variations have since appeared [Allen & Kennedy 1982,1984, Ottenstein & Ottenstein 1984, Ferrante et al. 1987]. The definition presented here is yet another variation, adapted to the particular needs of the program-integration problem.

The program dependence graph for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges[1]. The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

a) There is a distinguished vertex called the *entry vertex*.

b) For each variable x used in P, there is a vertex called the *initial definition of* x. This vertex represents an initial assignment to x with a special value **uninitialized**.

c) For each variable used in P, there is a second vertex called the *final use of* x. It represents an access to the final value of x computed by P.

The edges of G_P represent *dependencies* among program components. An edge represents either a *control dependency* or a *data dependency*. Control dependency edges are labeled either **true** or **false**, and the

[1] A *directed graph* G consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b,c) \in E(G)$ is directed from b to c; we say that b is the *source* and c the *target* of the edge. Throughout the paper, the term "vertex" is used to refer to elements of dependency graphs, whereas the term "node" refers to elements of derivation trees.

source of a control dependency edge is always the entry vertex or a predicate vertex. A control dependency edge from vertex v_1 to vertex v_2, denoted $v_1 \rightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately). A method for determining control dependency edges for arbitrary programs is given in [Ferrante et al. 1987]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependency edges of G_P can be determined in a much simpler fashion. For the language under consideration here, a program dependence graph contains a *control dependency edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

i) v_1 is the entry vertex, and v_2 represents a component of P that is not subordinate to any control predicate.

ii) v_1 represents a control predicate, and v_2 represents a component of P immediately subordinate to the control construct whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \rightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \rightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.

Note that there are no control dependencies to initial definitions and final uses of variables.

A data dependency edge from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. In this paper, program dependence graphs contain two kinds of data-dependency edges, representing *flow dependencies* and *def-order dependencies*.

A program dependence graph contains a flow dependency edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 is an assignment statement that defines variable x.

ii) v_2 is an assignment statement or predicate that uses x.

iii) Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control-flow graph for the program [Aho et al. 1986] by which the definition of x at v_1 reaches the use of x at v_2. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph, and final uses of variables are considered to occur at its end).

A flow dependency that exists from vertex v_1 to vertex v_2 will be denoted by $v_1 \rightarrow_f v_2$.

Flow dependencies can be further classified as *loop independent* or *loop carried*. A flow dependency $v_1 \rightarrow_f v_2$ is loop independent, denoted $v_1 \rightarrow_{li} v_2$, if the execution path by which v_2 is reached from v_1 includes no backedge of the control-flow graph; otherwise, it is a loop carried dependency. A loop-carried dependency edge is labeled with the loop that carries the dependence; that is, if the execution path by which v_2 is reached from v_1 includes a backedge to the predicate of loop L (in the control-flow graph), then the edge from v_1 to v_2 is labeled with L. Such a dependency is denoted by $v_1 \rightarrow_{lc(L)} v_2$.

A program dependence graph contains a def-order dependency edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 and v_2 both define the same variable.

ii) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

iii) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.

iv) v_1 occurs to the left of v_2 in the program's abstract syntax tree.

A def-order dependency from v_1 to v_2 is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$.

Note that a program dependence graph is a multi-graph (*i.e.* it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependency edge between two vertices, each is labeled by a different loop that carries the dependency. When there is more than one def-order edge between two vertices, each is labeled by a vertex that is flow-dependent on both the definition that occurs at the edge's source and the definition that occurs at the edge's target.

Example. Figure 1 shows an example program and its program dependence graph. The boldface arrows represent control dependency edges; dashed arrows represent def-order dependency edges; solid arrows

---

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

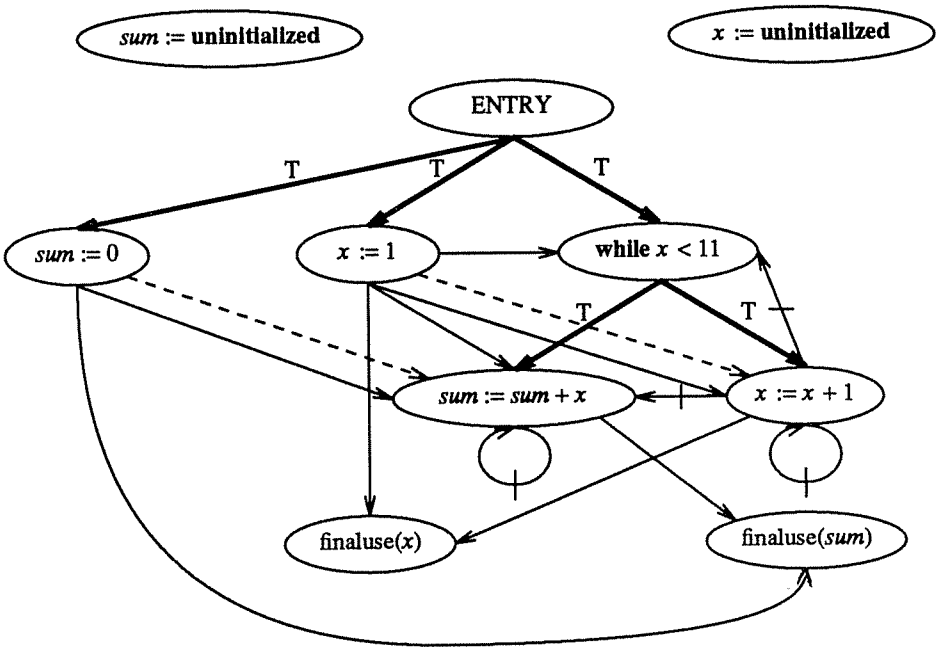


Figure 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependency edges, dashed arrows represent def-order dependency edges, solid arrows represent loop-independent flow dependency edges, and solid arrows with a hash mark represent loop-carried flow dependency edges.

represent loop-independent flow dependency edges; solid arrows with a hash mark represent loop-carried flow dependency edges.

The function GeneratePDG, given in Figure 2, creates a program dependence graph (a PDG) for a given program. The data-dependency edges of the PDG 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; these methods are described in detail in Appendix A.

We shall assume that elements of PDG's are also labeled with some additional pieces of information. Recall that we have assumed that the editor used to modify programs provides a unique-naming capability so that statements and predicates are identified consistently in different versions. Each component that occurs in a program is an object whose identity is recorded by a unique identifier that is guaranteed to persist across different editing sessions and machines. It is these identifiers that are used to determine "identical" vertices when we perform operations on components from different PDG's (*e.g.* $V(G') \cap V(G)$).

4.1.1. Program dependence graphs and program behavior

In choosing which dependency edges to include in our program dependence graphs our goal has been to partially characterize programs that have the same behavior. In particular, two inequivalent programs should not have the same program dependence graph, although two equivalent programs may have different program dependence graphs.

We can illustrate the need for each of the different kinds of edges included in our definition by demonstrating some sample inequivalent programs that would be indistinguishable if PDG's were to lack a particular kind of edge. For example, the distinction between loop-independent and loop-carried flow dependencies is necessary to distinguish between the following two program fragments:

```
function GeneratePDG(P) returns a program dependence graph
declare
   P: an abstract syntax tree representation of a program
   G: a program dependence graph
begin
   V(G) := the set of assignment statements, if predicates, and while predicates of P, an entry vertex,
           for each variable in P an initial definition and a final use
   E(G) := ∅
   Traverse P and insert control dependency edges into G
   Compute loop-carried and loop-independent reaching definitions for each node of P (see Appendix A)
   Traverse P and insert loop-independent and loop-carried flow dependency edges into G
   Compute def-order dependencies for each node of P (see Appendix A)
   Traverse P and insert def-order dependency edges into G
   return(G)
end
```

Figure 2. The function GeneratePDG(P) creates the program dependence graph for program P.

```
x := 0                          x := 0
while P do                      while P do
   y := x                          if Q then x := 1 fi
   if Q then x := 1 fi             y := x
od                              od
```

The PDG's for these fragments have identical vertices, control dependency edges, and def-order dependency edges. If we ignore the distinction between loop-independent and loop-carried flow dependencies, they have identical flow dependency edges as well; however, in the left-hand fragment, the flow dependency from the assignment statement $x := 1$ to the assignment $y := x$ is a loop-carried dependency, whereas the corresponding dependency in the right-hand fragment is a loop-independent one.

Def-order dependencies are needed in PDG's to be able to distinguish between the program fragments:

```
if P then x := 0 fi             if Q then x := 1 fi
if Q then x := 1 fi             if P then x := 0 fi
y := x                          y := x
```

Here the PDG's for these fragments have identical vertices, control dependency edges, and flow dependency edges. If PDG's did not contain def-order dependency edges, these programs would have identical PDG's, although they do not have equivalent behaviors. Including def-order dependences causes them not to have identical PDG's; in the left-hand fragment, there is a def-order dependency from the assignment statement $x := 0$ to the assignment $x := 1$, whereas in the right-hand fragment, the def-order dependency runs in the other direction, from $x := 1$ to $x := 0$.

4.1.2. Program slices

For a vertex s of a PDG G, the *slice* of G with respect to s, written as 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 flow or control 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:

$$V(G / S) = V(G / (\bigcup_i s_i)) = \bigcup_i V(G / s_i)$$

It is useful to define $V(G / v) = \varnothing$ for any $v \notin G$.

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 included only if G / S 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 define

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

We say that G is a *feasible* program dependence graph iff G is the program dependence graph of some program P. For any $S \subseteq V(G)$, if G is a feasible PDG, the slice G / S is also a feasible PDG (although it may omit some final-use vertices); it corresponds to the program P' obtained by restricting the syntax tree of P to just the statements and predicates in $V(G / S)$ [Ottenstein & Ottenstein 1984].

The significance of a slice is that it captures a portion of a program's behavior. The programs P' and P, corresponding to the slice G / S and the graph G, respectively, "compute the same values" at each program point $s \in S$. In our case a program point may be (1) an assignment statement, (2) a control predicate, or (3)

a final use of a variable. Because a statement or control predicate may be reached repeatedly in a program, by "computing the same value" we mean (respectively) (1) for any assignment statement in S, the same *sequence* of values are assigned to the target variable, (2) for a predicate in S, the same *sequence* of boolean values are produced, and (3) for each final use in S, the same value for the variable is produced.

4.2. Determining the Differences in Behavior of a Variant

In this section we characterize (an approximation to) the difference between the behavior of *Base* and its variants. Since we do not know the specification of *Base* or its variants, we assume that *any* and *only* changes in behavior of a variant with respect to *Base* are significant. The program dependence graphs are a convenient representation from which to determine these changes. (The other reason for working with the PDG representations is that they are simple to merge since they relax the relative ordering of statements within control structures).

In what follows, let G, G_A, and G_B represent the program dependence graphs of the programs *Base*, A, and B, respectively, that are produced by GeneratePDG of Section 4.1.

If the slice of variant G_A at vertex v differs from the slice of G at vertex v (*i.e.* they are different graphs), then values at v are computed in a different manner by the respective programs. This means that the values at v may differ, and we take this as our definition of changed behavior. We define the *affected points* D_A of G_A as the subset of vertices of G_A whose slices in G and G_A differ:

$$D_A = \{ v \mid v \in G_A \wedge G/v \neq G_A/v \}$$

The slice G_A/D_A captures the behavior of A that differs from *Base*.

There is a simple technique to determine D_A that avoids computing all of the slices stated in the definition. The technique requires at most two complete examinations of G_A, and is based on the following three observations:

a) All statements and predicates that are in G_A but not in G are affected points.

b) Each vertex w of G_A that has a different set of incoming control or flow edges in G_A than in G gives rise to a set of affected points – those vertices that can be reached via zero or more control or flow edges from w.

c) Each vertex w of G_A that has an incoming def-order edge $w' \longrightarrow_{do(u)} w$ that does not occur in G gives rise to a set of affected points – those vertices that can be reached via zero or more control or flow edges from u.

The justification for observation a) is straightforward: for $w \in V(G_A) - V(G)$, G/w is the empty graph, whereas $w \in V(G_A/w)$, so G_A/w is not empty. The justification for observation b) is also straightforward. By the definition of slicing, when w differs in incoming flow or control edges, G_A/w and G/w can not be the same, hence w itself is affected. For any vertex v that is flow or control dependent on w, the slices G_A/v and G/v contain w. Therefore, if w is affected, all *successors* of w via control and flow dependencies are also affected.

The justification for observation c) is more subtle. When a def-order edge $w' \longrightarrow_{do(u)} w$ occurs in G_A but not in G, then the slice G_A/u will include both w' and w and the def-order edge between them, while G/u will not include this edge. Hence u is affected. The reverse situation, where $w' \longrightarrow_{do(u)} w$ occurs in G but not in G_A means u is affected if $u \in G_A$. But it is not necessary to examine this possibility since either $w' \longrightarrow_{do(u)} w$ in G is replaced by $w \longrightarrow_{do(u)} w'$ in G_A, in which case $w' \in G_A$ will contribute u as affected, or else one or both of the flow edges $w \longrightarrow_f u$ and $w' \longrightarrow_f u$ in G will be missing in G_A, in which case u is affected by the change in incoming flow edges. As before, for any vertex v that is flow or

control dependent on u, the slices G_A / v and G / v contain u; therefore, if u is affected, all *successors* of u via control and flow dependencies are affected. Note that neither w' itself nor w itself is necessarily an affected point.

Observations a), b), and c) serve to characterize the set of affected points. If $v \in G_A$ is affected there must be some w in G_A / v with different incoming edges in G_A and G. By the arguments above, either w itself is an affected point (cases a) and b)), or it contributes a vertex $u \in V(G_A / v)$ that is an affected point (case c)); therefore, it is possible to identify v as an affected point by following control and flow edges. This latter observation forms the basis for the program AffectedPoints(G', G), given in Figure 3. It computes the set of affected points of G' with respect to G by examining all vertices w in G' that have a different set of incoming edges in G' than in G, and collecting the affected points that each vertex contributes. Then a flooding algorithm is used to find all vertices reachable from this set by flow or control edges; this is the set of affected points of G'.

```
function AffectedPoints(G', G) returns a set of vertices
declare
   G', G: program dependence graphs
   S, Answer: sets of vertices
   w, u, b, c: individual vertices
begin
   S := ∅
   for each vertex w in G' do
      if w is not in G then
         Insert w in S
      fi
      if the sets of incoming flow and control edges to w in G' is different from the incoming sets to w in G then
         Insert w in S
      fi
      for each def-order edge w' ⟶_do(u) w that occurs in G' but not in G do
         Insert u in S
      od
   od
   Answer := ∅
   while S ≠ ∅ do
      Select and remove an element b from S
      Insert b in Answer
      for each vertex c such that b ⟶_f c or b ⟶_c c in G' and c ∉ (Answer ∪ S) do
         Insert c into S
      od
   od
   return(Answer)
end
```

Figure 3. The function AffectedPoints determines the points in the program dependence graph G' that may yield different values in G' than in G.

4.3. Merging Program Dependence Graphs

We now show how to merge the program dependence graphs of two variants so that the differences between the behavior of *Base* and its variants are preserved. The approach we describe accommodates the simultaneous merge of any number of variants, but for the sake of exposition we consider the common case of two variants A and B.

The changed behavior in variant A is characterized by the slice G_A / D_A. The behavior at program points $\overline{D}_A = V(G_A) - D_A$ remains unchanged, so the unchanged behavior is characterized by $G_A / \overline{D}_A$. For every $v \in \overline{D}_A$ we have $G_A / v = G / v$ hence $G_A / \overline{D}_A = G / \overline{D}_A$.

Similarly, in variant B, G_B / D_B characterizes the new behavior, $\overline{D}_B = V(G_B) - D_B$ characterizes the unchanged points in B, and $G_B / \overline{D}_B = G / \overline{D}_B$ characterizes the unchanged behavior of B. The unchanged program points common to both variants are $\overline{D}_A \cap \overline{D}_B$ and the unchanged behavior common to both is

$$G / (\overline{D}_A \cap \overline{D}_B) = G_A / (\overline{D}_A \cap \overline{D}_B) = G_B / (\overline{D}_A \cap \overline{D}_B)$$

The merged graph G_M should be composed of the elements of G_A that characterize the changed behavior of A, the elements of G_B that characterize the changed behavior of B, and the elements from G that characterize common unchanged behavior[2].

$$G_M = (G_A / D_A) \cup (G_B / D_B) \cup (G / (\overline{D}_A \cap \overline{D}_B))$$

A simple way to implement the construction of G_M is to mark the vertices and edges of G_A / D_A in G_A, G_B / D_B in G_B, and either $G_A / (\overline{D}_A \cap \overline{D}_B)$ in G_A or $G_B / (\overline{D}_A \cap \overline{D}_B)$ in G_B. Then $v \in V(G_M)$ if v is marked in G_A or G_B (and similarly for the edges of G_M).

4.4. Determining Whether Two Versions Interfere

A merged program dependence graph, G_M, that is created by the method described in the previous section can fail to reflect the changed behavior of the two variants, A and B, in two ways. First, because the union of two feasible PDG's is not necessarily a feasible PDG, G_M may not correspond to a feasible PDG. Second, it is possible that G_M will not preserve the differences in behavior of A or B with respect to *Base*. If either condition occurs, we say that A and B interfere. Testing for interference due to the former condition is addressed in Section 4.5; this section describes a criterion for determining whether a merged program dependence graph preserves the changed behavior of A and B.

To insure that the changed behavior of variants A and B is preserved in G_M, we introduce a non-interference criterion based on comparisons of slices of G_A, G_B, and G_M; the condition that must hold for the behavior of A and B to be preserved in G_M is:

$$G_M / D_A = G_A / D_A \quad \text{and} \quad G_M / D_B = G_B / D_B$$

On vertices in $\overline{D}_A \cap \overline{D}_B$ the graphs G_A and G_B agree and hence G_M is correct for these vertices.

The verification of the invariance of the slices in G_M and the variant graphs is closely related to the problem of finding the difference between two PDGs: G_M must agree with variant A on D_A and with B on D_B. Therefore an easy way to test for non-interference is to verify that

$$\text{AffectedPoints}(G_M, G_A) \cap D_A = \emptyset \quad \text{and} \quad \text{AffectedPoints}(G_M, G_B) \cap D_B = \emptyset$$

---

[2]A simplified form of this definition involving only two slices is given in Appendix B.

4.5. Reconstituting a Program From the Merged Program Dependence Graph

This section describes procedure ReconstituteProgram, which is invoked as step five of the program integration algorithm. Given a program dependence graph G_M that was created by merging non-interfering variants A and B, ReconstituteProgram determines whether G_M is feasible (*i.e.* corresponds to some program), and, if it is, creates an appropriate program from G_M.

Because we are assuming a restricted set of control constructs, each vertex of G_M is immediately subordinate to at most one predicate vertex, *i.e.* the control dependencies of G_M define a tree T rooted at the entry vertex. The crux of the program-reconstitution problem is to determine, for each predicate vertex v (and for the entry vertex as well), an ordering on v's children in T. Once all vertices are ordered, T corresponds closely to an abstract-syntax tree. We assume that a function, named TransformToSyntaxTree, has been provided to convert tree T with ordered vertices into the corresponding abstract-syntax tree.

The algorithm for reconstituting a program from program dependence graph G_M is presented in outline form in Figure 4 as the function ReconstituteProgram. In ReconstituteProgram, the tree T induced on G_M by its control dependencies is traversed in post-order. For each vertex v of T visited during the traversal, an attempt is made to determine an acceptable order for v's children; this attempt is performed by the procedure OrderRegion, which is explained in detail below.

Given infeasible program dependence graph G_M, ReconstituteProgram will fail in one of two ways. Failure can occur because procedure OrderRegion determines that there is no acceptable ordering for the children of some vertex. Failure can also occur at a later point, after OrderRegion succeeds in ordering all vertices of G_M. In this case, TransformToSyntaxTree is used to produce program P from G_M, GeneratePDG is applied to P to produce program dependence graph G_P, and G_P is compared to G_M. If G_M is infeasible it is not identical to the dependence graph of any program; thus, G_M is not identical to G_P, and ReconstituteProgram fails.

---

function ReconstituteProgram(G_M) **returns** a program or FAILURE
declare
  G_M: a program dependence graph
  T: the tree defined by the vertices and control edges of G_M
  v: a vertex of G_M
begin
[1] **for** each vertex v of T in a post-order traversal of T **do**
[2] **if** the vertices in the subtree rooted at v cannot be ordered **then return**(FAILURE) **else** order them **fi**
[3] **od**
[4] P := TransformToSyntaxTree(G_M);
[5] **if** G_M = GeneratePDG(P) **then return**(P)
[6] **else return**(FAILURE)
[7] **fi**
end

Figure 4. The operation ReconstituteProgram(G_M) creates a program corresponding to the program dependence graph G_M by ordering all vertices, or discovers that G_M is infeasible.

Conversely, given feasible program dependence graph G_M, ReconstituteProgram always succeeds. OrderRegion orders all vertices of G_M, and TransformToSyntaxTree produces program P with dependence graph identical to G_M.

Unfortunately, as we show in Appendix C, the problem of determining whether it is possible to order a vertex's children is NP-complete. Although we believe that a backtracking method for solving this problem will behave satisfactorily on actual programs, it is also possible to completely side-step this difficulty by having OrderRegion perform a limited amount of variable renaming. This matter is discussed in more detail in Section 4.5.3.

4.5.1. Procedure OrderRegion: Ordering vertices within a subtree

Definition. The collection of vertices subordinate to a particular predicate vertex v is called a *region*; v is the *region head.* If v represents the predicate of a conditional, v is the head of *two* regions; one region includes all statements in the "true" branch of the conditional, the other region includes all statements in the "false" branch of the conditional. For all vertices w, EnclosingRegion(w) is the region that includes w (*not* the region of which w is the head). Because the entry vertex and the vertices representing initial definitions and final uses of variables are not subordinate to any predicate vertex they are not included in any region.

Given region R, the main job of procedure OrderRegion is to find a total ordering of the vertices of R that preserves the flow and def-order dependencies of R, or to discover that no such ordering is possible. A secondary responsibility of OrderRegion is to project some information from the vertices of R onto the head of R.

To order the vertices of R, OrderRegion adds some edges to R and removes some edges from R. If this introduces a cycle in R, OrderRegion fails; otherwise, it creates a directed acyclic graph (a DAG) that can be topologically ordered to produce an acceptable ordering of R. The creation of the DAG is based on the following observations:

Observation (1). Loop-carried flow edges and edges with a single endpoint inside the region under consideration identify the *upwards-exposed uses* and *downwards-exposed definitions* [Aho et al. 1986] of the region. Once exposed uses and definitions are identified, new edges can be added to the region to exclude some erroneous topological orderings. For example, consider the dependence graph fragment shown in Figure 5. A legal topological ordering of the vertices of the region subordinate to vertex C is: F, G, D, E; however, the dependence graph of the program generated according to this ordering will be missing the flow edge from F to H, and will have an extra flow edge from D to H.

The new edges added to R run from uses of variables to definitions of the same variables, or from definitions of variables to other definitions of the same variables; such edges have been called *anti-dependencies* and *output dependencies,* respectively [Kuck et al. 1981].

Once these new edges have been added, the loop-carried flow edges and edges with a single endpoint inside the region are projected up onto the region head and play no further role in the ordering of the vertices within the region.

Identifying exposed uses and definitions, inserting anti- and output-dependence edges, and projecting edges onto the region head are accomplished by procedure PreserveExposedUsesAndDefs, explained in Section 4.5.2 below.

Observation 2. Even with the edges added to preserve exposed uses and definitions, topologically sorting R can lead to an incorrect ordering. Consider the dependence graph fragment shown in Figure 6. A legal topological ordering of the vertices of the region subordinate to vertex A is: B, D, C, E; however, the

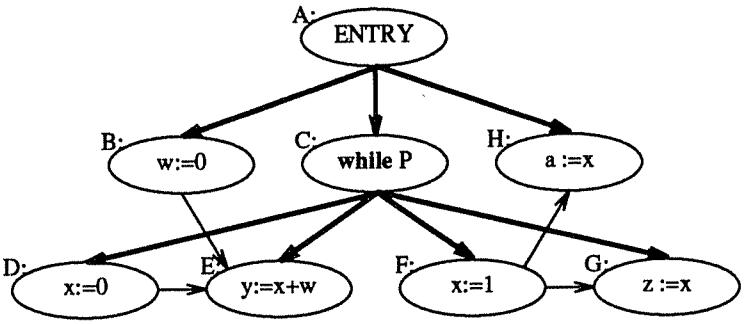


Figure 5. Dependence graph fragment: Topological ordering F, G, D, E, of the vertices subordinate to vertex C is not acceptable.

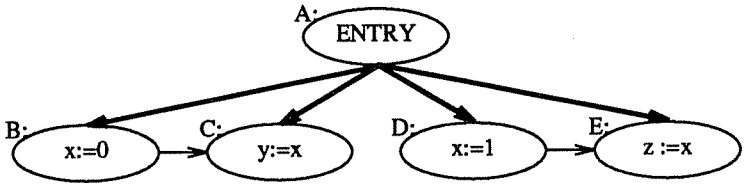


Figure 6. Dependence graph fragment: Topological ordering B, D, C, E is not acceptable.

dependence graph of the program generated according to this ordering will be missing the flow edge from B to C, and will have an extra flow edge from D to C.

This problem can be characterized as "letting a definition come between a def-use pair"; the ordering B, D, C, E allows the definition of x represented by vertex D to come between the def-use pair represented by vertices B and C. The solution to this problem also requires the insertion of new anti- and output-dependence edges, and is carried out by procedure PreserveSpans, explained in Section 4.5.3.

To ensure that future invocations of PreserveSpans have complete information, procedure PreserveSpans also projects information about variable definitions from the vertices in region R onto the head of R.

To summarize: Procedure OrderRegion calls procedures PreserveExposedUsesAndDefs, and PreserveSpans to add anti- and output dependence edges to the region to exclude erroneous topological orderings, and to project information from the vertices in the region onto the region head. Procedure OrderRegion is shown in Figure 7.

```
procedure OrderRegion(R)
declare
  R: a region
begin
  PreserveExposedUsesAndDefs(R)
  PreserveSpans(R)
  if R contains no cycles then TopSort(R) else fail fi
end
```

Figure 7. Procedure OrderRegion adds new edges to the given region to ensure that dependencies are respected, projects information onto the region head, and topologically sorts the vertices of the region.

4.5.2. Procedure PreserveExposedUsesAndDefs: Preserving upwards-exposed uses and downwards-exposed definitions

For all variables x, a use of x that is upwards-exposed within a region must precede all definitions of x within the region other than its loop-independent flow-predecessors (a use of x can be upwards-exposed and still have a loop-independent flow-predecessor that defines x within the region if the flow-predecessor represents a conditional definition). Vertex E in Figure 5 represents an upwards-exposed use of variable w.

Similarly, a definition of x that is downwards-exposed within a region must follow all other definitions of x within the region other than those to which it has a def-order edge (again, a definition of x can be downwards-exposed and still precede a conditional definition of x). Vertex F in the example of Figure 5 represents a downwards-exposed definition of variable x.

Procedure PreserveExposedUsesAndDefs uses flow edges having only one endpoint inside the given region R, and loop-carried flow edges having both endpoints inside R to identify exposed uses and definitions. It then adds edges to R to ensure that exposed uses and definitions are ordered correctly with respect to other definitions within the region. Finally, the edges used to identify exposed uses and definitions are removed from R and are projected onto the region head. Def-order edges with a single endpoint inside R are also projected onto head(R). This ensures that the region enclosing the head of R will be ordered correctly during a future call to OrderRegion. PreserveExposedUsesAndDefs performs the following four steps:

Step (1): Identify upwards-exposed uses.

A vertex with an incoming loop-independent flow edge whose source is outside region R, or with an incoming loop-carried flow edge with arbitrary source, represents an *upwards-exposed use* of the variable x defined at the source of the flow edge. Mark each such vertex UPWARDS-EXPOSED-USE(x).

Step (2): Identify downwards-exposed definitions.

A vertex that represents a definition of variable x and has an outgoing loop-independent flow edge whose target is outside region R, or has an outgoing loop-carried flow edge with arbitrary target, represents a downwards-exposed definition of x. Mark each such vertex DOWNWARDS-EXPOSED-DEF(x).

Step (3): Preserve exposed uses and definitions.

For each vertex n marked UPWARDS-EXPOSED-USE(x), add a new edge from n to all vertices m in the region such that m represents a definition of variable x, and m is not a loop-independent flow predecessor of n. For each vertex n marked DOWNWARDS-EXPOSED-DEF(x), add a new edge to n from all vertices m in the region such that m represents a definition of x and there is no def-order edge from n to m.

Step (4): Project edges onto the region head.

Replace all flow and def-order edges with source outside of R and target inside R with an edge from the same source to head(R). Replace all flow and def-order edges with source inside R and target outside of R with an edge from head(R) to the same target.

Consider each loop-carried flow edge with both source and target in R. If head(R) represents the predicate of the only loop responsible for the presence of this edge, then remove the edge without replacing it. Otherwise, replace the edge with a self-loop at head(R).

Figure 8 shows the example dependence graph fragment of Figure 5 after the four steps described above have been performed on the region subordinate to vertex C. The new edge added from vertex D to vertex F prevents F from preceding D in a topological ordering.

4.5.3. Dependencies induced by spans

In the example dependence graph fragment of Figure 8, the ordering D, F, E, G of the vertices subordinate to vertex C is a topological, but erroneous one. The problem with this ordering is that it allows the definition of variable x at vertex F to "capture" the use of x at vertex E. In general, once a definition of variable x has been executed, all uses of x reached by that definition must be executed before any other definition of x. This observation leads to the following definition:

Definition. The *span* of a definition d, where d defines variable x, is the set $\{d\}$ together with all uses of x that are in the same region as d, and are loop-independent flow targets of d:

$Span(d) = \{d\} \cup \{u \mid d \longrightarrow_{li} u\} \cap \text{EnclosingRegion}(d)$

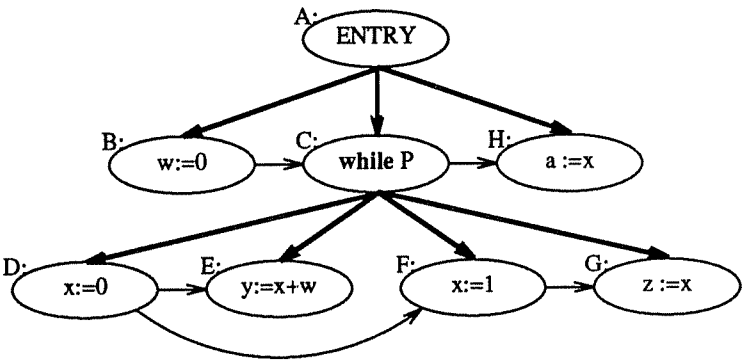


Figure 8. Dependence graph fragment with new edge D → F added to preserve the downwards-exposed definition of x at vertex F.

The span of a vertex that defines variable x is called an x-span.

Restating the observation above in terms of spans, once definition d of variable x has been executed, all vertices in Span(d) must be executed before executing any other definition of x. Furthermore, if any vertex in Span(d_1) must be executed before some vertex in Span(d_2), where d_1 and d_2 both represent definitions of variable x, then all vertices in Span(d_1) must be executed before d_2.

The exception to this rule is that a conditional definition of x may come between another definition and its use, if the conditional definition is also a flow predecessor of the use. To simplify this section's presentation, we begin by ruling out this kind of exception by considering only dependence graphs of programs without loops or conditionals; under this restriction, each use of variable x is reached by exactly one definition of x.

In this case, erroneous topological orderings are excluded by considering, for each variable x, all pairs $<d_1, d_2>$ of definitions of x. If there is some vertex v in Span(d_1) that must precede some vertex w in Span(d_2), (because of a path from v to w along existing flow, anti-dependence, or output dependence edges), then edges are added from all vertices in Span(d_1) – Span(d_2) to vertex d_2. Similarly, if there is a path from a vertex in Span(d_2) to a vertex in Span(d_1), edges are added from all vertices in Span(d_2) – Span(d_1) to vertex d_1. In the graph fragment of Figure 8, the edge E → F would be added because the edge D → F forms a path from Span(D) to Span(F), and vertex E is in Span(D) – Span(F).

The reason for taking the set difference Span(d_1) – Span(d_2), is that even in graphs corresponding to straight-line code, spans can overlap, as illustrated in Figure 9. Because C is itself in Span(B), adding edges from *all* vertices in Span(B) to C would create a self-loop at C, making a topological ordering impossible.

There may be pairs $<d_1, d_2>$ for which there is no path in either direction between Span(d_1) and Span(d_2). It is still necessary to add edges to force one span to precede the other so as to exclude erroneous topological orderings. Although it might seem that an arbitrary choice can be made, Figure 10 gives an example in which making the wrong choice leads to the introduction of a cycle in a fragment of a feasible graph.

The fragment of Figure 10 includes two x-spans: Span(A) and Span(D), and two y-spans: Span(B) and Span(C). There are paths neither between the two x-spans nor between the two y-spans; thus, it appears

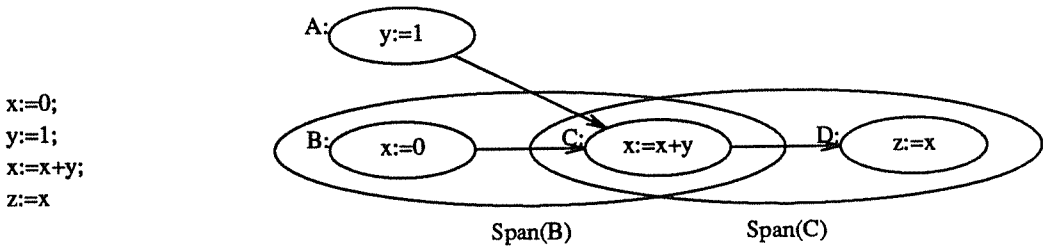


Figure 9. Straight-line code fragment and corresponding dependence graph fragment (control edges omitted) with overlapping x-spans.

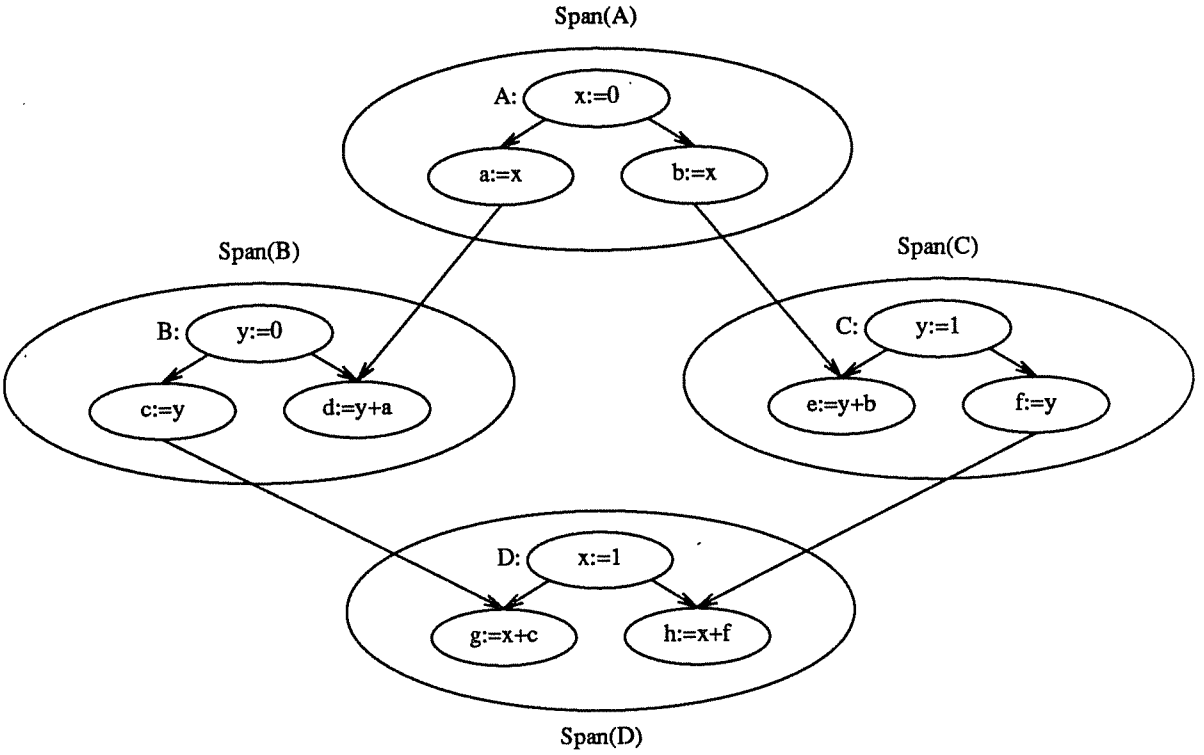


Figure 10. Graph fragment (control edges omitted) with two x -spans and two y -spans

that one is free to choose to add edges from the vertices of Span(A) to vertex D, or from the vertices of Span(D) to vertex A, or from the vertices of Span(B) to vertex C, or from the vertices of Span(C) to vertex B. However, while three out of these four choices lead to a successful ordering of the vertices, choosing to add edges from the vertices of Span(D) to vertex A leads to the introduction of a cycle. This is because the introduction of these new edges creates paths both from a vertex in Span(B) to a vertex in Span(C), and *vice versa*. Figure 11 shows the fragment of Figure 10 with the new edges added; the path from Span(C) to Span(B) is shown using dashed lines. The path from Span(B) to Span(C) is symmetric.

Unfortunately, as we show in Appendix C, the problem of determining the right choice in a situation like the one illustrated in Figure 10 is NP-complete. However, we expect that in practice there will be very few such choices to be made, and a simple backtracking algorithm will suffice: if a cycle is introduced when ordering spans, the algorithm backtracks to the most recent choice point, and tries a different choice. If all choices lead to the introduction of a cycle, the graph is infeasible.

It is also possible to side-step this difficulty entirely by slightly redefining our goals. Rather than insist that the program produced by ReconstituteProgram preserve the usage patterns of variables that exist in graph G_M, we can have OrderRegion perform a limited amount of variable renaming. In particular, when two x -spans are not connected by a path in either direction, all occurrences of the name x in one of the two

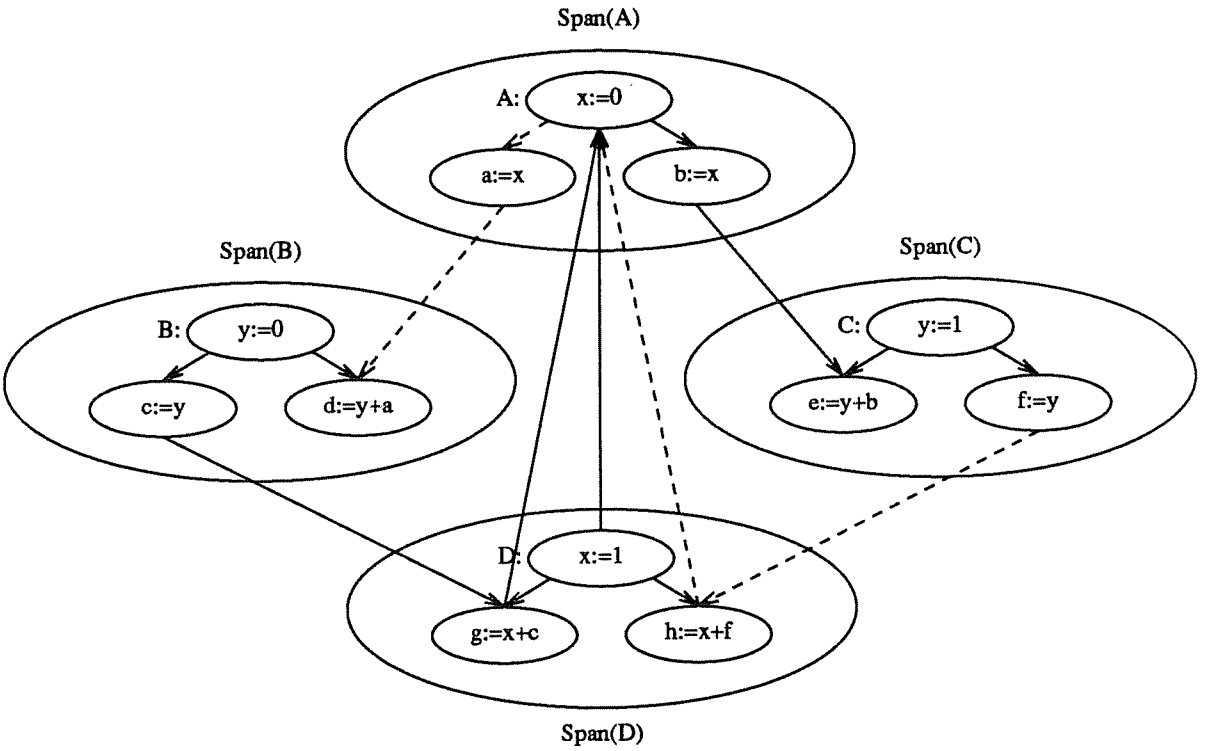


Figure 11. Span(D) has been chosen to precede Span(A). Paths have been created from Span(B) to Span(C) and *vice versa*. The path from Span(C) to Span(B) is indicated using dashed edges.

spans can be replaced by a new name not appearing elsewhere in the program. This renaming removes all problematic choices, and thus PreserveSpans need never backtrack. The disadvantage of this measure is that the integrated program will include variable names that did not appear in either variant. Further work is needed to determine whether this technique will be necessary in practice.

Recall that we have limited our discussion thus far to the dependency graphs of programs without loops or conditionals. Allowing these constructs introduces the possibility that spans may overlap in two new ways, as illustrated in Figure 12. In the first case there must be a def-order dependency edge from d_1 to d_2 or *vice versa*, or the graph would have failed the interference test of Section 4.4. In the second case there is a flow edge from d_1 to d_2. These edges are sufficient to force an ordering of the two spans; thus, allowing conditionals and loops does not complicate PreserveSpans.

The head of region R must represent definitions of all the variables defined in R when the head of R is considered as a vertex in its enclosing region. Therefore, the final step of procedure PreserveSpans is to project information about variable definitions from the vertices of a region onto the region head. For example, PreserveSpans would designate vertex C of Figure 5 as representing definitions of x, y, and z, because of the definitions of these variables at vertices D and F, E, and G, respectively.

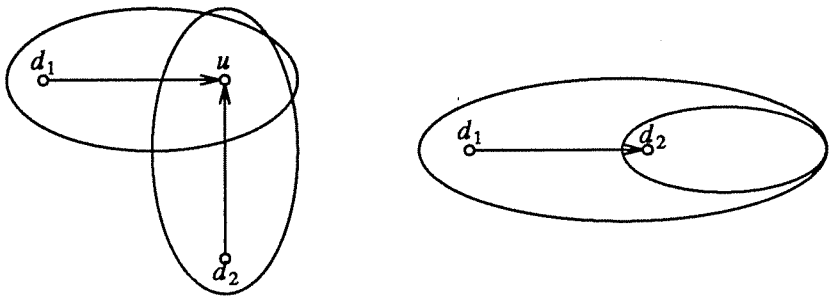


Figure 12. Conditionals and loops can lead to these kinds of overlapping spans.

4.6. Recap of the Program Integration Algorithm

The function Integrate, given in Figure 13, takes as input three programs, A, B, and *Base*, where A and B are variants of *Base*. Whenever the changes made to *Base* to create A and B do not interfere, Integrate produces a program P that integrates A and B.

```
function Integrate(A, B, Base) returns a program
declare
   A, B, Base: programs
   G_A, G_B, G_M: program dependence graphs
   D_A, D_B: subsets of the vertices of A and B, respectively
   P: a program
begin
   G := GeneratePDG(Base)
   G_A := GeneratePDG(A)
   G_B := GeneratePDG(B)
   D_A := AffectedPoints(G_A, G)
   D_B := AffectedPoints(G_B, G)
   G_M := (G_A / D_A) ∪ (G_B / D_B) ∪ (G_A / (D̄_A ∩ D̄_B))
   if G_M / D_A ≠ G_A / D_A ∨ G_M / D_B ≠ G_B / D_B then exit with failure fi
   P := ReconstituteProgram(G_M)
   if P = FAILURE then exit with failure fi
   return(P)
end
```

Figure 13. The function Integrate takes as input three programs A, B, and *Base*, where A and B are variants of *Base*. Whenever the changes made to *Base* to create A and B do not interfere, Integrate produces a program P that integrates A and B.

Example. Consider how *Integrate* performs on the example from Section 3, on which *diff3* performed so miserably. The base program and its two variants are:

Base program

```
if P then x := 0 fi
if Q then x := 1 fi
y := x
if R then w := 3 fi
if S then w := 4 fi
z := w
```

Variant *A*

```
if Q then x := 1 fi
if P then x := 0 fi
y := x
if R then w := 3 fi
if S then w := 4 fi
z := w
```

Variant *B*

```
if S then w := 4 fi
if R then w := 3 fi
z := w
if P then x := 0 fi
if Q then x := 1 fi
y := x
```

The program dependence graphs for *Base*, *A*, and *B* are shown in Figure 14. The program-integration algorithm would determine that there is a single affected point of G_A: the assignment statement $y := x$. This means that the part of variant *A*'s computation that must be preserved is:

```
if Q then x := 1 fi
if P then x := 0 fi
y := x
```

Similarly, $z := w$ is the single affected point of G_B, and the part of variant *B*'s computation that must be preserved is:

```
if S then w := 4 fi
if R then w := 3 fi
z := w
```

The integration algorithm then merges the program dependence graphs G_A and G_B, tests them for interference (they do not interfere), and creates one of a number of programs, including the following three:

```
if S then w := 4 fi
if R then w := 3 fi
z := w
if Q then x := 1 fi
if P then x := 0 fi
y := x
```

```
if Q then x := 1 fi
if P then x := 0 fi
y := x
if S then w := 4 fi
if R then w := 3 fi
z := w
```

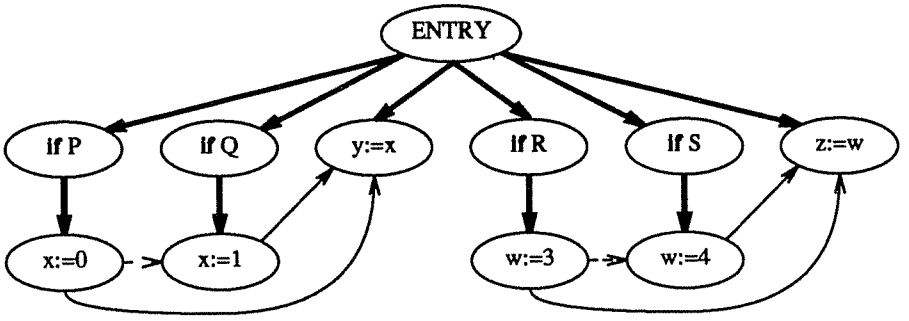
```
if Q then x := 1 fi
if P then x := 0 fi
if S then w := 4 fi
if R then w := 3 fi
y := x
z := w
```

These programs and any of the other possible products of *Integrate* are a satisfactory outcome for integrating *Base*, *A*, and *B*.

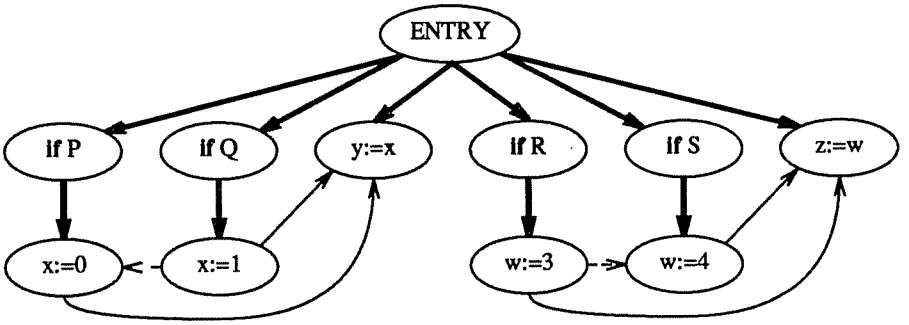
5. APPLICATIONS OF PROGRAM INTEGRATION FOR PROGRAMMING IN THE LARGE

Programming in the large addresses problems of organizing and relating designs, documentation, individual software modules, software releases, and the activities of programmers. The manipulation of related versions of programs is at the heart of a number of these issues. In some respects, the program-integration problem is the key operation for creating a programming environment to support programming in the large.

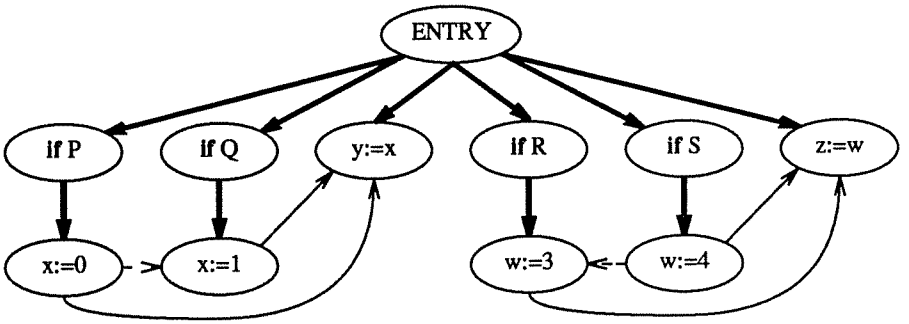
One context in which the program integration problem arises is when several related versions of a system exist and one desires to make the same enhancement or correction to all of them. In this situation, the changes that one makes to a base version of a system must be repeated on the source code for other versions that are to receive the enhancement. The algorithm *Integrate* provides a way for changes made to the



G_{Base}



G_A

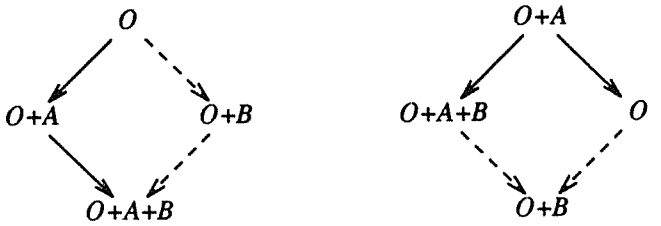


G_B

Figure 14. Program dependence graphs for *Base*, *A*, and *B*.

base version to be automatically installed in the other versions.

Our program-integration algorithm also makes it possible to separate consecutive edits on the same program into individual edits on the original base program. For example, consider the case of two consecutive edits to a base program O; let $O+A$ be the result of the first modification to O and let $O+A+B$ be the result of the modification to $O+A$. Now suppose we want to create a program $O+B$ that includes the second modification but not the first. This is represented by case (a) in the following diagram:



However, by re-rooting the development-history tree so that $O+A$ is the root, the diagram is turned on its side and becomes a program-integration problem (situation (b)). The base program is now $O+A$, and the two variants of $O+A$ are O and $O+A+B$. Instead of treating the differences between O and $O+A$ as changes that were made to O to create $O+A$, they are now treated as changes made to $O+A$ to create O. For example, when O is the base program, a vertex v that occurs in $O+A$ but not in O is a "new" vertex arising from an insertion; when A is the base program, we treat the missing v in O as if a user had deleted v from A to create O. Version $O+A+B$ is still treated as being a program version derived from $O+A$.

6. RELATION TO PREVIOUS WORK

We have not seen any other work that permits one to integrate programs so as to preserve changes to a base program's behavior. In this section, we elaborate on some technical differences between the program dependence graphs employed in this paper and the program dependence representations that have been defined and used by others.

Our use of the term "program dependence graph" could lead to some confusion because the data structure defined in Section 4.1 is similar, but not identical, to the program dependence graphs defined by several other authors. In fact, the term does not have a standard definition, and slightly different definitions of program dependence representations have been given, depending on the application. Nevertheless, they are all variations on a theme introduced in [Kuck et al. 1972], and share the common feature of having explicit representations of both control dependencies and data dependencies.

Previous program dependence representations have included data dependency edges to represent flow dependencies and two other kinds of data dependencies, called *anti-dependencies* and *output dependencies*. (All three kinds may be further characterized as loop independent or loop carried). Def-order dependencies have not been previously defined.

For flow dependencies, anti-dependencies, and output dependencies, a program component v_2 has a dependency on component v_1 due to variable x only if execution can reach v_2 after v_1 and if there is no intervening definition of x along the execution path by which v_2 is reached from v_1. There is a flow dependency if v_1 defines x and v_2 uses x; there is an anti-dependency if v_1 uses x and v_2 defines x; there is an output dependency if v_1 and v_2 both define x.

Although def-order dependencies resemble output dependencies in that they both relate two assignments to the same variable, they are two different concepts. An output dependency $v_1 \longrightarrow_o v_2$ between two definitions of x can hold only if there is no intervening definition of x along some execution path from v_1

to v_2; however, there can be a def-order dependency $v_1 \longrightarrow_{do} v_2$ between two definitions even if there is an intervening definition of x along *all* execution paths from v_1 to v_2. This situation is illustrated by the following example program fragment, which demonstrates that it is possible to have a program in which there is a dependency $v_1 \longrightarrow_{do} v_2$ but not $v_1 \longrightarrow_{o} v_2$, and *vice versa*:

```
[1]     x := 10
[2]     if P then
[3]             x := 11
[4]             x := 12
[5]     fi
[6]     y := x
```

The one def-order dependency, $1 \longrightarrow_{do([6])} 4$, exists because the assignments to x in lines [1] and [4] both reach the use of x in line [6]. In contrast, the output dependencies are $1 \longrightarrow_{o} 3$ and $3 \longrightarrow_{o} 4$, but there is no output dependency $1 \longrightarrow_{o} 4$.

One of the drawbacks of using output dependencies in place of def-order dependencies is that using output dependencies would cause some equivalent programs to have unequal PDG's. This situation is illustrated below by the two program fragments:

```
x := 10          x := 11
a := x           b := x
x := 11          x := 10
b := x           a := x
x := 12          x := 12
c := x           c := x
```

In the left-hand fragment, the output dependencies are $x := 10 \longrightarrow_{o} x := 11$ and $x := 11 \longrightarrow_{o} x := 12$; in the right-hand fragment, they are $x := 11 \longrightarrow_{o} x := 10$ and $x := 10 \longrightarrow_{o} x := 12$. In contrast, none of the elements in the two fragments are related by def-order dependencies, so under our definition their PDG's are equal.

A final reason for choosing def-order dependencies over output dependencies is that def-order dependencies make it easier to define both the operation that merges two program dependence graphs and the interference test. One can get a feeling for the difficulties that would occur if PDG's were to contain output dependencies by considering the complications that arise in defining a program slice. Suppose statement [6] is an affected point of:

```
[1]     x := 10
[2]     if P then
[3]             x := 11
[4]             x := 12
[5]     fi
[6]     y := x
```

The vertices of the slice consist of all assignments and predicates of the fragment, except for [3]. Notice, however, that among its edges the slice must contain the output dependency [1] $\longrightarrow_{o}$ [4], which does not appear in the original PDG; the original PDG contains the dependencies [1] $\longrightarrow_{o}$ [3] and [3] $\longrightarrow_{o}$ [4]. The correct definition of slicing involves examining some, but not all, transitive output dependencies; similar complications would occur in the definition of the operation that merges two PDG's.

In other definitions that have been given for program dependence graphs, there is an additional control dependence edge for each predicate of a **while** statement -- each predicate has an edge to itself labeled **true**. By including the additional edge, the predicate's outgoing **true** edges consist of every program element that is guaranteed to be executed (eventually) when the predicate evaluates to **true**. This kind of edge was left out of our definition because it was not necessary for our purposes.

The problem of generating program text from a program dependence graph has previously been addressed only in a context that admits a considerably simpler solution. In previous work, the program dependence graph is known to correspond to some program. For example, in the work on program slicing, because the slice is derived from a program dependence graph whose text is known, when creating the textual image of a slice, it suffices to take the text of the original program and delete all tokens that do not correspond to components of the slice [Ottenstein & Ottenstein 1984].

Our work requires a solution to a more general problem because the final program dependence graph is created by merging two other program dependence graphs. The merged program dependence graph may not correspond to any program at all, but even if it does, this program is not known *a priori*, when ReconstituteProgram is invoked. As shown in Appendix C, the problem of deciding whether a PDG is feasible is NP-complete.

Ferrante and Mace describe an algorithm for generating sequential code for programs written in a language with a multiple GOTO operator and impose the condition that the algorithm not duplicate any code in this process [Ferrante & Mace 1984]. Programs written in the language they consider have a close correspondence to the subgraph of control dependencies of a program dependence graph. They discuss the application of their algorithm to compiling a program dependence graph for execution on a sequential machine; however, they assume that only a certain class of optimizing transformations has been applied to the original PDG (which was generated from some program). They assert that the transformations of this class preserve the property that the resulting graph corresponds to some program. Thus, while their results are relevant to generalizing ReconstituteProgram to work on PDG's generated from arbitrary programs [Ferrante et al. 1987], they will have to be extended to account for infeasibility due to a PDG's data dependencies.

7. FUTURE WORK

In this paper, the problem of program integration is studied in an extremely simplified setting. For this reason, the procedure described in the paper is not a realistic method for integrating programs; however, we feel that the approach that we have developed represents a promising basis for further work. Below, we outline some possible ways to extend our work.

Among the obvious deficiencies of the present study are the absence of programming constructs and data types found in the languages used for writing "real" programs. One area for further work, therefore, is to extend the integration method to handle other programming language constructs, such as **break** statements, procedure calls, and I/O statements, as well as other data types, such as records, arrays, and pointers. For each extension to the programming language, it will be necessary to make some adjustments to certain phases of the integration method. For example, the data-flow analysis required to determine data dependencies in the presence of **break** statements is more complicated, but straightforward.

To incorporate procedure calls, it will be necessary to perform interprocedural data-flow analysis to compute aliasing and may-mod information. This information can then be used by an extended form of program slicing to cut down on the number of components that a procedure call contributes to a slice. Because techniques for using interprocedural data-flow analysis in conjunction with program slicing have already been developed by Weiser [Weiser 1984], we anticipate few difficulties in extending our program integration methods in this direction.

One way to incorporate input and output statements is to treat them as accesses to two special objects *input* and *output* [Ottenstein & Ottenstein 1984]. The objects *input* and *output* may be thought of as streams that get updated whenever operations are performed on them. For example, an output statement

write x

would be treated as an assignment

output := *output* | StringValueOf(x)

where the symbol | represents string concatenation. Consequently, output statements would be treated just like assignment statements in terms of detecting changes to a base program's behavior, and the relative order of output statements appearing in a program P would be captured in G_P by flow dependencies.

Programming languages with record data types can be handled in a straightforward manner by treating each field of a record variable as a separate variable. The simplest way of handling arrays is to treat an update to any cell as an update to the entire array. More sophisticated analysis of array access patterns, such as the analysis of "recurrences" used in some vectorizing compilers, does not appear to permit a more precise analysis of whether or not two program variants interfere. To incorporate pointer-valued variables, an analysis of pointer usage probably will be necessary; without the information that such an analysis would provide, an update via a dereferenced pointer would have to be considered to update every location in memory.

We anticipate that it will be useful to create different varieties of program-integration procedures based on our techniques, but that produce integrated programs with slightly different properties. For example, it may be useful to develop a variation of the integration method in which the integrated program does not read the same kind of input data files that are used by the base program; in cases where it is not important to preserve the format of the program's input data, this might make it possible to perform integrations that would otherwise fail. Similarly, it may be useful to develop a variation in which the integrated program does not produce the same kind of output as the base program. A somewhat different possibility exists when one can anticipate that a successfully integrated program will never have to be examined by a human programmer. Under these conditions, there are perhaps more liberal notions of program integration; for example, the integration procedure should be permitted to rename freely any variable that occurs in the program.

APPENDIX A: DATA DEPENDENCIES

This appendix discusses how to determine the data dependencies that appear in a program's program dependence graph. We present definitions, expressed with attribute grammars, that describe how to generate the PDG's loop-independent flow dependency edges, loop-carried flow dependency edges, and def-order dependency edges. For each case, the definition is presented as an attribute grammar over the following (ambiguous) context-free grammar:

Root → S
S → *Id* := *Exp*
S_1 → S_2 ; S_3
S_1 → **if** *Exp* **then** S_2 **else** S_3 **fi**
S_1 → **while** *Exp* **do** S_2 **od**

(In this grammar, the subscripts on the S's are not meant to indicate different nonterminals; they are used to distinguish between the different occurrences of S in each production. For example, in the second production, the three occurrences of nonterminal S are labeled S_1, S_2, and S_3). We do not actually provide an explicit method for computing the quantities being defined; however, all of the attribute grammars discussed below are noncircular, which means that their attributes can be computed by any of a number of

algorithms for attribute evaluation[3].

Reaching definitions

One way to compute data dependencies involves first computing a more general piece of information: the set of *reaching definitions* for each program point. A definition of variable x at some program point q *reaches* point p if there is an execution path from q to p such that no other definition of x appears on the path. The set of reaching definitions for a program point p is the set of definitions that reach p. A program's sets of reaching definitions may be expressed with an attribute grammar using four attributes for each S nonterminal: *exposedDefs*, *killedVars*, *reachingDefsBefore*, and *reachingDefsAfter*.

The *reachingDefsBefore* attribute is an inherited attribute of S; the other three are synthesized attributes of S. The values of *killedVars* attributes are sets of variable names; *S.killedVars* consists of the set of variables that must be assigned to when S executes. The values of the *exposedDefs*, *reachingDefsBefore*, and *reachingDefsAfter* attributes are sets of triples that record information about a program's definitions. Each triple is of the form:

<variable name, program location, program location>

The first program location that occurs in a triple indicates the position of a definition; the second program location either contains the value **null** or the location of one of the program's loops. A triple for definition d is an element of *S.exposedDefs* if d is defined within S and is downwards exposed in S, that is, if it reaches the end of S. *S.reachingDefsBefore* and *S.reachingDefsAfter* are the definitions that reach the beginning and end of S, respectively.

The relationships between these attributes are expressed by the equations of the grammar presented in Figure 15.

The attribute *S.killedVars* consists of variables that are guaranteed to be assigned to within S. Thus, if S is an assignment statement, there is always an assignment made to the *Id* of the statement. Because the body of a **while**-loop may never execute, the loop as a whole is not guaranteed to make an assignment to any variable.

A definition d is in *S.exposedDefs* if d is a definition within S and d reaches the end of S. Thus, if S is an assignment statement, the triple <*Id*, *&S*, **null**> is in *S.exposedDefs* because the definition of *Id* reaches the end of S. (The notation "*&S*" denotes the program point represented by S). For the statement-concatenation production, the definitions reaching the end of S_1 are the exposed definitions from S_2 that also reach the end of S_3 together with the exposed definitions from S_3.

The *S.reachingDefsBefore* and *S.reachingDefsAfter* attributes consist of the definitions that reach the beginning and end of S, respectively. For example, in an assignment statement, the definitions in *reachingDefsAfter* are the definitions in *reachingDefsBefore* that are not killed by the assignment, together with <*Id*, *&S*, **null**>, which represents the assignment itself. In a **while** loop, S_1.*reachingDefsAfter* represents the set of definitions that possibly reach the end of the loop; it is given the value:

$$S_1.reachingDefsAfter = S_1.reachingDefsBefore \cup S_2.exposedDefs$$

S_1.*reachingDefsBefore* contributes the definitions that occur earlier than the **while** loop, and S_2.*exposedDefs* contributes the exposed definitions that occur within the loop body. The S_2.*reachingDefsBefore* attribute of a **while** loop is handled slightly differently because it necessary to tag the definitions in S_2.*exposedDefs* with $\&S_1$ to indicate the loop carrying these reaching definitions:

---

[3]Readers who are not familiar with attribute grammars or attribute evaluation could consult either the original paper on attribute gram-

attributes

S.exposedDefs:	synthesized
S.killedVars:	synthesized
S.reachingDefsBefore:	inherited
S.reachingDefsAfter:	synthesized

attribute equations

Root $\rightarrow$ *S*

$S.reachingDefsBefore = \emptyset$

S $\rightarrow$ *Id* := *Exp*

$S.killedVars = \{Id\}$

$S.exposedDefs = \langle Id, \&S, \textbf{null}\rangle$

$S.reachingDefsAfter = \{ \langle i,a,l\rangle \mid \langle i,a,l\rangle \in S.reachingDefsBefore \wedge i \neq Id \} \cup \{\langle Id, \&S, \textbf{null}\rangle\}$

$S_1 \rightarrow S_2 \ ; \ S_3$

$S_1.killedVars = S_2.killedVars \cup S_3.killedVars$

$S_1.exposedDefs = \{ \langle i,a,l\rangle \mid \langle i,a,l\rangle \in S_2.exposedDefs \wedge i \notin S_3.killedVars \} \cup S_3.exposedDefs$

$S_2.reachingDefsBefore = S_1.reachingDefsBefore$

$S_3.reachingDefsBefore = S_2.reachingDefsAfter$

$S_1.reachingDefsAfter = S_3.reachingDefsAfter$

$S_1 \rightarrow$ **if** *Exp* **then** S_2 **else** S_3 **fi**

$S_1.killedVars = S_2.killedVars \cap S_3.killedVars$

$S_1.exposedDefs = S_2.exposedDefs \cup S_3.exposedDefs$

$S_2.reachingDefsBefore = S_1.reachingDefsBefore$

$S_3.reachingDefsBefore = S_1.reachingDefsBefore$

$S_1.reachingDefsAfter = S_2.reachingDefsAfter \cup S_3.reachingDefsAfter$

$S_1 \rightarrow$ **while** *Exp* **do** S_2 **od**

$S_1.killedVars = \emptyset$

$S_1.exposedDefs = S_2.exposedDefs$

$S_2.reachingDefsBefore = S_1.reachingDefsBefore \cup \{ \langle i,a,\&S_1\rangle \mid \langle i,a,\textbf{null}\rangle \in S_2.\text{exposedDefs} \}$

$S_1.reachingDefsAfter = S_1.reachingDefsBefore \cup S_2.exposedDefs$

Figure 15. An attribute grammar that describes the generation of a program's sets of reaching definitions.

$S_2.reachingDefsBefore = S_1.reachingDefsBefore \cup \{ \langle i,a,\&S_1\rangle \mid \langle i,a,\textbf{null}\rangle \in S_2.\text{exposedDefs} \}$

Flow dependencies

Having computed the reaching definitions for each statement S, we can determine the sources of flow dependency edges for each site where variables are used (*i.e.* in expressions in assignment statements, **if** statements, and **while** loops). In each case, the sources of flow dependencies are computed as a function of the value of the *reachingDefsBefore* attribute of the left-hand side S nonterminal. We assume that an *Exp* nonterminal has a synthesized attribute *used* whose value is the set of all variables used in *Exp*; an *Exp* is flow dependent on the set of vertices computed by restricting $S_1.reachingDefsBefore$ to the variables in *Exp.used*:

$\{ a \mid \langle i,a,l\rangle \in S_1.reachingDefsBefore \wedge i \in Exp.used \}$

By making some small changes to the equations of the attribute grammar given in Figure 15, we define two variations on reaching definitions that are used to compute the sources of loop-independent and loop-

mars [Knuth 1968] or any one of several compiler textbooks that discuss these matters, such as [Waite & Goos 1983].

carried flow dependencies for each use site. For instance, when the third attribute equation associated with the **while**-loop is changed from:

$$S_2.reachingDefsBefore = S_1.reachingDefsBefore \cup \{ <i, a, \&S_1> | <i, a, \mathbf{null}> \in S_2.\text{exposedDefs} \}$$

to:

$$S_2.reachingDefsBefore = S_1.reachingDefsBefore \quad (1)$$

the *reachingDefsBefore* and *reachingDefsAfter* attributes compute just the loop-independent reaching definitions. In this case, $\{ a | <i, a, l> \in S_1.reachingDefsBefore \wedge i \in Exp.used \}$ computes the sources of *Exp*'s loop-independent flow dependencies.

To compute loop-carried flow dependencies, the third equation associated with the **while**-loop is used in its original form:

$$S_2.reachingDefsBefore = S_1.reachingDefsBefore \cup \{ <i, a, \&S_1> | <i, a, \mathbf{null}> \in S_2.\text{exposedDefs} \}$$

However, the third attribute equation associated with the assignment statement is altered from:

$$S.reachingDefsAfter = \{ <i, a, l> | <i, a, l> \in S.reachingDefsBefore \wedge i \neq Id \} \cup <Id, \&S, \mathbf{null}>$$

to:

$$S.reachingDefsAfter = \{ <i, a, l> | <i, a, l> \in S.reachingDefsBefore \wedge i \neq Id \}$$

In this case, $\{ a | <i, a, l> \in S_1.reachingDefsBefore \wedge i \in Exp.used \}$ computes the sources of *Exp*'s loop-carried flow dependencies.

Def-order dependencies

Determining the def-order dependencies that occur in a program also depends on having computed the program's sets of reaching definitions. A program's sets of def-order dependencies may be expressed by attaching three additional attributes to each S nonterminal: *flowEdges*, *flowEdgesBefore*, and *flowEdgesAfter*. The *flowEdges* attribute is a synthesized attribute of S whose elements are pairs of program locations that represent the source and target of a flow edge whose target occurs in the program fragment that is subordinate to S. Thus, the value of *S.flowEdges* in the production $Root \rightarrow S$ represents the set of flow edges in the entire program.

The equations for the attributes *flowEdgesBefore* and *flowEdgesAfter*, shown in Figure 16, thread this information through the program left to right. At each assignment statement S, the set that is passed on to *S.flowEdgesAfter* is *S.flowEdgesBefore* without the flow edges whose source is S.

For each assignment statement S of a program, we use the value of *S.flowEdgesBefore* to compute the targets of all def-order edges whose source is S: a def-order edge $S \longrightarrow_{do(u)} t$ exists for each t such that

$$<s, u> \in S.flowEdgesBefore \wedge <t, u> \in S.flowEdgesBefore \wedge s = \&S$$

APPENDIX B: CONSOLIDATING SLICES OF A PROGRAM DEPENDENCE GRAPH

In Section 4.1.2 we defined the program slice G / s for a PDG G and a vertex s in G. The program slice definition extends to a set of vertices in G. It is a natural extension since the program slice of a set of program points is precisely the graph union of the program slices of the individual program points, as we show next.

attributes

S.flowEdges:	synthesized
S.flowEdgesBefore:	inherited
S.flowEdgesAfter:	synthesized

attribute equations

Root → *S*
  $S.flowEdgesBefore = S.flowEdges$

S → *Id* := *Exp*
  $S.flowEdges = \{ \langle a, \&S \rangle \mid \langle i, a, l \rangle \in S.reachingDefsBefore \wedge i \in Exp.used \}$
  $S.flowEdgesAfter = S.flowEdgesBefore - \{ \langle s, t \rangle \mid \langle s, t \rangle \in S.flowEdgesBefore \wedge s = \&S \}$

$S_1 \rightarrow S_2$; S_3
  $S_1.flowEdges = S_2.flowEdges \cup S_3.flowEdges$
  $S_2.flowEdgesBefore = S_1.flowEdgesBefore$
  $S_3.flowEdgesBefore = S_2.flowEdgesAfter$
  $S_1.flowEdgesAfter = S_3.flowEdgesAfter$

S_1 → **if** *Exp* **then** S_2 **else** S_3 **fi**
  $S_1.flowEdges = \{ \langle a, \&S_1 \rangle \mid \langle i, a, l \rangle \in S_1.reachingDefsBefore \wedge i \in Exp.used \}$
  $S_2.flowEdgesBefore = S_1.flowEdgesBefore$
  $S_3.flowEdgesBefore = S_1.flowEdgesBefore$
  $S_1.flowEdgesAfter = S_2.flowEdgesAfter \cap S_3.flowEdgesAfter$

S_1 → **while** *Exp* **do** S_2 **od**
  $S_1.flowEdges = \{ \langle a, \&S_1 \rangle \mid \langle i, a, l \rangle \in S_1.reachingDefsBefore \wedge i \in Exp.used \}$
  $S_2.flowEdgesBefore = S_1.flowEdgesBefore$
  $S_1.flowEdgesAfter = S_2.flowEdgesAfter$

Figure 16. An attribute grammar that describes the generation of a program's def-order dependency edges.

Theorem. For any collection $\bigcup_i s_i$ of program points, we have $\bigcup_i G / s_i = G / \bigcup_i s_i$.

Proof. The graph $\bigcup_i G / s_i$ consists of vertices $\bigcup_i V(G / s_i)$ and edges $\bigcup_i E(G / s_i)$.

(1) $\bigcup_i V(G / s_i) = V(G / \bigcup_i s_i)$ by the definition given in Section 4.1.2.

(2) (a) For each $u \longrightarrow w \in \bigcup_i E(G / s_i)$ we have $u \longrightarrow w \in E(G / s_i)$ for some i. Since $s_i \subseteq \bigcup_i s_i$, $u \longrightarrow w \in E(G / \bigcup_i s_i)$, so $\bigcup_i E(G / s_i) \subseteq E(G / \bigcup_i s_i)$.

(b) For each $u \longrightarrow_{f,c} w \in E(G / \bigcup_i s_i)$ we have $w \in V(G / \bigcup_i s_i) = \bigcup_i V(G / s_i)$. Hence $w \in V(G / s_i)$ for some i. Since $u \longrightarrow_{f,c} w$ we must have $u \in V(G / s_i)$ as well, so $u \longrightarrow_{f,c} w \in E(G / s_i) \subseteq \bigcup_i E(G / s_i)$.

For each $u \longrightarrow_{do(t)} w \in E(G / \bigcup_i s_i)$ we have $t \in V(G / \bigcup_i s_i) = \bigcup_i V(G / s_i)$ by the definition of def-order edges in a slice. Hence $t \in V(G / s_i)$ for some i. Since $u, w \longrightarrow_f t$ we have $t, u, w \in V(G / s_i)$, so $u \longrightarrow_{do(t)} w \in E(G / s_i) \subseteq \bigcup_i E(G / s_i)$.

Since G contains only flow, control and def-order edges, we have $E(G / \bigcup_i s_i) \subseteq \bigcup_i E(G / s_i)$.

Combining (a) and (b) we have $\bigcup_i E(G / s_i) = E(G / \bigcup_i s_i)$.

Combining (1) and (2) we have $\bigcup_i G / s_i = G / \bigcup_i s_i$.

If we recall that for $v \notin V(G)$ we defined G / v to be the empty graph, we may eliminate from the statement of the theorem the requirement that the s_i be vertices in G. Although in general the union of two

feasible program dependence graphs is itself not necessarily feasible, the theorem shows that this *is* the case for slices from the same feasible PDG, since their union is a slice and all slices of a feasible PDG are feasible.

We can use the theorem above to reduce the definition of G_M in Section 4.3 to the union of two slices. Starting with the definition of G_M

$G_M = (G_A / D_A) \cup (G_B / D_B) \cup (G / (\overline{D}_A \cap \overline{D}_B))$

we use $G / (\overline{D}_A \cap \overline{D}_B) = G_A / (\overline{D}_A \cap \overline{D}_B) = G_B / (\overline{D}_A \cap \overline{D}_B)$ and idempotency of $\cup$ to obtain

$G_M = (G_A / D_A \cup G_A / (\overline{D}_A \cap \overline{D}_B)) \cup (G_B / D_B \cup G_B / (\overline{D}_A \cap \overline{D}_B)$

and apply the theorem above to consolidate the slices:

$G_M = (G_A / (D_A \cup (\overline{D}_A \cap \overline{D}_B))) \cup (G_B / (D_B \cup (\overline{D}_A \cap \overline{D}_B)))$

Now we simplify the set terms using the fact that for $v \notin (\overline{D}_B - V(G_A))$, we have $G_A / v = \varnothing$ so that

$G_M = G_A / (D_A \cup \overline{D}_B) \cup G_B / (D_B \cup \overline{D}_A)$

Thus the merged graph G_M can be defined as the graph union of a slice of G_A with a slice of G_B.

APPENDIX C: THE INTRACTABILITY OF DECIDING WHETHER A PROGRAM DEPENDENCE GRAPH IS FEASIBLE

This appendix shows that the problem of deciding whether or not a program dependence graph is feasible is NP-complete. We call this problem the PDG-FEASIBILITY problem.

Theorem. The PDG-FEASIBILITY problem is NP-complete.

To see that PDG-FEASIBILITY is in NP, recall the following observation that was made in Section 4.5 when discussing ReconstituteProgram:

> Because we are assuming a restricted set of control constructs, each vertex of [PDG G] is immediately subordinate to at most one predicate vertex, *i.e.* the control dependencies of [G] define a tree T rooted at the entry vertex. The crux of the program-reconstitution problem is to determine, for each predicate vertex v (and for the entry vertex as well), an ordering on v's children in T.

In Section 4.5, we also assumed that we have been furnished the function TransformToSyntaxTree that converts a control-dependence tree T with ordered vertices into the corresponding abstract-syntax tree. Note that G is feasible if and only if there exists an ordering of T's vertices that yields a program P for which $G_P = G$. Clearly a nondeterministic algorithm for deciding feasibility can guess an ordering for the children of each predicate vertex in T. By creating the program P = TransformToSyntaxTree(T) and then testing whether $G_P = G$, it is possible to check whether G is feasible; both steps take at most time polynomial in the size of G.

To show that PDG-FEASIBILITY is NP-hard, we show that it contains an NP-hard problem, RESTRICTED-PDG-FEASIBILITY, as a special case. In particular, we restrict our attention to PDG's that contain assignment vertices but no predicate vertices and in which, for all variables x, there are no overlapping x-spans. A program dependence graph in this restricted class is feasible if and only if its vertices can be topologically ordered such that for all pairs $<d_1, d_2>$ of definitions of the same variable, either all vertices in Span(d_1) precede all vertices in Span(d_2) or *vice versa* (henceforth referred to as a *legal ordering*).

Theorem. RESTRICTED-PDG-FEASIBILITY is NP-hard.

Proof. We transform 3SAT (3 CNF Satisfiability) to RESTRICTED-PDG-FEASIBILITY as follows. Let $U = \{u_1, u_2, \ldots, u_n\}$ be the set of variables and $C = \{c_1, c_2, \ldots, c_n\}$ be the set of clauses in an arbitrary instance of 3SAT. Without loss of generality, we will assume that each clause of C contains no more than one barred or unbarred occurrence of each variable (*i.e.* there is no more than one u_i or $\bar{u}_i$ per clause). We will construct a PDG $G = (V, E)$ such that G has a legal ordering if and only if C is satisfiable.

For each (logical) variable $u_i \in U$, there is a (program) variable x_i and two x_i-spans in G, which we call X_i and $\bar{X}_i$. The heads of X_i and $\bar{X}_i$ represent the assignment statements $x_i := 1$ and $x_i := 0$, respectively.

For each occurrence of u_i or $\bar{u}_i$ in a clause $c_j \in C$, there is a program variable x_i^j and two assignment vertices in $V(G)$, v_{ij} and $\bar{v}_{ij}$. Both v_{ij} and $\bar{v}_{ij}$ represent assignment statements of the form $x_i^j := x_i$; however, v_{ij} occurs in the span X_i, whereas $\bar{v}_{ij}$ occurs in the span $\bar{X}_i$. For each occurrence of u_i or $\bar{u}_i$ in $c_j \in C$, there is also a second program variable, y_i^j, and two additional assignment vertices, w_{ij} and $\bar{w}_{ij}$. Both w_{ij} and $\bar{w}_{ij}$ represent assignment statements of the form $y_i^j := \cdots x_i^j \cdots$ (*i.e.* x_i^j is used on the right-hand side of the assignment; in some cases, other variables will be used on the right-hand side of the assignment as well); however, w_{ij} occurs in the span headed by v_{ij}, whereas $\bar{w}_{ij}$ occurs in the span headed by $\bar{v}_{ij}$.

To test whether the individual clauses of C are satisfied, we introduce some additional edges in $E(G)$ as a satisfaction-testing component. The purpose of the edges is to encode the conditions for satisfying a clause into the ordering dependencies of the graph.

For each clause $c_j = z_1 \vee z_2 \vee z_3$, where each of the z_k represents an occurrence of some variable u_i (either barred or unbarred), we define the right-hand sides for three of the w_{kj} vertices so as to introduce three flow edges to represent the following pairings: (z_1, z_2), (z_2, z_3), and (z_3, z_1). For each pair we introduce an edge whose source is the w vertex that corresponds to the first member of the pair and whose target is the "complement" of the w vertex that corresponds to the second member of the pair. For example, for the clause $c_j = \bar{u}_1 \vee u_2 \vee \bar{u}_3$, we define vertex $\bar{w}_{2j}$ to be "$y_2^j := x_2^j + y_1^j$", and introduce edge $(\bar{w}_{1j}, \bar{w}_{2j})$; we define vertex w_{3j} to be "$y_3^j := x_3^j + y_2^j$", and introduce edge (w_{2j}, w_{3j}); and define vertex w_{1j} to be "$y_1^j := x_1^j + y_3^j$", and introduce edge $(\bar{w}_{3j}, w_{1j})$. The subgraph corresponding to this clause is shown in Figure 17.

Clearly the construction of G can be accomplished in polynomial time. What remains to be shown is that G has a legal ordering if and only if C is satisfiable.

First, suppose that O is a legal ordering of the vertices of G. The truth assignment for each variable u_i of U is obtained according to the relative order in O of the heads of X_i and $\bar{X}_i$: variable u_i is assigned the value T if and only if the head of X_i precedes the head of $\bar{X}_i$. The constraints of the spans X_i and $\bar{X}_i$ ensure that the vertices v_{ik} and $\bar{v}_{ik}$ have the same relative order as the heads of X_i and $\bar{X}_i$; similarly, the vertices w_{ik} and $\bar{w}_{ik}$ have the same relative order as v_{ik} and $\bar{v}_{ik}$.

If all variable occurrences in some clause c had the value F, then there would be a cyclic set of constraints among the vertices of c's satisfaction-testing component. Because O is a legal ordering and hence a topological ordering, at least one variable occurrence in each clause must have the value T.

Conversely, suppose that $t: U \longrightarrow \{T, F\}$ is a satisfying truth assignment for C. A corresponding legal ordering O can be obtained by adding some additional edges to G and then topologically sorting the resulting graph. For each pair of x-spans, edges are introduced to force all of the vertices of one span to precede all of the vertices of the other span.

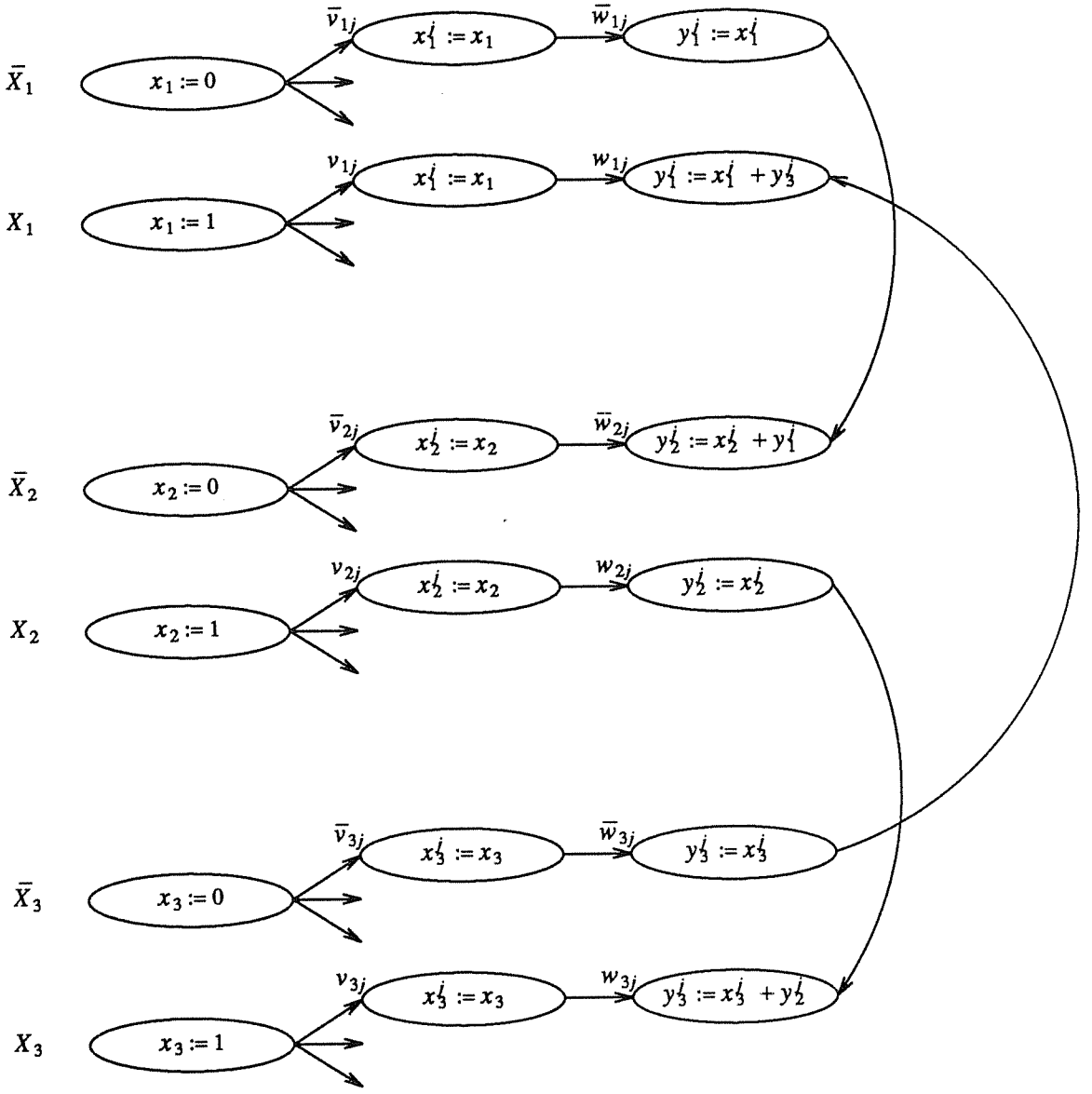


Figure 17. Subgraph representing the clause $c_j = \bar{u}_1 \vee u_2 \vee \bar{u}_3$.

a) If u_i is a variable for which $t(u_i) = T$, then for all clauses c_k that contain either u_i or $\bar{u}_i$, we add the edges:
$w_{ik} \longrightarrow \bar{v}_{ik}$
$v_{ik} \longrightarrow HeadOf\ (\bar{X}_i)$

b) If u_i is a variable for which $t(u_i) = F$ then for all clauses c_k that contain either u_i or $\bar{u}_i$, we add the edges:

$\overline{w}_{ik} \longrightarrow v_{ik}$
$\overline{v}_{ik} \longrightarrow HeadOf\ (X_i)$

Assuming the truth assignment x_1 = F, x_2 = F, x_3 = T, the augmented version of the graph of Figure 17 is shown in Figure 18. What remains to be shown is that this augmented graph has a legal ordering.

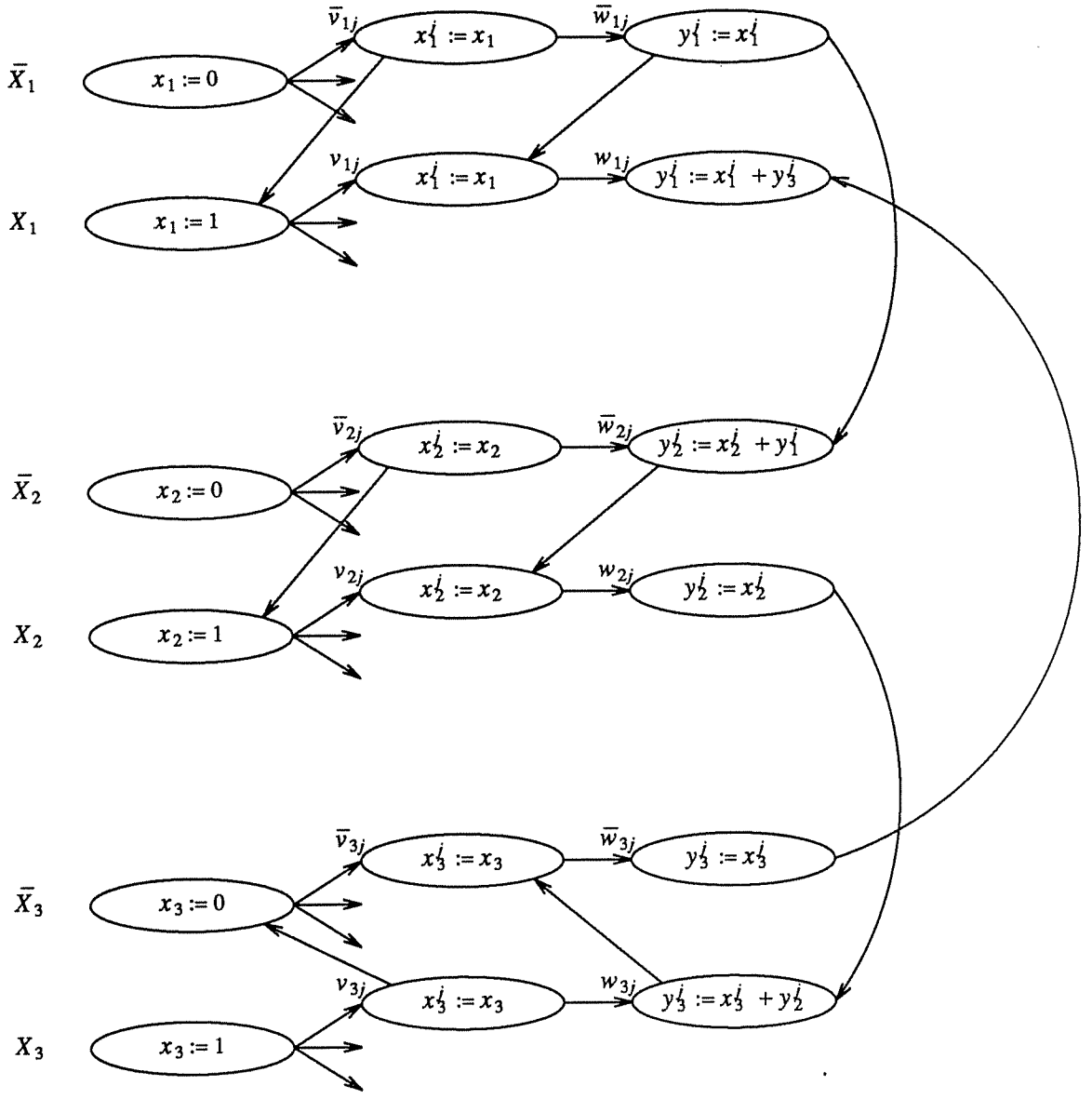


Figure 18. Augmented subgraph representing the clause $c_j = \overline{u}_1 \vee u_2 \vee \overline{u}_3$, and assuming the truth assignment x_1 = F, x_2 = F, x_3 = T.

If u_i is a variable for which $t(u_i) = T$, then the head of X_i has in-degree 0 and the remaining vertices of X_i have in-degree 1. (Note that X_i includes the vertex "$x_i := 1$" and all v_{ij} vertices, but *not* the w_{ij} vertices, which are in the x_i^j-spans). The vertices of X_i can be placed next in the ordering, and then the vertices and their out-edges can be removed from the graph. At this point, the head of $\overline{X}_i$ has in-degree 0 and can be processed as well.

If u_i is a variable for which $t(u_i) = F$, the corresponding operations are: first process the vertices of the span $\overline{X}_i$ and then process the head of X_i.

At this point, each clause corresponds to a group of nine vertices that have no interconnections with the vertices for any other clause. This situation is illustrated in Figure 19. The in-degree of each vertex in a group is at most 2. Because of the way edges were added between (barred and unbarred) w_{ik} vertices and (barred and unbarred) v_{ik} vertices, the only way a cycle could occur in a group is if each variable of the clause were assigned the value F. However, because t is a satisfying assignment, each clause has at least one variable occurrence with value T. Therefore, there is no cycle, and the group of nine vertices can be ordered.

REFERENCES

[Aho et al. 1986]
Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.

[Allen & Kennedy 1982]
Allen, J.R. and Kennedy, K. PFC: A program to convert FORTRAN to parallel form. TR 82-6, Dept. of Math. Sciences, Rice Univ., Houston, Tex., Mar. 1982.

[Allen & Kennedy 1984]
Allen, J.R. and Kennedy, K. Automatic loop interchange. In Proceedings of the SIGPLAN 84 Symposium on Compiler Construction Montreal, Can., June 20-22, 1984, pp. 233-246.

[Dijkstra 1976]
Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Ferrante & Mace 1984]
Ferrante, J. and Mace, M. On linearizing parallel code. In Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages, New Orleans, La., Jan. 14-16, 1985, pp. 179-189.

[Ferrante et al. 1987]
Ferrante, J., Ottenstein, K., and Warren, J. The program dependence graph and its use in optimization. To appear in *ACM Trans. on Prog. Lang. and Syst.* Preliminary version appeared in *Lecture Notes in Computer Science*, vol. 167: *6th Int. Symp. on Programming* (Toulouse, France, Apr. 1984), Springer-Verlag, New York, 1984, pp. 125-132.

[Hoare 1969]
Hoare, C.A.R. An axiomatic basis for computer programming. *Comm of the ACM 12*, 10 (Oct. 1969), 576-583.

[Hunt & McIlroy]
Hunt, J.W. and McIlroy, M.D. An algorithm for differential file comparison. Computing Science Tech. Rep. 41, Bell Laboratories, Murray Hill, N.J.

[Knuth 1968]
Knuth, D.E. Semantics of context-free languages. *Math. Syst. Theory 2*, 2 (June 1968), 127-145. Correction. *ibid. 5*, 1 (Mar. 1971), 95-96.

[Kuck 1978]
Kuck, D.J. *The Structure of Computers and Computations, Vol. 1.* John Wiley and Sons, New York, 1978.

[Kuck et al. 1972]
Kuck, D.J., Muraoka, Y., and Chen, S.C. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Trans. on Computers C-21* (Dec. 1972), 1293-1310.

[Kuck et al. 1981]
Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M. Dependence graphs and compiler optimizations. In Conference Record of the Eighth ACM Symposium on Principles of Programming Languages, Williamsburg, Va., Jan. 26-28, 1981, pp. 207-218.

[Ottenstein & Ottenstein 1984]
Ottenstein, K. and Ottenstein, L. The program dependence graph in a software development environment. In Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh,

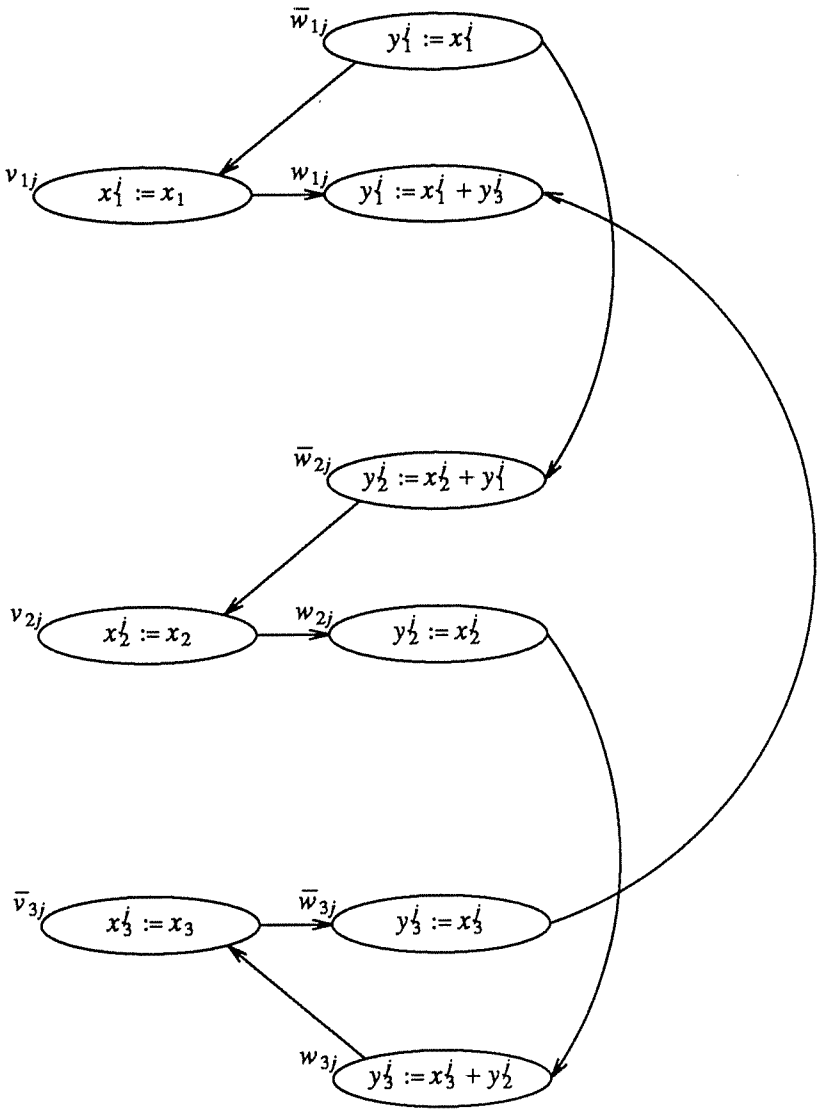


Figure 19. Topological sort in progress on the augmented subgraph of Figure 18. At this point the clause c_j corresponds to the nine vertices shown here.

Penn., Apr. 23-25, 1984. Appeared as joint issue: *SIGPLAN Notices* (ACM) *19*, 5 (May 1984), and *Soft. Eng. Notes* (ACM) *9*, 3 (May 1984), 177-184.

[Tichy 1982]
Tichy, W.F. Design, implementation, evaluation of a revision control system. In Proceedings of the Sixth International Conference on Software Engineering (Tokyo, Japan, Sept. 13-16, 1982), pp. 58-67.

[Towle 1976]
Towle, R. Control and data dependence for program transformations. Ph. D. dissertation and Tech. Report 76-788, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, Illinois, Mar. 1976.

[Waite & Goos 1983]
Waite, W.M. and Goos, G. *Compiler Construction*. Springer-Verlag, New York, 1983.

[Weiser 1982]
Weiser, M. Programmers use slices when debugging. *Commun. ACM 25*, 7 (July 1982), 446-452.

[Weiser 1984]
Weiser, M. Program slicing. *IEEE Trans. on Softw. Eng. SE-10*, 4 (July 1984), 352-357. Preliminary version appeared in Proceedings of the Fifth Int. Conf. on Software Engineering, (San Diego, Calif., Mar. 9-12, 1981), pp. 439-449.