Integrating Noninterfering 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. To date, the only available tools for assisting with program integration are variants of *text-based* differential file comparators; these are of limited utility because one has no guarantees about how the program that is the product of an integration behaves compared to the programs that were integrated.

This paper concerns the design of a *semantics-based* tool for automatically integrating program versions. The main contribution of the paper is an algorithm 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), the algorithm produces a program *M* that integrates *A* and *B*. The algorithm is predicated on the assumption that differences in the *behavior* of the variant programs from that of *Base*, rather than differences in the *text*, are significant and must be preserved in *M*. Although it is undecidable 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, the integration algorithm employs a program representation that is similar (although not identical) to the *dependence graphs* that have been used previously in vectorizing and parallelizing compilers. The algorithm also makes use of the notion of a *program slice* to find just those statements of a program that determine the values of potentially affected variables.

The program-integration problem has not been formalized previously. 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.

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**]:

A preliminary version of this paper appeared in the *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15, 1988), ACM, New York, 1988 [16].

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and DCR-8603356, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from IBM, DEC, Siemens, and Xerox.

Authors' current addresses: S. Horwitz and T. Reps, Computer Sciences Department, University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706; Jan Prins, Department of Computer Science, Sitterson Hall 083a, University of North Carolina, Chapel Hill, NC 27514.

Management—*programming teams, software configuration management*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, optimization*; E.1. [**Data**]: Data Structures—*graphs*

General Terms: Algorithms, Design, Languages, Management, Theory

Additional Key Words and Phrases: Control dependence, data dependence, data-flow analysis, dependence graph, program integration, program slice

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. 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 in a manner that combines their separate features.

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 must 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 appreciate 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.

A text-based approach has the advantage of being applicable to merging documents, data files, and other text objects as well as to merging programs. Unfortunately, this approach is 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 the 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.

® UNIX is a trademark of AT&T Bell Laboratories.

This paper describes a radically different approach based on the assumption that any change in the *behavior*, rather than the *text*, of a variant with respect to the base program is significant and must be preserved in the merged program. We present an algorithm, called *Integrate*, that could serve as the basis for building an automatic program-integration tool. Algorithm Integrate takes as input three programs A, B, and *Base*, where A and B are two variants of *Base*.[1] Algorithm Integrate either determines that the changes made to *Base* to produce A and B may interfere (in a sense defined in Sections 2 and 4.4), or it produces a new program M that integrates A and B with respect to *Base*. To find those components of a program that represent potentially changed behavior, algorithm Integrate makes use of *dependence graphs*, similar to those that have been used previously for representing programs in vectorizing and parallelizing compilers [2, 4, 11, 22], and an operation on these graphs called *program slicing* [24, 30].

A preliminary implementation of a program-integration tool based on the algorithm presented here has been embedded in a program editor created using the Synthesizer Generator [25, 26]. Data-flow analysis on programs is carried out according to the editor's defining attribute grammar and used to construct the programs' dependence graphs. An integration command invokes the integration algorithm, reports whether the variant programs interfere, and, if there is no interference, creates the integrated program.

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

The paper is organized into seven sections. 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 correspond to the five steps of algorithm Integrate. The first step is to build the dependence graphs that represent the programs *Base*, A, and B (the dependence graph that represents program P is denoted by G_P). Section 4.1 defines program dependence graphs and the operation of program slicing. The second step, discussed in Section 4.2, uses program slicing to determine sets of *affected points* of G_A and G_B as computed with respect to G_{Base}. These sets capture the essential differences between *Base* and the variant programs. 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.

[1] In fact, the approach we describe can accommodate any number of variants, but for the sake of exposition we consider the common case of two variants A and B.

Although, as we have shown in [18], the problem of determining whether G_M 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. Section 4.6 summarizes algorithm Integrate, states a theorem that characterizes how the semantics of the integrated program relates to the semantics of programs *Base*, *A*, and *B*, and discusses the algorithm's complexity.

Section 5 discusses applications of program integration in program-development environments. Section 6 describes related work, concentrating on the technical differences between the kind of dependence graphs we employ and the dependence representations that have been defined by others. Section 7 discusses some of the issues we have addressed in extending our work and outlines some problems for future research.

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 [8, 13], 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* that halts 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* halts and 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 halts, such that the specification $\{P_A \wedge P_B\}\ M\ \{Q_A \wedge Q_B\}$ is satisfied.

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 synthesis. 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, this integration criterion is not suitable at the present time as the basis for building a usable program-integration system.

Given the problems inherent in specification-based integration, we chose to investigate a different definition of the program-integration problem (with a different interference criterion). While specification-based integration ignores program *Base*, *Base* plays an important role in our approach. Our basic assumption is that any change in the *behavior* of the variants with respect to *Base* is significant and must be preserved in *M*. A further assumption is that the integrated version *M* must be composed of exactly the statements and control structures that appear as components of *Base*, *A*, and *B*.

Our notion of changed behavior in program *A* (respectively, *B*) with respect to *Base* is roughly the following: if there exists an initial state and variable *x* for which the final value of *x* computed by *Base* is different from the final value computed by *A* (*B*), then the computation of *x* is considered to be a change in

behavior of A (B) with respect to *Base*. The goal of program integration is to produce a program M that preserves the changed behaviors of both A and B with respect to *Base* (i.e., if *Base* and A (B) disagree on the final value of x, then M agrees with A (B)) and also preserves the behaviors that are *unchanged* in both A and B with respect to *Base* (i.e., if *Base*, A, and B all compute the same final value of x, then M also computes that final value). Variants A and B *interfere* with respect to *Base* if there exists an initial state and variable x such that *Base*, A, and B each compute different final values for x.

Although it is undecidable whether a program modification actually leads to a change in behavior, it is still possible to base an algorithm on this definition of program integration. In particular, it is possible to determine a safe approximation of (i.e., a superset of) the set of changed computations. To compute this information, we use a *dependence-graph* representation of programs similar to those used previously for representing programs in vectorizing and parallelizing compilers [2, 4, 11, 22]. We also use *program slices* [24, 30] to find just those components 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.)

To simplify the program-integration problem to a manageable level, we allow ourselves two further assumptions. First, we confine our attention to a simplified programming language with the following characteristics:[2] expressions contain only scalar variables and constants; statements are either assignment statements, conditional statements, while loops, or a restricted kind of "output statement" called an *end statement*, which can only appear at the end of a program. An end statement names one or more of the variables used in the program. The variables named in the end statement are those whose final values are of interest to the programmer; when execution terminates, the final state is defined on only those variables in the end statement. Thus a program is of the form:

program
  stmt_list
end(*id**)

Second, we make two assumptions about the editor used to create variants A and B from copies of *Base*.

(1) The editor provides a tagging capability so that common components (i.e., statements and predicates) can be identified in all three versions. Each component's tag is guaranteed to persist across different editing sessions and machines; tags are allocated by a single server, so that two different editors cannot allocate the same new tag.

(2) The operations on program components supported by the editor are insert, delete, and move. When editing a copy of *Base* to create a variant, a newly inserted component is given a previously unused tag; the tag of a component that is deleted is never reused; a component that is moved from its original position in *Base* to a new position in the variant retains its tag from *Base*.

[2] We believe that our approach to program integration can be extended to more realistic programming languages. For example, we have made some progress in extending the algorithm to handle languages with procedure calls [19] and with pointer variables [15].

A tagging facility meeting these requirements can be supported by language-based editors, such as those that can be created by such systems as MENTOR [9], GANDALF [12, 23], and the Synthesizer Generator [25, 26].

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. There is one aspect of this goal that is not addressed by the algorithm described in this paper. In particular, when the final step of the integration algorithm 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 desirable 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 take into account program semantics. This has two undesirable consequences:

(1) If the variants of the base program do interfere (under a semantic criterion), *diff3* still goes ahead and produces an "integrated" program.
(2) Even when the variants do not interfere (under a semantic criterion), 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

```
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
end(y, z)
```

Variant *A*

```
program
  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
end(y, z)
```

Variant *B*

```
program
  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
end(y, z)
```

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

```
program
  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
end(y, z)
```

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

```
program
  if Q then x := 1 fi
  if P then x := 0 fi
  y := x
end(y, z)
```

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." Under some versions of

UNIX, 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

```
program
<<<<<<< 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
end(y, z)
```

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:

```
program
    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
  end(y, z)
```

```
program
  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
end(y, z)
```

```
program
  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
end(y, z)
```

In contrast to 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*.

4. AN ALGORITHM FOR INTEGRATING NONINTERFERING VERSIONS OF PROGRAMS

4.1 The Program Dependence Graph

Different definitions of program dependence representations have been given, depending on the intended application; they are all variations on a theme introduced in [21], and share the common feature of having an explicit representation of data dependences (see below). The "program dependence graphs" defined in [11] introduced the additional feature of an explicit representation for control dependences (see below). The definition of program dependence graph given below differs from [11] in two ways. First, our definition covers only the restricted language described earlier, and hence is less general than the one given in [11]. Second, because of the particular needs of the program-integration problem, we omit certain classes of data dependence edges and introduce one new class; reasons for these changes are provided in Section 6.1. Despite these

differences, the structures we define and those defined in [11] share the feature of explicitly representing both control and data dependences; therefore, we refer to our graphs as "program dependence graphs," borrowing the term from [11].

The *program dependence graph* (or PDG) for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges.[3] Program dependence graph G_P includes four kinds of vertices:

(1) For each assignment statement and control predicate that occurs in program P, there is a vertex labeled with the assignment or predicate.

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

(3) For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined (see [1]), there is a vertex called the *initial definition of* x. This vertex represents an assignment to x from the initial state. The vertex is labeled "$x := InitialState(x)$."

(4) For each variable x named in P's end statement, there is a vertex called the *final use of* x. This vertex represents an access to the final value of x computed by P, and is labeled "$FinalUse(x)$."

We assume that vertices of PDGs are also labeled with an additional piece of information (which is not shown in our examples). Recall that we have assumed that the editor used to modify programs provides a tagging capability. Vertices of a PDG are labeled with the tags of the corresponding program components.

The edges of G_P represent *dependences* between program components. An edge represents either a *control dependence* or a *data dependence*. Control dependence edges are labeled either **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \rightarrow_c v_2$, 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 dependence edges for arbitrary programs is given in [11]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under consideration here, the control dependence edges reflect a program's nesting structure; program dependence graph G_P contains a *control dependence edge* from vertex v_1 to vertex v_2 iff one of the following holds:

(1) v_1 is the entry vertex, and v_2 represents a component of P that is not subordinate to any control predicate; these edges are labeled **true**.

(2) 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

[3] 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.

labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.[4]

Note that initial-definition and final-use vertices have no incoming control dependence edges.

A data dependence 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 dependence edges, representing *flow dependences* and *def-order dependences*.

The data dependence edges of a program dependence graph are computed using data-flow analysis. For the restricted language considered in this paper, the necessary computations can be defined in a syntax-directed manner (see [14]).

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

(1) v_1 is a vertex that defines variable x.

(2) v_2 is a vertex that uses x.

(3) 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 [1] 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 the beginning of the control-flow graph, and final uses of variables are considered to occur at its end.)

A flow dependence that exists from vertex v_1 to vertex v_2 will be denoted by $v_1 \rightarrow_f v_2$. (When it is necessary to indicate that a dependence is due to a particular variable x, it will be denoted by $v_1 \rightarrow_f^x v_2$).

Flow dependences are further classified as *loop independent* or *loop carried* [3]. A flow dependence $v_1 \rightarrow_f v_2$ is carried by loop L, denoted by $v_1 \rightarrow_{lc(L)} v_2$, if in addition to (1), (2), and (3) above, the following also hold:

(4) There is an execution path that both satisfies the conditions of (3) above and includes a backedge to the predicate of loop L; and

(5) Both v_1 and v_2 are enclosed in loop L.

A flow dependence $v_1 \rightarrow_f v_2$ is loop independent, denoted by $v_1 \rightarrow_{li} v_2$, if in addition to (1), (2), and (3) above, there is an execution path that satisfies (3) above and includes *no* backedge to the predicate of a loop that encloses both v_1 and v_2. It is possible to have both $v_1 \rightarrow_{lc(L)} v_2$ and $v_1 \rightarrow_{li} v_2$.

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

(1) v_1 and v_2 are both assignment statements that define the same variable.

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

[4] In other definitions that have been given for control dependence edges, there is an additional 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 is unnecessary for our purposes, and hence is left out of our definition.

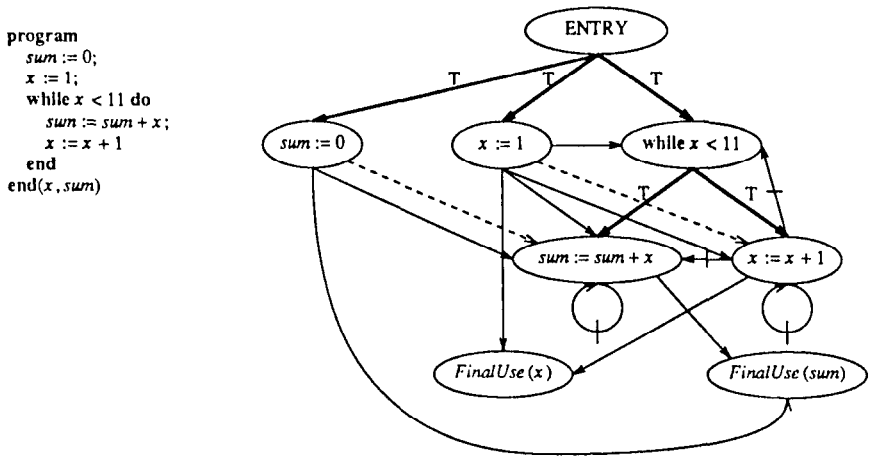


Fig. 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependence edges, dashed arrows represent def-order dependence edges, solid arrows represent loop-independent flow dependence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

(3) There exists a program component v_3 such that $v_1 \rightarrow_f v_3$ and $v_2 \rightarrow_f v_3$.
(4) v_1 occurs to the left of v_2 in the program's abstract syntax tree.

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

Note that a program dependence graph is a multigraph (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 dependence edge between two vertices, each is labeled by a different loop that carries the dependence. 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 dependence edges; dashed arrows represent def-order dependence edges; solid arrows represent loop-independent flow dependence edges; solid arrows with a hash mark represent loop-carried flow dependence edges.

4.1.1 *Def-order Dependences versus Anti- and Output Dependences.* Previous program dependence representations have included flow dependence edges as well as edges for two other kinds of data dependences, called *antidependences* and *output dependences.* (All three kinds may be further characterized as loop independent or loop carried.) Def-order dependences have not been previously defined. The definition of program dependence graphs given in Section 4.1 omits anti- and output dependences in favor of def-order dependences. Our reasons for using this definition are discussed in Section 6.1; this section merely clarifies the *differences* among these three kinds of dependences.

For flow dependences, antidependences, and output dependences, a program component v_2 has a dependence on component v_1 due to variable x only if

execution can reach v_2 after v_1 and there is no intervening definition of x along the execution path by which v_2 is reached from v_1. There is a flow dependence if v_1 defines x and v_2 uses x (a "write-read" dependence); there is an antidependence if v_1 uses x and v_2 defines x (a "read-write" dependence); there is an output dependence if v_1 and v_2 both define x (a "write-write" dependence).

Although def-order dependences resemble output dependences in that they are both "write-write" dependences, they are two different concepts. An output dependence $v_1 \rightarrow_o v_2$ 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 dependence $v_1 \rightarrow_{do} v_2$ 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 dependence $v_1 \rightarrow_{do} v_2$ but not $v_1 \rightarrow_o v_2$, 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 dependence, [1] $\rightarrow_{do([6])}$ [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 dependences are [1] $\rightarrow_o$ [3] and [3] $\rightarrow_o$ [4], but there is no output dependence [1] $\rightarrow_o$ [4].

4.1.2 *Program Slices.* For a vertex s of a program dependence graph 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): $V(G/s) = \{w \in V(G) \mid w \rightarrow_{c,f}^{*} s\}$. 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 $V(G/v) = \emptyset$ 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 $v \rightarrow_{do(u)} w$ is only included if, in addition to v and w, $V(G/S)$ also contains the vertex u that is directly flow dependent on the definitions at v and w. In terms of the three types of edges in a PDG, we have

$$\begin{aligned} E(G/S) = &\{(v \rightarrow_f w) \in E(G) \mid v, w \in V(G/S)\} \\ &\cup \{(v \rightarrow_c w) \in E(G) \mid v, w \in V(G/S)\} \\ &\cup \{(v \rightarrow_{do(u)} w) \in E(G) \mid u, v, w \in V(G/S)\} \end{aligned}$$

Example. Figure 2 shows the graph that results from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for x.

4.1.3 *Program Dependence Graphs and Program Semantics.* In choosing which dependence edges to include in our program dependence graphs, our goal has been to characterize partially programs that have the same behavior—two inequivalent programs should not have the same program dependence graph, although two equivalent programs may have different program dependence graphs.

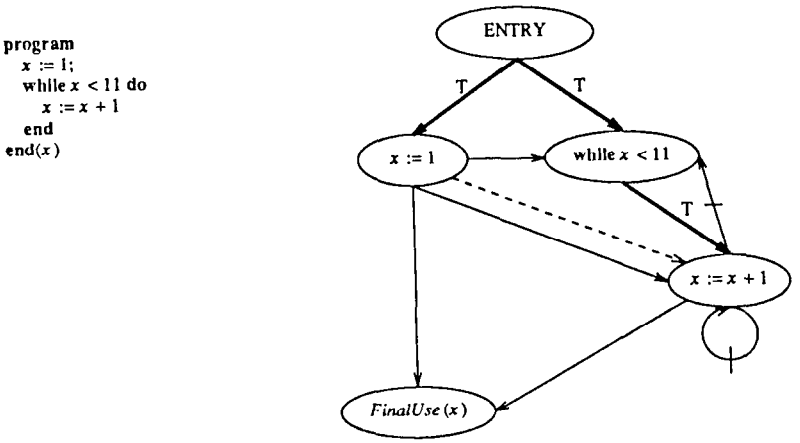


Fig. 2. The graph that results from slicing the example from Figure 1 with respect to the final-use vertex for x, together with the one program to which it corresponds.

This property is crucial to the correctness of our program-integration algorithm. In particular, the final step of the algorithm reconstitutes the integrated program from a program dependence graph. Because this graph may correspond to more than one program, we need to know that all such programs are equivalent.

The relationship between a program's PDG and the program's execution behavior has been addressed in [17, 18]. It is shown in [17, 18] that if the program dependence graphs of two programs are isomorphic, then the programs have the same behavior. It is also shown that if any of the different kinds of edges included in our definition of program dependence graphs were omitted, programs with different behavior could have the same program dependence graph. The concept of "programs with the same behavior" is formalized as the concept of *strong equivalence*, defined as follows:

Definition. Two programs P and Q are *strongly equivalent* iff for any state σ, either P and Q both diverge when initiated on σ or they both halt with the same final values for all variables. If P and Q are not strongly equivalent, we say they are *inequivalent.*

The term "divergence" refers to both nontermination (for example, because of infinite loops) and abnormal termination (for example, because of division by zero).

The main result of [17, 18] is the following theorem (we use the symbol $\approx$ to denote isomorphism between program dependence graphs):

THEOREM (Equivalence Theorem [17, 18]). *If P and Q are programs for which $G_P \approx G_Q$, then P and Q are strongly equivalent.*

Restated in the contrapositive, the theorem reads: Inequivalent programs have nonisomorphic program dependence graphs.

The relationship between a program's PDG and a slice of the PDG has been addressed in [27]. 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; it corresponds to the program P' obtained by restricting the syntax tree of P to just the statements and predicates in $V(G/S)$ [27].

THEOREM (Feasibility of Program Slices [27]). *For any program P, if G_S is a slice of G_P (with respect to some set of vertices), then G_S is a feasible PDG.*

Example. Figure 2 shows the one program that corresponds to the graph that results from slicing the graph in Figure 1 with respect to the final-use vertex for x.

The significance of a slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice [27]. In our case, a program point may be (1) an assignment statement, (2) a control predicate, or (3) a final use of a variable in an end statement. Because a statement or control predicate may be reached repeatedly in a program, by "computing the same sequence of values for each element of the slice," we mean: (1) for any assignment statement the same *sequence* of values is assigned to the target variable; (2) for a predicate the same *sequence* of Boolean values is produced; and (3) for each final use the same value for the variable is produced.

THEOREM (Slicing Theorem [27]). *Let Q be a slice of program P with respect to a set of vertices. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in G_Q: (1) Q halts on σ', (2) P and Q compute the same sequence of values at each program point of Q, and (3) the final states agree on all variables for which there are final-use vertices in G_Q.*

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 the behavior of a variant with respect to *Base* are significant. The program dependence graphs are a convenient representation from which to determine these changes.

Recall the assumption made in Section 4.1 that the vertices of a PDG are labeled with the tags maintained by the editor on program components. These tags provide a means for identifying PDG vertices that correspond in all three versions. It is these tags that are used to determine "identical" vertices when we perform operations on vertices from different PDGs (e.g., $V(G') - V(G)$). Similarly, when we speak below of "identical slices," where the slices are actually taken in different graphs, we mean that the slices are isomorphic under the mapping provided by the editor-supplied tags.

If the slice of variant G_A at vertex v differs from the slice of G_{Base} 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* $AP_{A,Base}$ of G_A as the subset of vertices of G_A whose slices

in G_{Base} and G_A differ:

$$AP_{A,Base} = \{v \in V(G_A) \mid (G_{Base}/v) \neq (G_A/v)\}.$$

The slice $G_A/AP_{A,Base}$ captures the behavior of A that differs from *Base*. Note that when there is a vertex v that is present in G_{Base} but not in G_A, any vertex still present in G_A that in G_{Base} depends on v is an affected point of G_A; thus, although such "deleted" vertices are not themselves affected points, they may have indirect effects on $AP_{A,Base}$ (and hence on $G_A/AP_{A,Base}$).

Example. Figure 1 shows a program that sums the integers from 1 to 10 and its corresponding program dependence graph. We now consider two variants of this program, shown in Figure 3 with their program dependence graphs:

(1) In variant A, two statements have been added to the original program to compute the product of the integer sequence from 1 to 10.
(2) In variant B, one statement has been added to compute the mean of the sequence.

These two programs represent noninterfering extensions of the original summation program. The set $AP_{A,Base}$ contains three vertices: the assignment vertices labeled "*prod* := 1" and "*prod* := *prod* * *x*" as well as the final-use vertex for *prod*. Similarly, $AP_{B,Base}$ contains two vertices: the assignment vertex labeled "*mean* := *sum*/10" and the final-use vertex for *mean*. Figure 4 shows the slices $G_A/AP_{A,Base}$ and $G_B/AP_{B,Base}$, which represent the changed behaviors of A and B, respectively.

There is a simple technique to determine $AP_{A,Base}$ 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:

(1) All vertices that are in G_A but not in G_{Base} are affected points.
(2) Each vertex w of G_A that has a different set of incoming control or flow edges in G_A than in G_{Base} gives rise to a set of affected points—those vertices that can be reached via zero or more control or flow edges from w.
(3) Each vertex w of G_A that has an incoming def-order edge $w' \rightarrow_{do(u)} w$ that does not occur in G_{Base} 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 (1) is straightforward: for $w \in V(G_A) - V(G_{Base})$, G_{Base}/w is the empty graph, whereas $w \in V(G_A/w)$, so G_A/w is not empty. The justification for observation (2) is also straightforward. By the definition of slicing, when w differs in incoming flow or control edges, G_A/w and G_{Base}/w cannot be the same, hence w itself is affected. For any vertex v that is (directly or indirectly) flow or control dependent on w in G_A, the slice G_A/v contains the subgraph G_A/w. Therefore, if w is affected, all *successors* of w in G_A via control and flow dependences are also affected.

The justification for observation (3) is more subtle. When a def-order edge $w' \rightarrow_{do(u)} w$ occurs in G_A but not in G_{Base}, then the slice G_A/u will include both w' and w and the def-order edge between them, while G_{Base}/u will not include this edge. Hence u is affected. The reverse situation, where $w' \rightarrow_{do(u)} w$

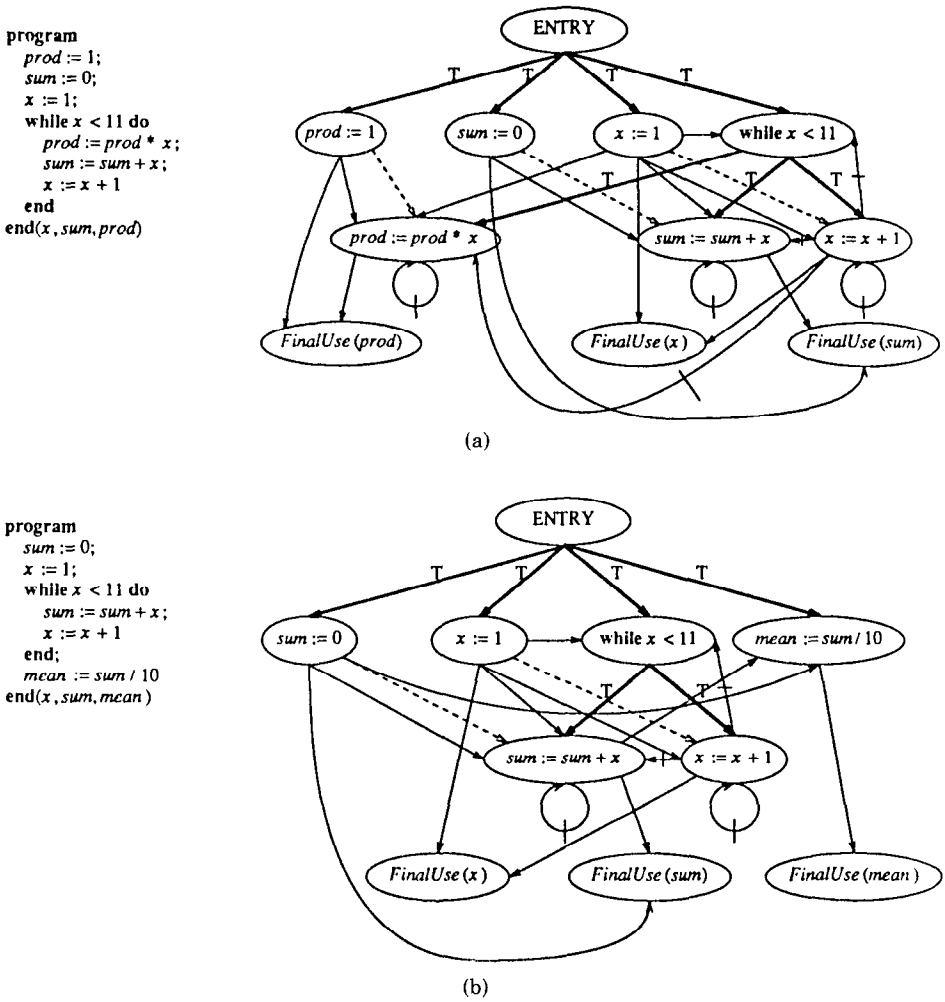


Fig. 3. Variants A and B of the base program shown in Figure 1, and their program dependence graphs.

occurs in G_{Base}, but not in G_A, means u is affected if $u \in V(G_A)$. But it is not necessary to examine this possibility since either $w' \rightarrow_{do(u)} w$ in G_{Base} is replaced by $w \rightarrow_{do(u)} w'$ in G_A, in which case $w' \in V(G_A)$ will contribute u as affected, or else one or both of the flow edges $w \rightarrow_f u$ and $w' \rightarrow_f u$ in G_{Base} 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 (directly or indirectly) flow or control dependent on u, the slice G_A/v contains the subgraph G_A/u; therefore, if u is affected, all *successors* of u via control and flow dependences are affected. Note that neither w' itself nor w itself is necessarily an affected point.

Observations (1), (2), and (3) serve to characterize the set of affected points. If $v \in V(G_A)$ is affected, there must be some w in G_A/v with different incoming edges in G_A and G_{Base}. By the arguments above, either w itself is an affected point

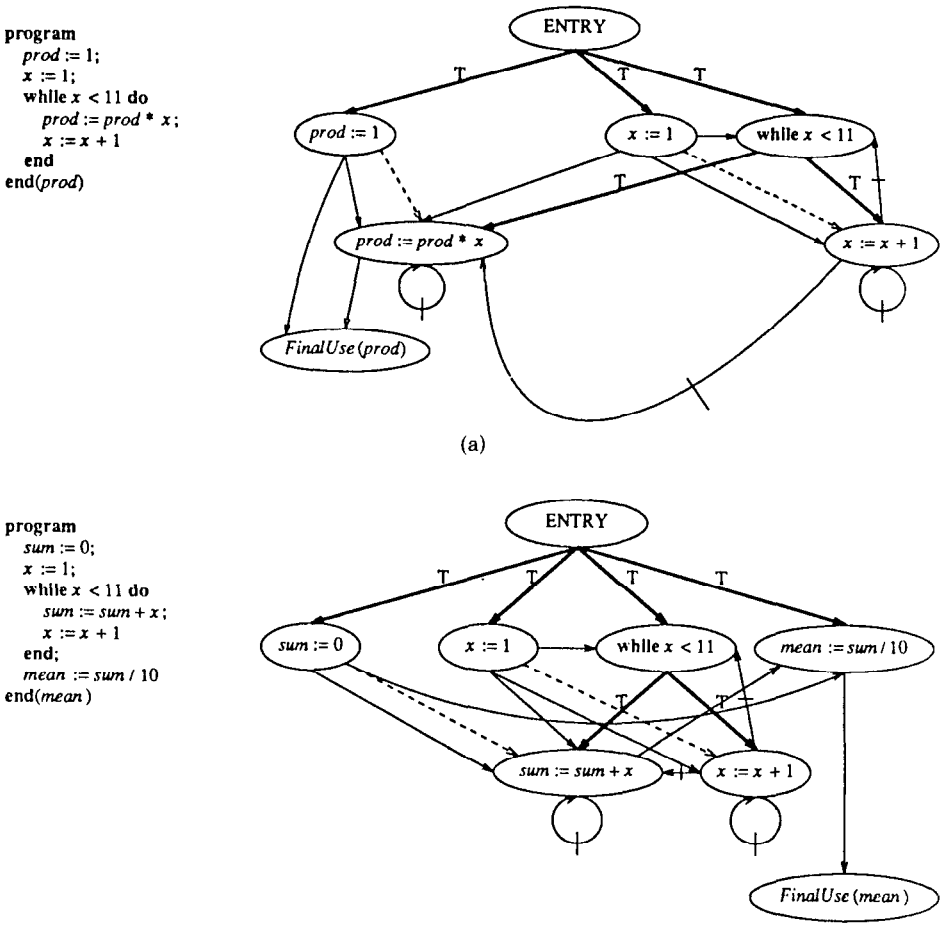


Fig. 4. The slices that represent the changed behaviors of A and B.

(cases (1) and (2)), or it contributes a vertex $u \in V(G_A/v)$ that is an affected point (case (3)); 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 function AffectedPoints(G', G), given in Figure 5.

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 worklist algorithm is used to find all vertices reachable from this set by flow or control edges.

4.3 Merging Program Dependence Graphs

We now show how to create the merged program dependence graph G_M. Graph G_M is formed by taking the union of three slices; these slices represent the

```
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 into S
      fi
      if the sets of incoming flow or control edges to w in G' are different from the incoming sets to w in G then
         Insert w into S
      fi
      for each def-order edge w' --->dd(u) w that occurs in G' but not in G do
         Insert u into S
      end
   end
   Answer := ∅
   while S ≠ ∅ do
      Select and remove an element b from S
      Insert b into Answer
      for each vertex c such that b --->f c or b --->c c is an edge in G' and c ∉ (Answer ∪ S) do
         Insert c into S
      end
   end
   return(Answer)
end
```

Fig. 5. The function AffectedPoints determines the points in the program dependence graph G' that may yield different values in G' than in G.

changed behaviors of A and B with respect to *Base* and the behavior of *Base* that is preserved in both A and B.

The previous section discussed how to compute the slices $G_A/AP_{A,Base}$ and $G_B/AP_{B,Base}$, which represent the changed behaviors of A and B with respect to *Base*. The slice that represents preserved behavior is computed similarly. If the slice of G_{Base} with respect to vertex v is identical to the slices of G_A and G_B with respect to vertex v, then all three programs produce the same sequence of values at v. We define the *preserved points* $PP_{Base,A,B}$ of G_{Base} as the subset of vertices of G_{Base} with identical slices in G_{Base}, G_A, and G_B:

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

The slice $G_{Base}/PP_{Base,A,B}$ captures the behavior of *Base* that is preserved in both A and B.

Example. When integrating the base program from Figure 1, variant A from Figure 3(a) and variant B from Figure 3(b), the slice $G_{Base}/PP_{Base,A,B}$ consists of G_{Base} in its entirety. That is, the graph that represents the behavior of the original program that is preserved in both variant A and variant B is identical to the graph shown in Figure 1.

The merged graph G_M is formed by taking the graph union of the slices that characterize the changed behavior of A, the changed behavior of B, and behavior of *Base* preserved in both A and B.

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

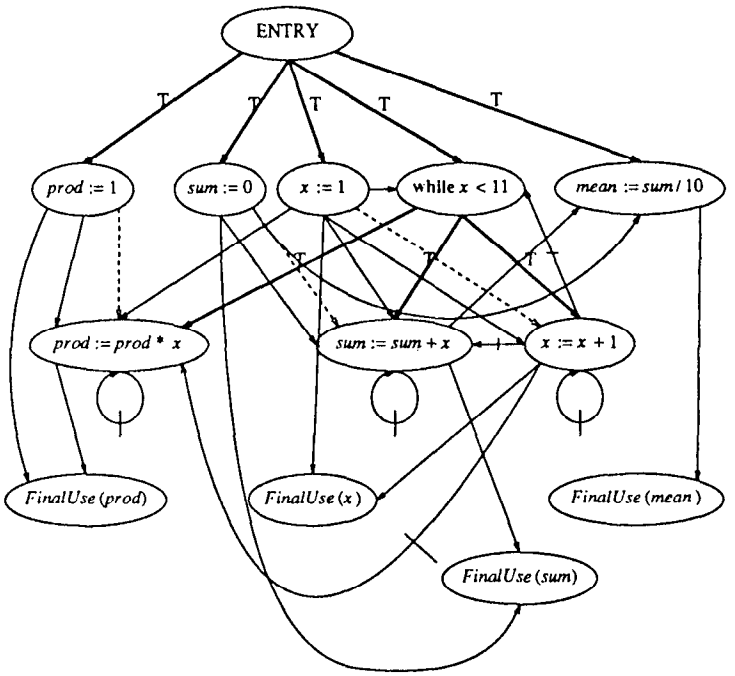


Fig. 6. G_M is created by taking the union of the graphs shown in Figures 4(a), 4(b), and 1.

Example. The merged graph G_M, shown in Figure 6, is formed by taking the union of the graphs shown in Figure 4(a), Figure 4(b), and Figure 1.

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 PDGs is not necessarily a feasible PDG, G_M may not be 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 noninterference criterion based on comparisons of slices of G_A, G_B, and G_M; the condition that must hold for the changed behavior of A and B to be preserved in G_M is

$$G_M/AP_{A,Base} = G_A/AP_{A,Base} \quad \text{and} \quad G_M/AP_{B,Base} = G_B/AP_{B,Base}.$$

On vertices in $PP_{Base,A,B}$ 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 affected points: G_M must agree with

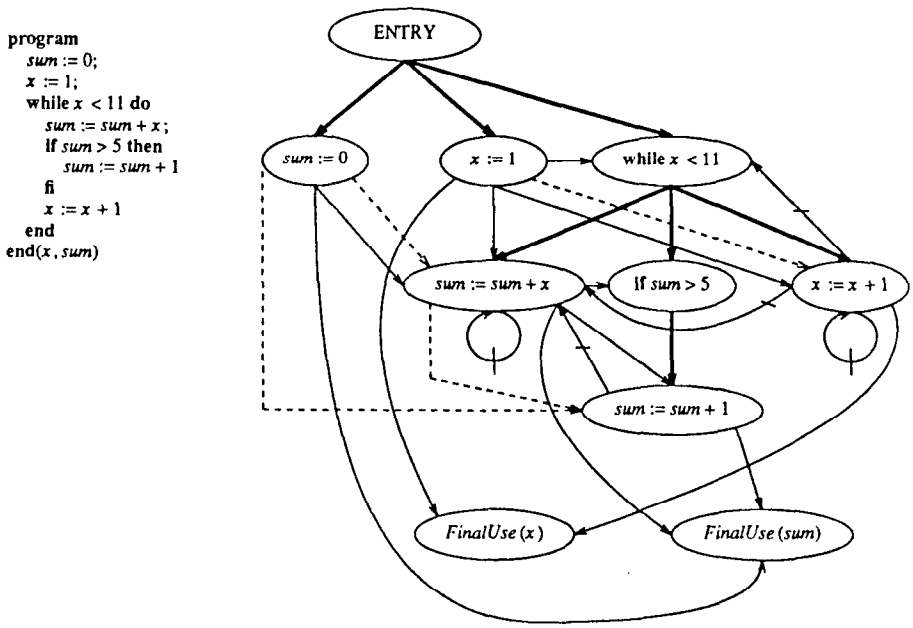


Fig. 7. Variant C and its program dependence graph.

variant A on $AP_{A,Base}$ and with B on $AP_{B,Base}$. Therefore, an easy way to test for noninterference (using function AffectedPoints) is to verify that

$$AP_{M,A} \cap AP_{A,Base} = \emptyset \quad \text{and} \quad AP_{M,B} \cap AP_{B,Base} = \emptyset.$$

Example. An inspection of the merged graph shown in Figure 6 reveals that there is no interference; the slices $G_M/AP_{A,Base}$ and $G_M/AP_{B,Base}$ are identical to the graphs that appear in Figures 4(a) and 4(b), respectively.

To illustrate interference, consider integrating the base program of Figure 1, variant B from Figure 3(b), and variant C from Figure 7. As in the previous integration example, the slice $G_B/AP_{B,Base}$ is shown in Figure 4(b); the slice $G_C/AP_{C,Base}$ includes all of the vertices of variant C except for $FinalUse(x)$. The merged graph is shown in Figure 8.

Variants B and C interfere (with respect to *Base*) because B's changed behavior (with respect to *Base*) is not preserved in the merged graph G_M. In particular, the vertex "*mean* := *sum*/10" is an affected point of B with respect to *Base*, but the slice G_M/"*mean* := *sum*/10" includes vertices "*sum* := *sum* + 1" and "**if** *sum* > 5", which are not included in the slice G_B/"*mean* := *sum*/10."

4.5 Reconstituting a Program From the Merged Program Dependence Graph

The final step of the integration algorithm involves reconstituting a program from the merged program dependence graph. Given a program dependence graph G_M that was created by merging variants A and B, function ReconstituteProgram

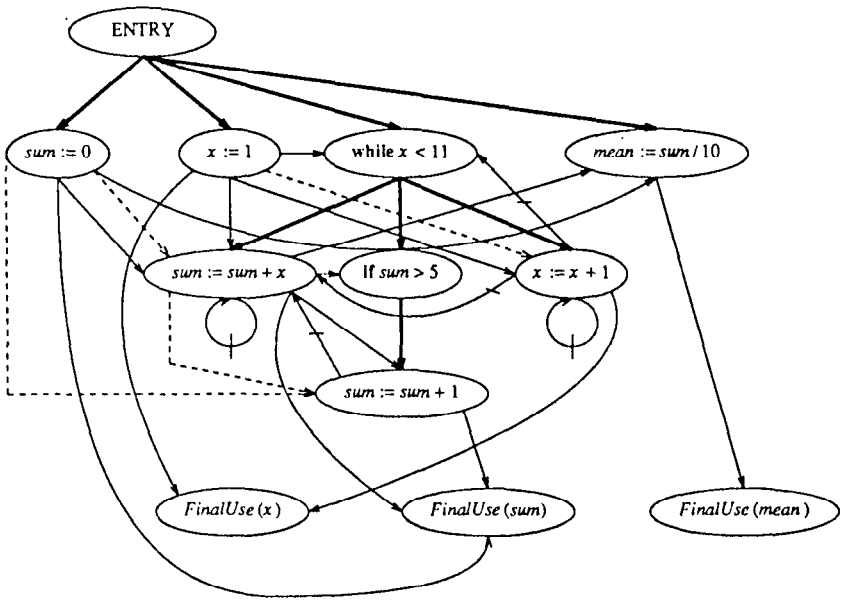


Fig. 8. The merged program dependence graph G_M resulting from the integration of *Base*, *B*, and *C*.

must determine whether G_M is feasible (i.e., corresponds to some program), and, if it is, create an appropriate program from G_M.

Example. The program dependence graph shown in Figure 6 is feasible and corresponds to the program:

```
program
  prod := 1;
  sum := 0;
  x := 1;
  while x < 11 do
    prod := prod * x;
    sum := sum + x;
    x := x + 1
  end;
  mean := sum/10
end(x, sum, prod, mean)
```

Because we are assuming a restricted set of control constructs, each vertex of G_M has at most one incoming control dependence edge (from a predicate vertex or the entry vertex), that is, the control dependences of G_M define a tree 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 the targets of v's outgoing control dependence edges that is consistent with the data dependences of G_M. Once all vertices are ordered, the control dependence subgraph of G_M can be easily converted to an abstract-syntax tree.

Unfortunately, as we have shown in [18], the problem of determining whether it is possible to order a vertex's children is NP-complete. We have explored two

```
function ReconstituteProgram(G_M) returns a program or FAILURE
declare
  G_M, G, G_P: program dependence graphs
  v, w: vertices of G
begin
[1]  G := a copy of G_M
[2]  for each vertex v of G in a post-order traversal of the control-dependence subgraph of G do
[3]     if OrderRegion(G, { w | (v --T_c--> w) ∈ E(G) }) fails then return( FAILURE ) fi
[4]     If v represents an if-predicate then
[5]        If OrderRegion(G, { w | (v --F_c--> w) ∈ E(G) }) fails then return( FAILURE ) fi
[6]     fi
[7]  end
[8]  P := TransformToSyntaxTree(G);
[9]  if G_M = G_P then return( P )
[10] else return( FAILURE )
[11] fi
end
```

Fig. 9. 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.

approaches to dealing with this difficulty:

(1) For graphs created by merging PDGs of actual programs, it is likely that problematic cases rarely arise. We have explored ways of reducing the search space, in the belief that a backtracking method for solving the remaining step can be made to behave satisfactorily. These techniques are described in the remainder of this section.

(2) It is possible to sidestep completely the need to solve an NP-complete problem by performing a limited amount of variable renaming. This technique is described in Section 4.5.3, where it can be used to avoid any difficult ordering step that remains after applying the techniques outlined in approach (1).

The rest of this section describes the function ReconstituteProgram, which is invoked as step five of the program-integration algorithm. ReconstituteProgram is presented in outline form in Figure 9.

ReconstituteProgram alters graph G, which is a copy of G_M; G_M itself is saved, unaltered, for use in the test on line [9]. In the **for**-loop (lines [2]–[7]), the tree induced on G by its control dependences is traversed in postorder. For each vertex v 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. We assume that a function, named TransformToSyntaxTree, has been provided to convert G with ordered vertices into the corresponding abstract-syntax tree.

ReconstituteProgram can fail in two different 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. In this case, TransformToSyntaxTree is used to produce program P from G, P's program dependence graph G_P is built, and G_P is compared to G_M; failure occurs if G_M and G_P are not identical. Examples of these kinds of failure are given in Section 4.5.4.

The correctness of ReconstituteProgram is captured by the following theorem.

THEOREM. *ReconstituteProgram(G_M) succeeds iff graph G_M is feasible.*

It is easy to show that ReconstituteProgram fails when G_M is infeasible: If G_M is infeasible, there is *no* program whose dependence graph is isomorphic to G_M; hence the test in step [9] of ReconstituteProgram (see Figure 9) must fail.

The proof that ReconstituteProgram fails *only* when G_M is infeasible is rather lengthy and is omitted here; the proof can be found in [5].

4.5.1 *Procedure OrderRegion: Ordering Vertices Within a Region.*

Definition. The subgraph induced on a collection of vertices, all of which are targets of control dependence edges from some 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 (however, the entry vertex is a region head).

Given region R, the main job of procedure OrderRegion (shown in Figure 10) is to find a total ordering of the vertices of R that preserves the flow and def-order dependences of G, or to discover that no such ordering is possible.

Note that simply using a topological ordering of the region is not satisfactory. For example, consider the dependence graph fragment shown in Figure 11.

A topological ordering of the vertices of the region subordinate to vertex C is F, D, G, E; however, the dependence graph of the program generated according to this ordering would incorrectly have flow edges from D to G and from D to H, rather than the ones from F to G and from F to H.

A secondary responsibility of OrderRegion is to project onto the head of R information from the vertices of R regarding variable uses, variable definitions, and incoming and outgoing edges. This projection ensures that, when the head of R is considered as a vertex in its enclosing region, it represents all uses and definitions that occur in R.

To order the vertices of R, OrderRegion calls procedures PreserveExposed-UsesAndDefs and PreserveSpans (discussed below). These procedures add edges to R to force an ordering of the vertices consistent with the region's data dependences. (This process is roughly that of introducing anti- and output dependences consistent with the flow and def-order dependences of region R. As explained in Section 6.1, there are fundamental problems in trying to perform integration with a dependence representation that includes anti- and output dependences; thus, OrderRegion must discover these dependences from the merged graph.) If this process introduces a cycle in R, OrderRegion fails; otherwise, a topological sort of region R produces an ordering consistent with the region's data dependences.

Information is projected onto the head of region R both by procedure PreserveExposedUsesAndDefs, which projects the loop-carried flow edges of R and the edges of G with only a single endpoint in R onto the region head, and by procedure ProjectUsesAndDefs, which projects onto the head of R information from the vertices in region R about variable uses and definitions. For example,

Fig. 10. Procedure OrderRegion adds new edges to the given region to ensure that dependences are respected, projects information onto the region head, and topologically sorts the vertices of the region.

```
procedure OrderRegion(G, R)
declare
  G: a graph
  R: a region of G
begin
  PreserveExposedUsesAndDefs(G, R)
  If PreserveSpans(R) fails then fail else TopSort(R) fi
  ProjectUsesAndDefs(G, R)
end
```

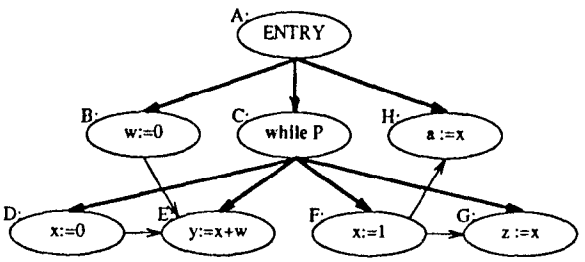


Fig. 11. Dependence graph fragment: Topological ordering F, D, E, G, of the vertices subordinate to vertex C is not acceptable.

procedure ProjectUsesAndDefs would designate vertex C of Figure 11 as representing uses of w and x and definitions of x, y, and z.

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 [1] 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 11 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 11 represents a downwards-exposed definition of variable x.

Procedure PreserveExposedUsesAndDefs uses flow edges of G 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 end-point inside R are also projected onto head(R). This ensures that the region that includes 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

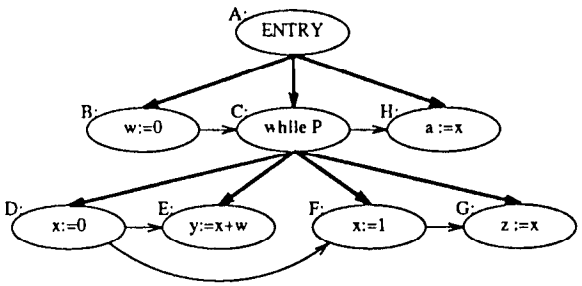


Fig. 12. Dependence graph fragment with new edge D → F added to preserve the downwards-exposed definition of x at vertex F.

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.[5] 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.* Let S stand for $R \cup \{\text{head}(R)\}$. Replace all flow and def-order edges with source outside of S and target inside S with an edge (of the same kind) from the source to head(R). Replace all flow and def-order edges with source inside S and target outside of S with an edge (of the same kind) from head(R) to the target. Consider each loop-carried flow edge $v_1 \rightarrow_{lc(L)} v_2$ such that both v_1 and v_2 are in S. If head(R) = L, then remove the edge; otherwise, replace the edge with a loop-carried flow edge $\text{head}(R) \rightarrow_{lc(L)} \text{head}(R)$.

Figure 12 shows the example dependence graph fragment of Figure 11 after the four steps described above have been performed on the region headed by vertex C.

The edge from D to F was added in Step (3), due to F being downwards-exposed, and this prevents F from preceding D in a topological ordering. The edges from B to C and from C to H were added in Step (4), replacing those from B to E and F to H, respectively.

[5] Our use of the term "downwards-exposed" is slightly nonstandard; we consider a definition to be downwards-exposed in code segment C only if it reaches the end of C and the variable it defines is live at the end of C.

4.5.3 *Dependences Induced by Spans.* To simplify this section's presentation, we begin by considering regions that only include assignment statements; under this restriction, each use of variable x within a region is reached by at most one definition of x that occurs within the region.

In the example dependence graph fragment of Figure 12, the ordering D, F, E, G of the vertices subordinate to vertex C is a topological ordering, but an unacceptable one for our purposes. 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. The dependence graph of the program generated according to this ordering would incorrectly have a flow edge from F to E, rather than the one from D to E. In general, a definition d of variable x must precede all uses it reaches via loop-independent flow edges; other definitions of x must either precede d or follow all the uses reached by d. 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 loop-independent flow targets of d and in the same region as d.

$$\text{Span}(d, x) = \{d\} \cup \{u \mid (d \rightarrow_{li}^{x} u) \in E(\text{EnclosingRegion}(d))\}.$$

Span(d, x) is called an x-span, and vertex d is its *head.*

Restating the observation above in terms of spans, a definition d_1 of variable x must precede all vertices in Span(d_1, x); other definitions of x must either precede d_1 or follow all vertices in Span(d_1, x). Furthermore, for any other x-span with head d_2, if *any* vertex in Span(d_1, x) must precede a vertex in Span(d_2, x), then *all* vertices in Span(d_1, x) must precede d_2.

Unacceptable topological orderings are excluded by considering, for each variable x, all pairs $\langle d_1, d_2 \rangle$ of definitions of x. If there is some vertex v in Span(d_1, x) that must precede some vertex w in Span(d_2, x), because of a path from v to w, then edges are added from all vertices in Span(d_1, x) − Span(d_2, x) to vertex d_2. Similarly, if there is a path from a vertex in Span(d_2, x) to a vertex in Span(d_1, x), edges are added from all vertices in Span(d_2, x) − Span(d_1, x) to vertex d_1. For example, in the graph fragment of Figure 12, the edge E → F would be added because the edge D → F (introduced by PreserveExposedUsesAndDefs) forms a path from Span(D, x) to Span(F, x), and vertex E is in Span(D, x) − Span(F, x).

The reason for taking the set difference Span(d_1, x) − Span(d_2, x) is that, even in regions containing only assignment statements, spans can overlap, as illustrated in Figure 13.

Because C is itself in Span(B, x), adding edges from *all* vertices in Span(B, x) to C would create a self-loop at C, making a topological ordering impossible.

Allowing vertices that represent loops and conditionals introduces the possibility that spans may overlap in two new ways, as illustrated in Figure 14.

In the first case in Figure 14 there must be a def-order dependence edge from d_1 to d_2, or *vice versa*, or the graph would fail the interference test of Section 4.4. In the second case there is a flow edge from d_1 to d_2. These edges force an ordering of the two spans. Thus, allowing conditionals and loops does not complicate PreserveSpans.

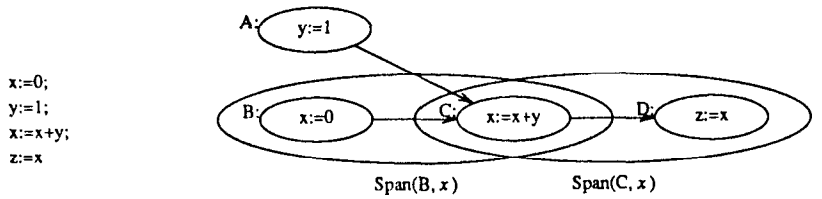


Fig. 13. Straight-line code fragment and corresponding dependence graph fragment (control edges omitted) with overlapping x-spans.

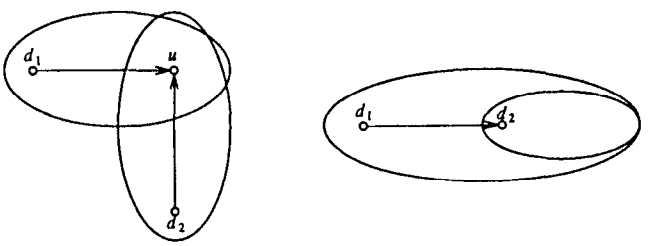


Fig. 14. Conditionals and loops can lead to the two additional kinds of overlapping spans shown above.

There may be pairs of spans, Span(d_1, x) and Span(d_2, x), such that there is no path in either direction between Span(d_1, x) and Span(d_2, x); such pairs are called *independent x-span pairs*. It is still necessary to add edges to force one span to precede the other so as to exclude unacceptable topological orderings. Although it might seem that an arbitrary choice can be made, Figure 15 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 15 includes two x-spans: Span(A, x) and Span(D, x), and two y-spans: Span(B, y) and Span(C, y). There are paths neither between the two x-spans nor between the two y-spans; thus, it appears that one is free to choose to add edges from the vertices of Span(A, x) to vertex D, or from the vertices of Span(D, x) to vertex A, or from the vertices of Span(B, y) to vertex C, or from the vertices of Span(C, y) 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, x) 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, y) to a vertex in Span(C, y), and *vice versa*. Figure 16 shows the fragment of Figure 15 with the new edges added; the path from Span(C, y) to Span(B, y) is shown using dashed lines. The path from Span(B, y) to Span(C, y) is shown using dotted lines.

Unfortunately, as we have shown in [18], the problem of determining the right choice in a situation like the one illustrated in Figure 15 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, procedure PreserveSpans backtracks to the most recent choice point and

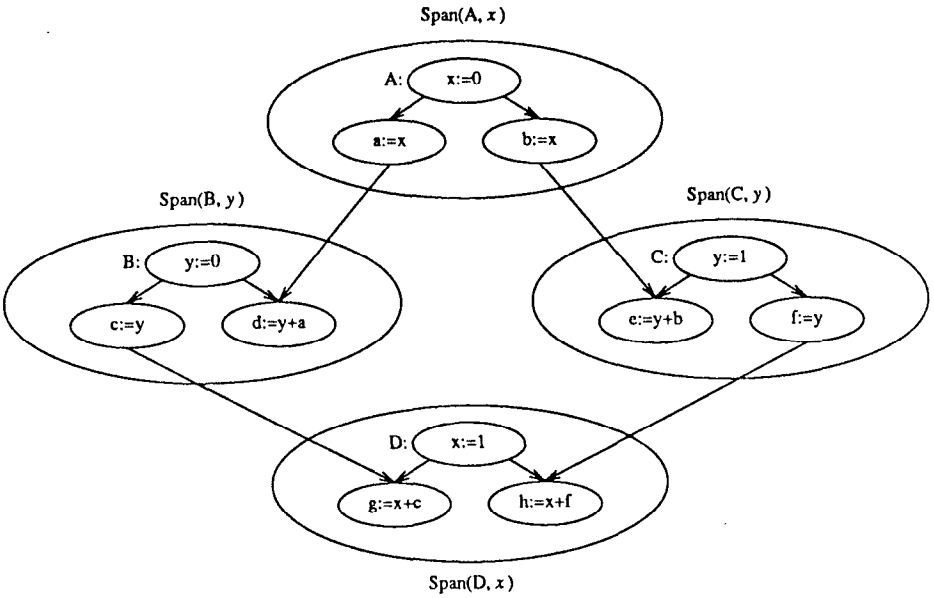


Fig. 15. Graph fragment (control edges omitted) with two x-spans and two y-spans.

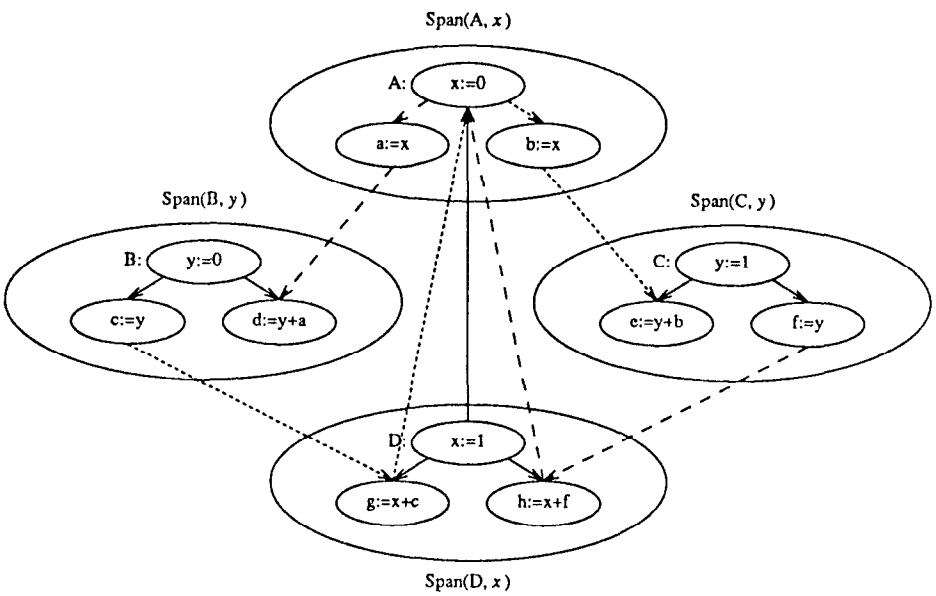


Fig. 16. Span(D, x) has been chosen to precede Span(A, x). Paths have been created from Span(B, y) to Span(C, y) and *vice versa*. The path from Span(C, y) to Span(B, y) is indicated using dashed edges; the path from Span(B, y) to Span(C, y) is indicated using dotted edges.

tries a different choice. If all choices lead to the introduction of a cycle, the graph is infeasible. Procedure PreserveSpans is presented in Figure 17.

PreserveSpans makes use of an auxiliary procedure, OrderDependentSpans, to order any span pairs of region R whose relative order is forced by a connecting

```
procedure PreserveSpans(R)
declare
   R: a region
   h1, h2: vertices of R
   Stack: a stack
begin
   TransitivelyClose(R)
   If R is cyclic then fail fi
   Unmark all edges of R
   OrderDependentSpans(R)
   Stack := EmptyStack()
   do
      R is acyclic and there exist independent x-span pairs (for some variable x) with heads h1 and h2 →
         Push(Stack, R, h1, h2)
         AddEdgeAndClose(R, (h1, h2))
         OrderDependentSpans(R)
    [] R is cyclic and Empty(Stack) → fail
    [] R is cyclic and ¬Empty(Stack) →
         R, h1, h2 := Pop(Stack)
         AddEdgeAndClose(R, (h2, h1))
         OrderDependentSpans(R)
   od
end

procedure OrderDependentSpans(R)
declare
   R: a region
   a, b, c, u, v, w: vertices of R
   A, B: sets of vertices
   x: a variable
begin
   while there exists an unmarked edge (v, w) in R do
[1]   Mark edge (v, w)
[2]   for each variable x ∈ (Defs(v) ∪ Uses(v)) ∩ (Defs(w) ∪ Uses(w)) do
         /* v is in an x-span and w is in an x-span */
         A := { u | v ∈ Span(u, x) } /* heads of x-spans of which v is a member */
         B := { u | w ∈ Span(u, x) } /* heads of x-spans of which w is a member */
[3]      for each vertex a ∈ A do
[4]         for each vertex b ∈ B do
[5]            for each c ∈ (Span(a, x) - Span(b, x)) do
                  if (c, b) ∉ E(R) then AddEdgeAndClose(R, (c, b)) fi
               end
            end
         end
      end
   end
end
```

Fig. 17. Procedure PreserveSpans introduces edges into region R to preserve the spans of R.

path. An invariant of the two procedures, established in the first line of PreserveSpans, is that graph R is transitively closed. The basic operation used in PreserveSpans and OrderDependentSpans is "AddEdgeAndClose(R, (a, b))", whose first argument is a graph and whose second argument is an edge to be added to the graph. AddEdgeAndClose(R, (a, b)) carries out two actions:

(1) edge (a, b) is inserted into R;

(2) any additional edges needed to transitively close R are inserted into R.

Because R is transitively closed, paths that force span orderings correspond to edges of R; furthermore, the cost of AddEdgeAndClose is quadratic (rather than cubic) in the number of vertices of R.

Each edge of R can be *marked* or *unmarked*; the edges added to R by AddEdgeAndClose (by either 1 or 2) are unmarked. Edges are marked at line [1]

in OrderDependentSpans. An invariant of the **while**-loop in OrderDependentSpans is that, for each marked edge e, all spans for which e forces an ordering are appropriately ordered. Thus, after an unmarked edge (v, w) is selected (and marked), the invariant is reestablished as follows: line [2] generates all variables x for which both v and w are elements of an x-span (but not necessarily the same x-span); lines [3] and [4] iterate over all pairs of x-spans (represented by their heads), such that v is a member of the first span and w is a member of the second; line [5] orders the two spans as forced by the presence of edge (v, w).

The initial call on OrderDependentSpans in PreserveSpans serves to introduce edges for all forced span orderings. The **do-od**-loop then implements a backtracking algorithm that examines all choices for independent span pairs. Each pair of independent spans (represented by their span heads, say h_1 and h_2) represents two possibilities—the elements of Span(h_1, x) could precede the elements of Span(h_2, x), or *vice versa*. The first possibility is represented by the call AddEdgeAndClose(R, (h_1, h_2)), which introduces an edge directed from h_1 to h_2; the second possibility (which is tried only in the backtracking step, guarded by the condition "R is cyclic and ¬Empty(*Stack*)") is represented by the call AddEdgeAndClose(R, (h_2, h_1)). In both cases, OrderDependentSpans is called to introduce edges for all span orderings forced as a consequence of the new edge. (A single edge, such as (h_1, h_2), may force an ordering between spans other than those headed by h_1 and h_2.)

The information needed for backtracking is kept as a stack of triples: the graph R as it existed before a given "choice," span head h_1, and span head h_2. Backtracking terminates with failure if R is cyclic and the stack is empty, because no alternative remains to be tried. When R is cyclic but the stack is not empty, one entry is popped from the stack and the "choice" is tried in the opposite direction. (Since there are only two choices to be tried for each pair of span heads, there is no Push before continuing the search with the second alternative.) PreserveSpans terminates with success if R is acyclic and there remain no independent x-span pairs.

The cost of OrderDependentSpans can be expressed in terms of the following parameters:

N	the maximum number of vertices in a region,
V	the number of variables in the program,
G	the maximum number of spans of which any vertex is a member,
S	the maximum size of a span.

Our statement of the complexity of OrderDependentSpans is based on the assumption that the set operations Insert, Delete, and MemberOf have unit cost, and that Union, Intersection, and Difference can be performed with linear cost. At most N^2 edges can be inserted in R; for each edge, the processing cost is N^2: the cost of reclosing R, plus the product of the costs of lines [2], [3], [4], and [5], which are $O(V)$, $O(G)$, $O(G)$, and $O(S)$, respectively. Thus, the cost of OrderDependentSpans is bounded by $O(N^2 \cdot (N^2 + V \cdot G^2 \cdot S))$.

PreserveSpans performs at least one call on OrderDependentSpans; if backtracking is needed, there can be an additional factor of 2^P, where P is the number

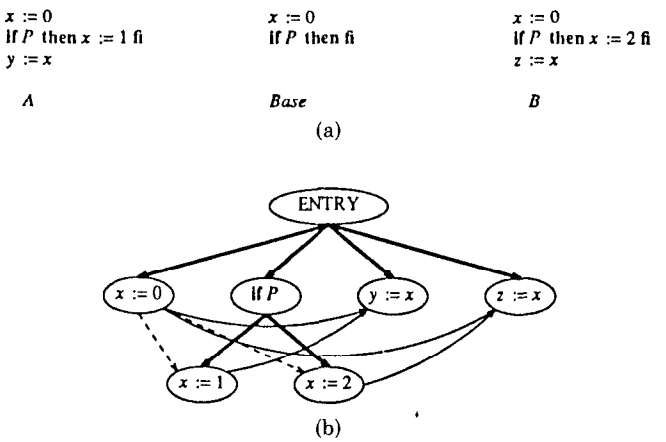


Fig. 18. Illustration of interference due to failure in OrderRegion. Fragments of a base program and two variants, and the infeasible merged program dependence graph. The vertices of G_M cannot be ordered so as to preserve both the flow edge from "$x := 1$" to "$y := x$", and the flow edge from "$x := 2$" to "$z := x$".

of pairs of independent spans that remain after the initial call on OrderDependentSpans.

It is possible to sidestep entirely the need for backtracking in PreserveSpans by allowing a limited amount of variable renaming to be performed. In particular, when two x-spans, s_1 and s_2, are independent, all occurrences of the name x in s_1 (as well as in any x-spans that overlap s_1 in the region) 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, and thus conflicts with our goal that the integrated program be composed of exactly the statements and control structures that appear as components of the base program and its variants. Further work is needed to determine whether this technique will be necessary in practice.

4.5.4 *Examples of Interference Due to Infeasibility.* In this section, we illustrate the two ways in which ReconstituteProgram can fail. Failure can occur in procedure OrderRegion because there is no acceptable ordering for the children of some vertex of the merged program dependence graph G_M. This kind of infeasibility is illustrated in Figure 18.

An attempt to integrate any programs *Base*, *A*, and *B* that include the program fragments shown in Figure 18(a) would produce a merged PDG that includes the subgraph shown in Figure 18(b). OrderRegion would fail because the children of the vertex "**if** P" cannot be ordered so as to preserve both the flow edge from "$x := 1$" to "$y := x$" and the flow edge from "$x := 2$" to "$z := x$."

Failure can also occur in ReconstituteProgram after acceptable orderings are found for the children of every vertex in G_M. After all calls to OrderRegion

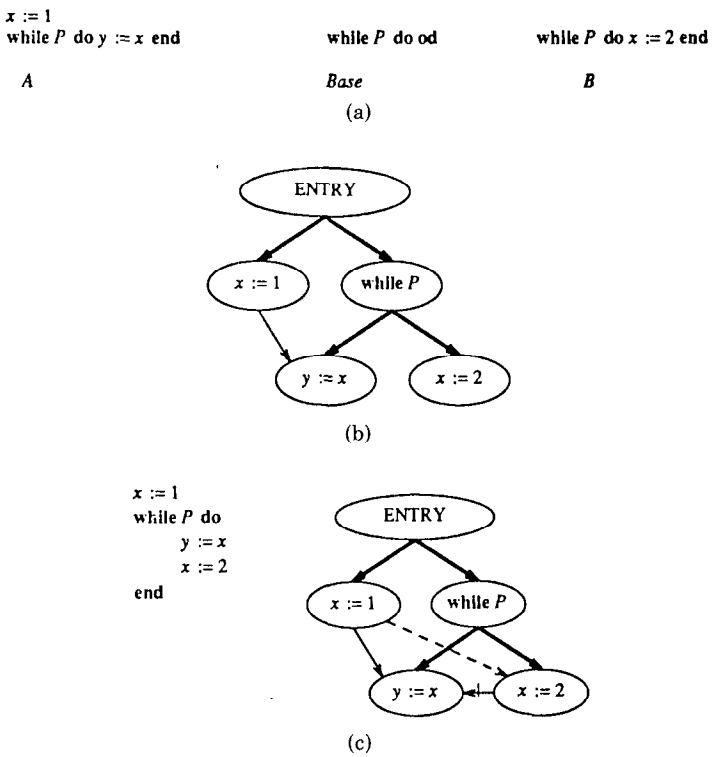


Fig. 19. Illustration of interference discovered in the final step of ReconstituteProgram. The merged dependence graph G_M, shown in (b), is not identical to the dependence graph of program Q, shown in (c), which is the program generated from G_M by ReconstituteProgram.

succeed, TransformToSyntaxTree is used to produce a program P, P's program dependence graph G_P is built, and G_P is compared to G_M; failure occurs if G_M and G_P are not identical. This kind of infeasibility is illustrated in Figure 19.

Again, an attempt to integrate any programs *Base*, *A*, and *B* that include the program fragments shown in Figure 19(a) would produce a merged PDG that includes the subgraph shown in Figure 19(b). OrderRegion would succeed, and a program P that includes the program fragment shown in Figure 19(c) would be produced. P's program dependence graph would include the subgraph shown in Figure 19(c), which is not identical to the subgraph shown in Figure 19(b); thus ReconstituteProgram would fail.

4.6 Recap of the Program Integration Algorithm

The function Integrate, given in Figure 20, 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, function Integrate produces a program P that integrates *A* and *B*.

```
function Integrate(A, B, Base) returns a program or FAILURE
declare
  Base, A, B, M: programs
  G_Base, G_A, G_B, G_M: program dependence graphs
begin
  G_M := (G_A / AP_{A,Base}) ∪ (G_B / AP_{B,Base}) ∪ (G_Base / PP_{Base,A,B})
  if (G_M / AP_{A,Base} ≠ G_A / AP_{A,Base}) ∨ (G_M / AP_{B,Base} ≠ G_B / AP_{B,Base}) then return( FAILURE ) fi
  M := ReconstituteProgram(G_M)
  if M = FAILURE then return( FAILURE ) fi
  return(M)
end
```

Fig. 20. 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, function Integrate produces a program P that integrates A and B.

The following theorem characterizes the execution behavior of the integrated program produced by function Integrate in terms of the behaviors of the base program and the two variants [27, 28].

THEOREM (Integration Theorem [27, 28]). *If A and B are two variants of Base for which integration succeeds (and produces program M), then for any initial state σ on which A, B, and Base all halt, (1) M halts on σ; (2) if x is a variable defined in the final state of A for which the final states of A and Base disagree, then the final state of M agrees with the final state of A on x; (3) if y is a variable defined in the final state of B for which the final states of B and Base disagree, then the final state of M agrees with the final state of B on y; and (4) if z is a variable on which the final states of A, B, and Base agree, then the final state of M agrees with the final state of Base on z.*

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

The cost of algorithm Integrate breaks down into three components: (1) building the program dependence graphs for *Base*, A, and B; (2) building the merged program dependence graph G_M and determining whether the changed behaviors of A and B are preserved in G_M; and (3) reconstituting a program from G_M.

(1) Building a program dependence graph is dominated by the cost of computing reaching definitions; for the limited language considered here, this has cost O((# program components) · (# of assignment statements)).

(2) Function AffectedPoints (Figure 5) is linear in the size of its arguments; slicing a graph is linear in the size of the slice. Consequently, the cost of creating the merged graph G_M is linear in the sum of the sizes of G_{Base}, G_A, and G_B. Similarly, the cost of testing for interference by the test described in Section 4.4 is linear in the sum of the sizes of G_A, G_B, and G_M.

(3) The cost of ReconstituteProgram is dominated by the cost of the calls on PreserveSpans made by OrderRegion. If no backtracking is needed, the cost of ReconstituteProgram is $O(R \cdot N^2 \cdot (N^2 + V \cdot G^2 \cdot S))$, where R is the number of regions in the program, and other quantities are as described in

Section 4.5.3; backtracking can contribute an additional exponential factor for each region.

5. APPLICATIONS TO PROGRAMMING IN THE LARGE

An environment for 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 these issues. In many respects, program integration is the key operation in an environment to support programming in the large. Three specific applications for program integration are discussed below.

5.1 Propagating Changes Through Related Versions

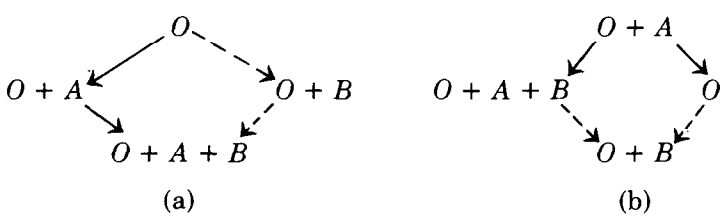
The program-integration problem arises when a family of related versions of a program has been created (for example, to support different machines or different operating systems), and the goal is to make the same change (e.g., an enhancement or a bug-fix) to all of them. Our program-integration algorithm provides a way for changes made to the base version to be automatically installed in the other versions.

For example, consider the diagram shown in Figure 21, where Figure 21(a) represents the original development tree for some module (branches are numbered as in RCS [29]).

In Figure 21(b), the variant numbered "1.1.2.1" represents the enhanced version of the base program "1.1" (created by editing a copy of base program "1.1"). Variant "1.1.2.2," which is obtained by integrating "1.1.2.1" and "1.2" with respect to "1.1," represents the result of propagating the enhancement to "1.2." Figure 21(c) represents the new development history after all integrations have been performed and the enhancement has been propagated to all versions.

5.2 Separating Consecutive Program Modifications

Another application of program integration permits separating 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 situation (a) in the following diagram:



Under certain circumstances, the development-history tree can be rerooted 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

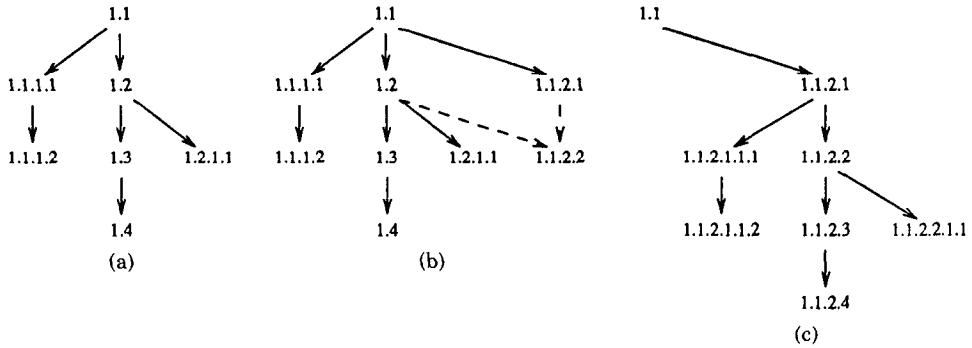


Fig. 21. Propagating changes through a development-history tree.

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 statement s that occurs in $O + A$ but not in O is a "new" statement arising from an insertion; when $O + A$ is the base program, we treat the missing s in O as if a user had deleted s from $O + A$ to create O. Version $O + A + B$ is still treated as being a program version derived from $O + A$. $O + B$ is created by integrating O and $O + A + B$ with respect to base program $O + A$.

5.3 Optimistic Concurrency Control

An environment for programming in the large must provide concurrency control; that is, it must resolve simultaneous requests for access to a program. Traditional database approaches to concurrency control assume that transactions are very short-lived, and so avoid conflict using locking mechanisms. This solution is not acceptable in programming environments where transactions may require hours, days, or weeks.

An alternative to locking is the use of an *optimistic concurrency control* strategy: grant all access requests and resolve conflicts when the transactions complete. The success of an optimistic concurrency control strategy clearly depends on the existence of an automatic program-integration algorithm to provide acceptable conflict resolution.

6. RELATION TO PREVIOUS WORK

We are not aware of any other work that permits the integration of program variants so as to preserve changes to a base program's behavior. One piece of work that addresses a related, but different, problem is [7]; however, it treats the integration of program *extensions*, not program *modifications*:

> A program extension extends the domain of a partial function without altering any of the initially defined values, while a modification redefines values that were defined initially [7].

In [7], functions A and B are merged without regard to *Base*. The function that results from the merge preserves the (entire) behavior of *both*; thus, A and

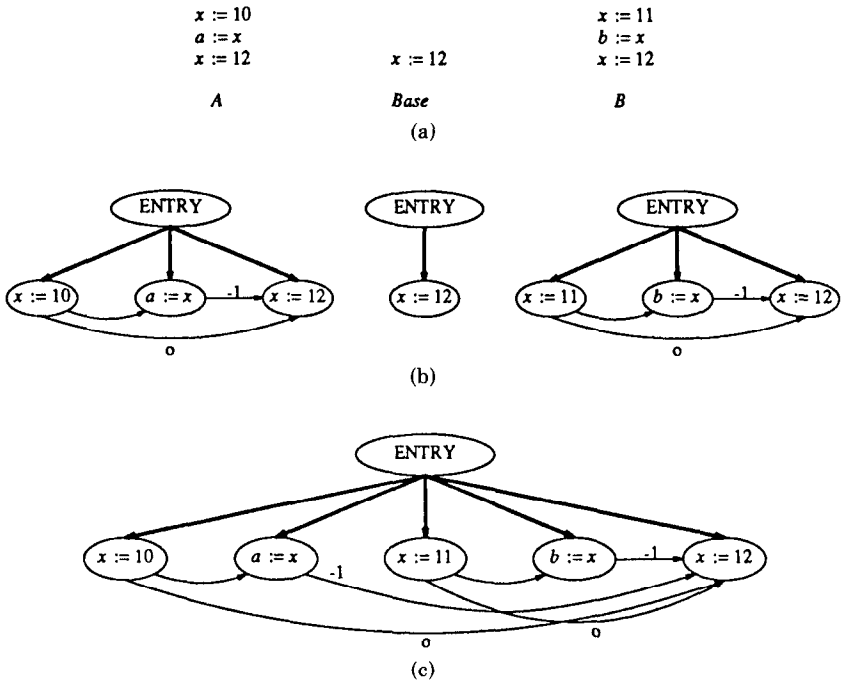


Fig. 22. A base program and two variants, the program dependence graphs that would be built for the three programs if program dependence graphs were to include anti- and output dependence edges, and the merged graph. Control dependence edges are shown in boldface; flow dependence edges are shown using (unlabeled) arrows; output dependence edges are shown using arrows labeled "o"; antidependence edges are shown using arrows labeled "−1".

B cannot be merged if they conflict at any point where both are defined. In contrast, this paper addresses the integration of *modifications* (in the sense defined in [7], quoted above). With our technique, a program that results from merging *A* and *B* preserves the *changed behavior* of *A* with respect to *Base*, the *changed behavior* of *B* with respect to *Base*, and the unchanged behavior common to all three.

In the rest of this section, we discuss some technical differences between the program dependence graphs and operations on them that are used in this paper and those used by others.

6.1 Program Dependence Graphs

There are several reasons for our use of program dependence graphs that include def-order dependence edges but omit anti- and output dependence edges. The basic problem is that, for the purposes of program integration, anti- and output dependences impose unnecessary ordering constraints. Two consequences of this problem are illustrated in Figures 22 and 23.

Figure 22 shows a base program and two variants, the program dependence graphs that would be built for the three programs if program dependence graphs

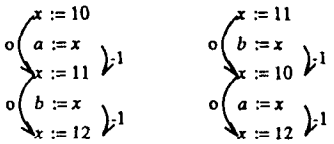


Fig. 23. Two strongly equivalent programs with different sets of anti- and output dependences (antidependences are shown to the right of the program using arrows labeled "−1"; output dependences are shown to the left of the program using arrows labeled "o"). The programs have the same (empty) sets of def-order dependences, and the same sets of flow dependences.

were to include anti- and output dependence edges, and the merged graph that combines the changed computations of the variants with the computation common to all three programs. The merged graph is infeasible; it is not possible to order the assignments to x so as to preserve the merged graph's anti- and output dependences. In contrast, if anti- and output dependences are omitted from the program dependence graphs of this example, the merged graph *is* feasible and corresponds to both of the programs shown in Figure 23 (ignore the anti- and output dependence annotations).

Figure 23 illustrates a second advantage of using def-order dependences rather than anti- and output dependences; using def-order dependences allows a larger class of equivalent programs to have the same program dependence graph. Figure 23 shows two strongly equivalent programs that have different sets of anti- and output dependences (and thus would have different program dependence graphs if such graphs included anti- and/or output dependences). The programs have the same (empty) sets of def-order dependences and the same sets of flow dependences; thus, they have the same program dependence graphs, using the definition from this paper.

6.2 Operations on Program Dependence Graphs

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 [24].

Our work requires a solution to a more general problem because the final program dependence graph is created by merging three 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 [18], 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 [10]. Programs written in the language they consider have a close correspondence to the subgraph of control dependences 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 (feasible) PDG. They

assert that the transformations of this class preserve the property that the resulting graph is feasible. Thus, while their results are relevant to generalizing ReconstituteProgram to work on PDGs generated from programs with arbitrary control flow [11], they will have to be extended to account for the possibility of infeasibility.

7. EXTENSIONS AND FUTURE WORK

In this paper, the problem of program integration is studied in an extremely simplified setting. For this reason, the algorithm described in the paper is not yet applicable to real programming languages; however, we feel that the approach that we have developed provides a strong foundation for creating a system that supports program integration. In this section, we describe some of the issues we have addressed in extending our work and outline some problems for future research.

7.1 Applicability to Realistic Languages

Among the obvious deficiencies of the present study are the absence of numerous programming constructs and data types found in languages used for writing "real" programs. Certainly, one area for further work is to extend the integration method to handle additional programming language constructs, such as declarations, **break** statements, and I/O statements, as well as other data types, such as records and arrays.

The major challenge when extending the integration method to handle other programming language constructs is devising a suitable extension of the program dependence representation. For example, the simplest way of handling arrays is to treat an update to any cell as a conditional update to the entire array. However, this strategy would preclude the integration of some noninterfering variants. Analyses of array index expressions developed for vectorizing compilers provide sharper information about the actual dependences among array references [2, 3, 6, 31]. Because the definition of program dependence graphs that we use for program integration differs from that used in previous work, previous results in this area will require adaptation.

We have recently made progress towards handling languages with procedure calls and pointer-valued variables. Our results in these areas are summarized below.

7.1.1 *Interprocedural Slicing Using Dependence Graphs.* As a first step toward extending our integration algorithm to handle languages with procedures, we have devised a multiprocedure dependence representation and developed a new algorithm for interprocedural slicing that uses this representation [19]. The algorithm generates a slice of an entire system, where the slice may cross the boundaries of procedure calls. It is both simpler and more precise than the one previous algorithm given for interprocedural slicing [30].

The method described in [30] does not generate a precise slice because it fails to account for the calling context of a called procedure. The imprecision of the

method can be illustrated using the following example:

```
program Main
  sum := 0;
  x := 1;
  while x < 11 do
    call Add(sum, x);
    call Add(x, 1)
  end
end(x, sum)

procedure Add(a, b)
  a := a + b
return
```

Using the algorithm from [30] to slice this system with respect to variable x at the end of program *Main*, we obtain everything except the final use of *sum* at the end of program *Main*:

```
program Main
  sum := 0;
  x := 1;
  while x < 11 do
    call Add(sum, x);
    call Add(x, 1)
  end
end(x)

procedure Add(a, b)
  a := a + b
return
```

However, further inspection shows that the value of x at the end of program *Main* is not affected by the first call on *Add* in *Main*, nor by the initialization of *sum* in *Main*. The reason these components are included in the slice is (roughly) the following: the statement "**call** *Add*(x, 1)" in program *Main* causes the slice to "descend" into procedure *Add*. When the slice reaches the beginning of *Add*, it "ascends" to *all* sites that call *Add*, both the site in *Main* at which it "descended" as well as the (irrelevant) site "**call** *Add*(*sum*, x)."

In contrast, our algorithm for interprocedural slicing correctly accounts for the calling context of a called procedure; in the example give above, the first call on *Add* in *Main* and the initialization of *sum* in *Main* are both correctly left out of the slice:

```
program Main
  x := 1;
  while x < 11 do
    call Add(x, 1)
  end
end(x)

procedure Add(a, b)
  a := a + b
return
```

A key element of this algorithm is an auxiliary structure that represents calling and parameter-linkage relationships. This structure, called the *linkage grammar*, takes the form of an attribute grammar. Transitive dependences due to procedure calls are determined using a standard attribute-grammar construction: the computation of the nonterminals' *subordinate characteristic graphs*. These dependences are the key to the slicing algorithm; they permit the algorithm to "come back up" from a procedure call (e.g., from procedure *Add* in the above example) without first descending to slice the procedure (it is placed on a queue of

procedures to be sliced later). This strategy prevents the algorithm from ever ascending to an irrelevant call site [19].

7.1.2 *Dependence Analysis for Pointer Variables.* To incorporate pointer-valued variables, an analysis of pointer usage is necessary; without the information that such an analysis provides, an update via a dereferenced pointer has to be considered a potential update to every location in memory.

We have devised a method for determining data dependences between program statements for programming languages that have pointer-valued variables (e.g., Lisp and Pascal). The method determines data dependences that reflect the usage of heap-allocated storage in such languages, which permits us to build (and slice) program dependence graphs for programs written in such languages. The method accounts for destructive updates to fields of a structure, and thus is *not* limited to simple cases where all structures are trees or acyclic graphs; the method is applicable to programs that build up structures that contain cycles.

Unlike the situation that exists for programs with (only) scalar variables—where there is a fixed "layout" of memory—for programs that manipulate heap-allocated storage, not all accessible memory locations are named by program variables. In the latter situation, new memory locations are allocated dynamically in the form of cells taken from the heap. To compute data dependences between constructs that manipulate and access heap-allocated storage, our starting point is the method described by Jones and Muchnick in [20], which, for each program point q, determines a set of structures that approximates the different "layouts" of memory that can possibly arise at q during execution. We extend the domain employed in the Jones–Muchnick abstract interpretation so that the (abstract) memory locations are labeled by the program points that set their contents. Flow dependences are then determined from these memory layouts according to the component labels found along the access paths that must be traversed to evaluate the program's statements and predicates during execution.

7.2 An Interactive Integration Tool

It remains to be seen how often integrations of real changes to programs of substantial size can be automatically accommodated by our integration technique. Due to fundamental limitations on determining information about programs via data-flow analysis and on testing equivalence of programs, both the procedure for identifying changed computations and the test for interference must be *safe* rather than *exact*. Consequently, the integration algorithm will report interference in some cases where no real conflict exists. Whether or not fully automatic integration is a realistic proposition can be determined only through experience; an integration tool must be built and used on real programs.

A successful integration tool will certainly have to provide facilities for programmers to cope with reported interference—facilities that would enable diagnosing spurious interference of the kind described above, as well as aids for resolving true conflicts. For these situations, it is not enough merely to detect and report interference; one needs a tool for *semiautomatic, interactive integration* so that the user can guide the integration process to a successful completion. Some rudimentary diagnostic facilities have been incorporated in a prototype

program-integration tool embedded in an editor created using the Synthesizer Generator [25, 26]. The tool's integration command invokes the integration algorithm on a base program and two variants, and reports whether the variant programs interfere. If interference is reported, it is possible for the user to examine sites of potential conflicts—sites which may or may not represent actual conflicts. (Roughly speaking, the sites reported are those at which slices of the two variants become "intertwined" in the merged graph.) The tool's slice command makes it possible for the user to display the elements of program slices; slicing can be invoked to provide further information about potential integration conflicts.

Further work on this tool is needed to provide capabilities for the user to resolve conflicts and create a satisfactory merged program. Renaming program variables and suppressing dependences between program components would be two ways a user might interact with an interactive integration tool. Conflict-resolution facilities could operate directly on the merged program dependence graph, which is built by the integration algorithm whether or not the variants interfere.

7.3 Alternative Program-Integration Criteria

We anticipate that it will be useful to define variations on the technique presented in this paper. It will undoubtably be desirable for users to be able to supply pragmas to furnish additional information to the program-integration system. For example, a user-supplied assertion that a change to a certain module in one variant does not affect its functionality (only its efficiency, for example) could be used to limit the scope of slicing and interference testing.

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.

Finally, there may be cases where it is desirable for programs produced through integration to have somewhat different semantic properties than those guaranteed by the algorithm given above. For example, consider the integration of programs that contain I/O statements. I/O statements could be treated as accesses to two special objects *input* and *output*, which may be thought of as streams that are updated whenever operations are performed on them. For example, an output statement "**write** x" could 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 dependence edges [24]. Unfortunately, the integration of a base program with two variants that both affect the output stream would fail due to interference. Thus, it may be useful to develop an alternative representation for output statements in dependence graphs that would allow the creation of an integrated program that would not necessarily preserve the output stream of either variant, but instead produce an interleaving of their output streams. In

cases where interleaved output is an appropriate property, this might make it possible to perform integrations that would otherwise fail.

ACKNOWLEDGMENTS

We would like to thank Dexter Kozen for his comments on an earlier version of the paper, the referees for their many helpful suggestions, and Thomas Bricker for his role in developing the prototype program-integration system.

REFERENCES

1. AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. ALLEN, J. R., AND KENNEDY, K. PFC: A program to convert Fortran to parallel form. Tech. Rep. MASC TR82-6, Dept. of Math. Sciences, Rice Univ., Houston, Tex., March 1982.
3. ALLEN, J. R. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. dissertation, Dept. of Math. Sciences, Rice Univ., Houston, Tex., April 1983.
4. ALLEN, J. R., AND KENNEDY, K. Automatic loop interchange. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction* (Montreal, June 20–22, 1984). *ACM SIGPLAN Not. 19,* 6 (June 1984), 233–246.
5. BALL, T., HORWITZ, S., AND REPS, T. Correctness of an algorithm for reconstituting a program from a dependence graph. Computer Sciences Dept., Univ. of Wisconsin, Madison. Tech. Rep. in preparation, Spring 1989.
6. BANNERJEE, U. Speedup of ordinary programs. Ph.D. dissertation and Tech. Rep. R-79-989, Dept. of Computer Science, Univ. of Illinois, Urbana, Oct. 1979.
7. BERZINS, V. On merging software extensions. *Acta Inf. 23* (1986), 607–619.
8. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood, Cliffs, N.J., 1976.
9. DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structured editors: The MENTOR experience. In *Interactive Programming Environments,* D. Barstow, E. Sandewall, and H. Shrobe, Eds., McGraw-Hill, New York, 1984, 128–140.
10. 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). ACM, New York, 1985, 179–189.
11. FERRANTE, J., OTTENSTEIN, K., AND WARREN, J. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3 (July 1987), 319–349.
12. HABERMANN, A. N., AND NOTKIN, D. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. SE-12,* 12 (Dec. 1986), 1117–1127.
13. HOARE, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12,* 10 (Oct. 1969), 576–580, 583.
14. HORWITZ, S., PRINS, J., AND REPS, T. Integrating non-interfering versions of programs. TR-690, Computer Sciences Dept., Univ. of Wisconsin, Madison, March 1987.
15. HORWITZ, S., PFEIFFER, P., AND REPS, T. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation* (Portland, Ore., June 21–23, 1989). ACM, New York, 1989.
16. HORWITZ, S., PRINS, J., AND REPS, T. Integrating non-interfering versions of programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15, 1988), ACM, New York, 1988, 133–145.
17. HORWITZ, S., PRINS, J., AND REPS, T. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan. 13–15, 1988). ACM, New York, 1988, 146–157.
18. HORWITZ, S., PRINS, J., AND REPS, T. On the suitability of dependence graphs for representing programs. Computer Sciences Dept., Univ. of Wisconsin, Madison, Aug. 1988. Submitted for publication.

19. HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation* (Atlanta, Ga., June 22–24, 1988). *ACM SIGPLAN Not. 23*, 7 (July 1988), 35–46.
20. JONES, N. D., AND MUCHNICK, S. S. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds., Prentice-Hall, Englewood Cliffs, N.J., 1981.
21. 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. Comput. C-21*, 12 (Dec. 1972), 1293–1310.
22. 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), ACM, New York, 1981, 207–218.
23. NOTKIN, D., ELLISON, R. J., STAUDT, B. J., KAISER, G. E., KANT, E., HABERMANN, A. N., AMBRIOLA, V., AND MONTANGERO, C. Special issue on the GANDALF project. *J. Syst. Softw. 5*, 2 (May 1985).
24. OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. 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, Pa., Apr. 23–25, 1984). *ACM SIGPLAN Not. 19*, 5 (May 1984), 177–184.
25. REPS, T., AND TEITELBAUM, T. The Synthesizer Generator. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., April 23–25, 1984). *ACM SIGPLAN Not. 19*, 5 (May 1984), 42–48.
26. REPS, T., AND TEITELBAUM, T. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer, New York, 1988.
27. REPS, T., AND YANG, W. The semantics of program slicing. TR-777, Computer Sciences Dept., Univ. of Wisconsin, Madison, June 1988.
28. REPS, T., AND YANG, W. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages* (Barcelona, March 13–17, 1989). *Lecture Notes in Computer Science, 352.* Springer, New York, 1989, 360–374.
29. TICHY, W. F. RCS: A system for version control. *Softw. Pract. Exper. 15*, 7 (July 1985), 637–654.
30. WEISER, M. Program slicing. *IEEE Trans. Softw. Eng. SE-10*, 4 (July 1984), 352–357.
31. WOLFE, M. J. Optimizing supercompilers for supercomputers. Ph.D. dissertation and Tech. Rep. R-82-1105, Dept. of Computer Science, Univ. of Illinois, Urbana, Oct. 1982.

Received April 1987; revised January 1989; accepted February 1989