Semantics-Based Program Integration

THOMAS REPS and SUSAN HORWITZ
University of Wisconsin – Madison

The need to integrate several versions of a base 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.

Our recent work addresses the problem of building a *semantics-based* tool for program integration; this paper describes the techniques we have developed, which provide the foundation for creating such a tool. Semantics-based integration is based on the assumption that a difference in the *behavior* of one of the variant programs from that of the base program, rather than a difference in the *text*, is significant and must be preserved in the merged program. Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with the base program. To determine this information, the integration algorithm employs a program representation that is similar (although not identical) to the *program dependence graphs* that have been used previously in vectorizing 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.

1. INTRODUCTION

Our concern is the development of a semantics-based tool for *integrating program variants*. The goal is to create a system that will either automatically combine several different but related variants of a base program, or else determine that the variants incorporate interfering changes.

The need to integrate several versions of a program into a common one arises frequently, but it is a tedious and time consuming task to integrate programs by hand. Anyone who has had to reconcile divergent lines of development will identify with the need for automatic assistance. Unfortunately, at present, the only available tools for integration are variants of differential file comparators, such as the Unix utility *diff*.

Portions of this paper are excerpted from [10] and [12].

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

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

The problem with current tools is that they implement an operation for merging files as strings of text. This approach has the advantage that such tools are as applicable to merging documents, data files, and other text objects as they are to merging programs. However, these tools are necessarily of limited utility for integrating programs because the manner in which two programs are merged is not *safe*. One has no guarantees about the way the program that results from a purely *textual* merge behaves in relation to the behaviors of the three 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.

Recently, in collaboration with J. Prins, we developed a radically different approach to integrating programs [10]. Our method is based on the assumption that any change in the *behavior* of one of the variant programs from that of the base program, rather than a difference in the *text*, is significant and must be preserved in the merged program. Although it is undecidable to determine whether a program modification actually leads to such a difference, it is possible to determine a safe approximation by comparing each of the variants with the base program. To determine this information, the integration algorithm employs a program representation that is similar, although not identical, to the *program dependence graphs* that have been used previously in vectorizing compilers. The algorithm also make use of the notion of a *program slice* to find just those statements of a program that determine the values of potentially affected variables.

Our integration algorithm 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 certain well-defined sense, the algorithm produces a program M that integrates A and B. A number of uses for this operation in a system to support programming in the large are described in Section 2. The integration algorithm itself is described and illustrated in Section 3 (it is described in full detail in [10]).

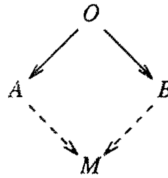
In order to study the integration problem in a simplified setting, we have initially restricted our attention to a programming language with only the most basic data types and control constructs; in particular, we assume that expressions contain only scalar variables and constants, and that the only statements used in programs are assignment statements, conditional statements, and while-loops. Current work, described in Section 4, concerns how to extend the set of language constructs to which the program-integration algorithm is applicable, as well as the construction of a prototype integration facility.

2. 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, the operation of integrating program variants is the key operation in an environment to support programming in the large.

One situation in which the integration operation would be employed is when a base version of a system is enhanced along different lines, either by users or maintainers, thereby creating a number of related versions with slightly different features. To create a new version that incorporates several of the enhancements simultaneously, the integration operation is applied to the base version and its variants. If no conflicts are found, the versions are merged to create an integrated version that combines their separate features.

Pictorially, we can represent this situation by the following diagram, where O represents the original program, A and B represent the enhanced versions, and M represents the integrated program:



Besides the basic program integration scenario described above, there are a number of other applications for program integration; three such applications are discussed below.

2.1. 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. One would like to be able to change the base version of the system and to have the change propagated automatically to all other versions. Of course, the change cannot be blindly applied to all versions since the differences among the versions might alter the effects of the change; one has to check for conflicts in the implementations of the different versions. 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 1, where Figure 1(a) represents the original development tree for some module (branches are numbered as in RCS [20]). In Figure 1(b), the variant numbered "1.1.2.1" represents the enhanced version of the base program "1.1." Variant "1.1.2.2," which is obtained by integrating "1.1," "1.1.2.1," and "1.2," represents the result of propagating the enhancement to "1.2." Figure 1(c) represents the new development history after all integrations have been performed and the enhancement has been propagated to all versions.

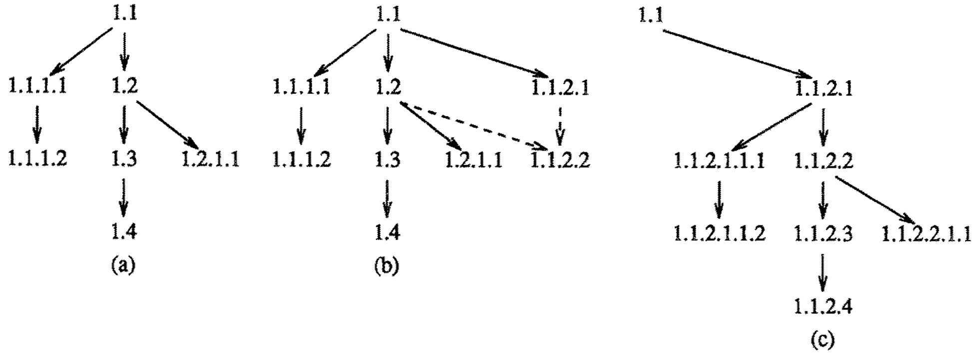
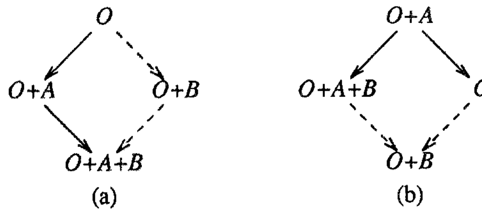


Figure 1. Propagating changes through a development-history tree.

2.2. 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:



By re-rooting the development-history tree so that $O+A$ is the root, the diagram is turned on its side and becomes a program-integration problem (situation (b)). The base program is now $O+A$, and the two variants of $O+A$ are O and $O+A+B$. Instead of treating the differences between O and $O+A$ as changes that were made to O to create $O+A$, they are now treated as changes made to $O+A$ to create O. For example, when O is the base program, a 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 $O+A$.

2.3. Optimistic Concurrency Control

An environment for programming in the large must provide concurrency control, *i.e.* 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.

3. DESCRIPTION OF THE INTEGRATION METHOD

The information used by our program integration method is encapsulated by a program representation called a *program dependence graph* (PDG). Different definitions of program dependence representations have been given, depending on the intended application; nevertheless, they are all variations on a theme introduced in [15], and share the common feature of having explicit representations of both control dependencies and data dependencies. The definition presented here has been adapted to the particular needs of the program-integration problem; it is similar, but not identical, to the program dependence representations used by others, such as the "program dependence graphs" defined in [7] and the "dependence graphs" defined in [16].

For program integration, there are two reasons to represent programs using program dependence graphs: (1) PDG's capture only relevant orderings of statements within control structures; (2) PDG's are a convenient representation on which to perform program slicing, which is used in the integration algorithm to determine the changes in behavior of each variant with respect to the base program.

3.1. The Program Dependence Graph

The definition of program dependence graph presented below is for a language with scalar variables, assignment statements, conditional statements, while loops, and a restricted kind of "output statement" called an *end statement*. An end statement, which can only appear at the end of a program, names one or more of the variables used in the program; when execution terminates, only those variables will have values in the final state. Intuitively, the variables named by the end statement are those whose final values are of interest to the programmer.

The program dependence graph for a program P, denoted by G_P, is a directed graph whose

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

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

b) For each variable x for which there is a path in the standard control-flow graph for P [1] on which x is used before being defined, 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 "$x := InitialState(x)$."

c) 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 "$FinalUse(x)$".

The edges of G_P represent *dependencies* between program components. An edge represents either a *control dependency* or a *data dependency.* Control dependency edges are labeled either **true** or **false**, and the source of a control dependency edge is always the entry vertex or a predicate vertex. A control dependency edge from vertex v_1 to vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately). A method for determining control dependency edges for arbitrary programs is given in [7]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependency edges of G_P can be determined in a much simpler fashion. For the language under consideration here, the control dependency edges reflect a program's nesting structure; a program dependence graph contains a *control dependency edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

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

ii) v_1 represents a control predicate, and v_2 represents a component of P immediately subordinate to the control construct whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.[2]

---

[1]A *directed graph* G consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c; we say that b is the *source* and c the *target* of the edge.

[2]In other definitions that have been given for control dependency 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 left out of our

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

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

i) v_1 is a vertex that defines variable x.

ii) v_2 is a vertex that uses x.

iii) Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control-flow graph for the program [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 its end.)

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

Flow dependencies are further classified as *loop independent* or *loop carried.* A flow dependency $v_1 \longrightarrow_f v_2$ is carried by loop L, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to i), ii), and iii) above, the following also hold:

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

v) Both v_1 and v_2 are enclosed in loop L.

A flow dependency $v_1 \longrightarrow_f v_2$ is loop independent, denoted by $v_1 \longrightarrow_{li} v_2$, if in addition to i), ii), and iii) above, there is an execution path that satisfies iii) 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 $v_1 \longrightarrow_{lc(L)} v_2$ and $v_1 \longrightarrow_{li} v_2$.

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

i) v_1 and v_2 both define the same variable.

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

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

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

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

---

definition because it is not necessary for our purposes.

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

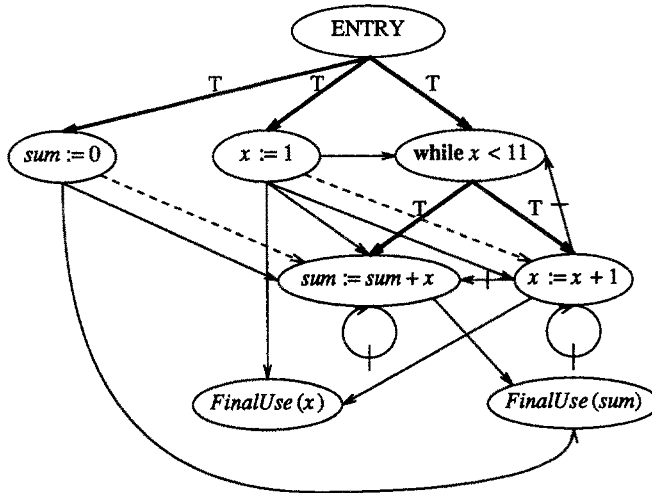


Figure 2. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependency edges, dashed arrows represent def-order dependency edges, solid arrows represent loop-independent flow dependency edges, and solid arrows with a hash mark represent loop-carried flow dependency edges.

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

Example. Figure 2 shows an example program and its program dependence graph. The bold-face arrows represent control dependency edges; dashed arrows represent def-order dependency edges; solid arrows represent loop-independent flow dependency edges; solid arrows with a hash mark represent loop-carried flow dependency edges.

The data-dependency edges of a program dependence graph (PDG) are computed using data-flow analysis. For the restricted language considered in this paper, the necessary computations can be defined in a syntax-directed manner (see [8]).

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

3.2. Program Slices

For a vertex s of a PDG G, the *slice* of G with respect to s, written as G / s, is a graph containing all vertices on which s has a transitive flow or control dependence (*i.e.* all vertices that can reach s via flow or control edges): $V(G/s) = \{ w \mid w \in V(G) \wedge w \longrightarrow^{*}_{c,f} s \}$. We extend the definition to a set of vertices $S = \bigcup_i s_i$ as follows: $V(G/S) = V(G/(\bigcup_i s_i)) = \bigcup_i V(G/s_i)$. It is useful to define $V(G/v) = \emptyset$ 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 $v \longrightarrow_{do(u)} w$ is included only if G/S contains the vertex u that is directly flow dependent on the definitions at v and w. In terms of the three types of edges in a PDG we define

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

3.3. Program Dependence Graphs and Program Semantics

The relationship between a program's PDG and the program's execution behavior has been addressed in [11] and [19]. The results from these papers are summarized below.

In [11] it is shown that if the program dependence graphs of two programs are isomorphic then the programs have the same behavior. 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 main result of [11] is the following theorem: (we use the symbol $\approx$ to denote isomorphism between program dependence graphs.)

THEOREM. (EQUIVALENCE THEOREM). *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 non-isomorphic program dependence graphs.

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 $V(G / S)$ [19].

THEOREM. (FEASIBILITY OF PROGRAM SLICES). *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.*

The significance of a slice is that it captures a portion of a program's behavior. The programs P' and P, corresponding to the slice G / S and the graph G, respectively, compute the same final values for all variables x for which *FinalUse* (x) is a vertex in S [19].

THEOREM. (SLICING THEOREM). *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.*

3.4. An Algorithm for Program Integration

An algorithm for integrating several related, but different variants of a base program (or determining that the variants incorporate interfering changes) has been presented in [10]. The algorithm presented there, called *Integrate*, 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 certain well-defined sense), Integrate produces a program M that exhibits the changed behavior of A and B with respect to *Base* as well as the behavior preserved in all three versions.

We now describe the steps of the integration algorithm. The first step determines slices that represent the changed behaviors of A and B and the behavior of *Base* preserved in both A and B; the second step combines these slices to form the merged graph G_M; the third step tests G_M for interference. The integration algorithm is illustrated with an example on which the algorithm succeeds (*i.e.* the variants do not interfere).

Step 1: Determining changed and preserved behavior

If the slice of variant G_A at vertex v differs from the slice of G_{Base} at v, then G_A and G_{Base} may compute different values at v. In other words, vertex v is a site that potentially exhibits

changed behavior in the two programs. Thus, we define the *affected points* of G_A with respect to G_{Base}, denoted by $D_{A,Base}$, to be the subset of vertices of G_A whose slices in G_{Base} and G_A differ $D_{A,Base} = \{ v \mid v \in V(G_A) \wedge (G_{Base} / v \neq G_A / v) \}$. We define $D_{B,Base}$ similarly. It follows that the slices $G_A / D_{A,Base}$ and $G_B / D_{B,Base}$ capture the respective behaviors of A and B that differ from $Base$.

Example. Figure 2 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: In variant A two statements have been added to the original program to compute the product of the integer sequence from 1 to 10; In variant B one statement has been added to compute the mean of the sequence. These two programs represent non-interfering extensions of the original summation program.

Variant A	Variant B
program *Main*	**program** *Main*
prod := 1;	*sum* := 0;
sum := 0;	*x* := 1;
x := 1;	**while** *x* < 11 **do**
while *x* < 11 **do**	*sum* := *sum* + *x*;
prod := *prod* * *x*;	*x* := *x* + 1
sum := *sum* + *x*;	**od**;
x := *x* + 1	*mean* := *sum* / 10
od	**end**(*x*, *sum*, *mean*)
end(*x*, *sum*, *prod*)	

The program dependence graphs for these two programs are shown in Figure 3.

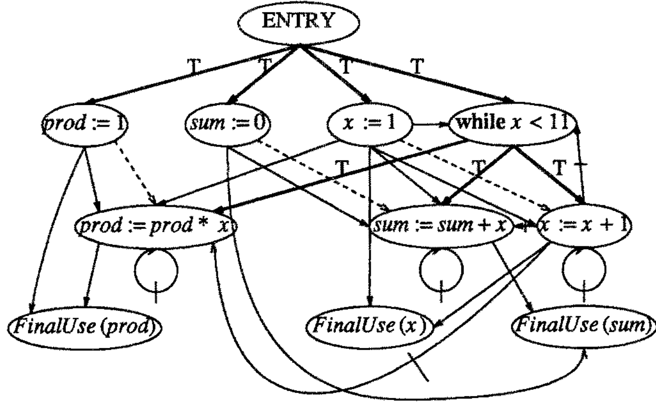
The set $D_{A,Base}$ contains three vertices: the assignment vertices labeled "*prod* := 1" and "*prod* := *prod* * *x*" as well as the final-use vertex for *prod*. Similarly, $D_{B,Base}$ contains two vertices: the assignment vertex labeled "*mean* := *sum* / 10" and the final-use vertex for *mean*.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

The preserved behavior of $Base$ in A corresponds to the slice $G_{Base} / \overline{D}_{A,Base}$, where $\overline{D}_{A,Base}$ is the complement of $D_{A,Base}$: $\overline{D}_{A,Base} = V(G_A) - D_{A,Base}$. We define $\overline{D}_{B,Base}$ similarly. Thus, the unchanged behavior common to both A and B is captured by the following slice: $G_{Base} / (\overline{D}_{A,Base} \cap \overline{D}_{B,Base})$.

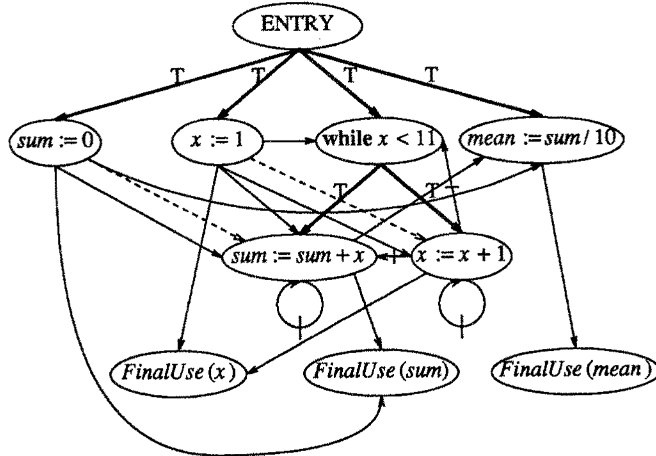
Example. In our example, the slice $G_{Base} / (\overline{D}_{A,Base} \cap \overline{D}_{B,Base})$ consists of G_{Base} in its entirety. That is, the graph that represents the behavior of the original program that is preserved in both variant A and variant B is identical to the graph shown in Figure 2.

Step 2: Forming the merged graph

The merged program dependence graph, G_M, is formed by unioning the three slices that represent the changed and preserved behaviors of the two variants:

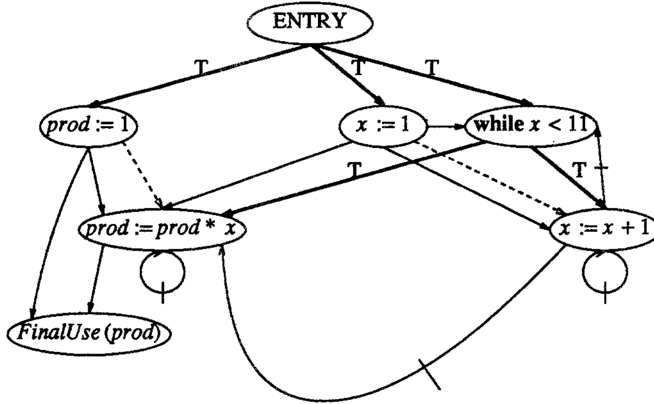


(a) The program dependence graph for variant A

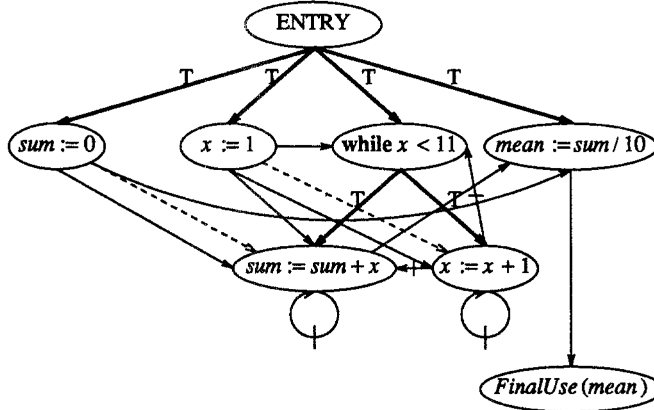


(b) The program dependence graph for variant B

Figure 3. The program dependence graphs for variants A and B.



(a) The slice $G_A / D_{A,Base}$.



(b) The slice $G_B / D_{B,Base}$.

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

$G_M = (G_A / D_{A,Base}) \cup (G_B / D_{B,Base}) \cup (G_{Base} / (\overline{D}_{A,Base} \cap \overline{D}_{B,Base}))$.

Example. The union of the slices from Figures 2 and 4 gives the program dependence graph G_M shown in Figure 5.

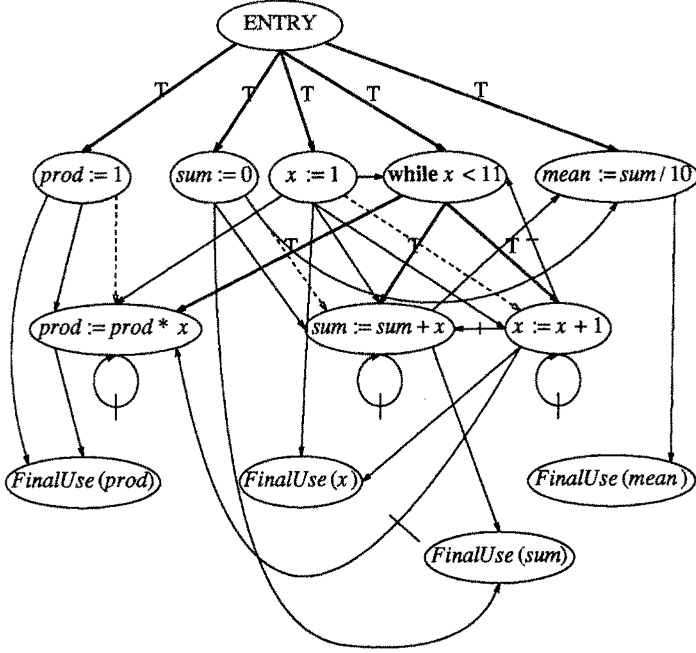


Figure 5. G_M is created by taking the union of the slices shown in Figures 2 and 4.

Step 3: Testing for interference

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

Example. An inspection of the merged graph shown in Figure 5 reveals that there is no interference; the slices $G_M / D_{A,Base}$ and $G_M / D_{B,Base}$ are identical to the graphs that appear in Figures 4(a) and 4(b), respectively.

The final step of the integration method involves reconstituting a program from the merged program dependence graph. However, it is possible that there is no such program; that is, the merged graph may be an infeasible program dependence graph. This is the second kind of interference that may occur.

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

Unfortunately, as we show in [8], the problem of determining whether it is possible to order a vertex's children is NP-complete. We have explored two approaches to dealing with this difficulty:

a) For graphs created by merging PDG's 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.

b) It is possible to side-step completely the need to solve an NP-complete problem by performing a limited amount of variable renaming. This technique can be used to avoid any difficult ordering step that remains after applying the techniques outlined in approach a).

The reader is referred to [8] and [10] for more details about feasibility testing, as well as a description of a method to reconstruct a program from the merged program dependence graph.

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

Example. The program dependence graph shown in Figure 5 corresponds to the program:

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

Using the Slicing Theorem and the definition of the merged graph G_M, we can show the following theorem, which characterizes the execution behavior of the integrated program in terms of the behaviors of the base program and the two variants [19]:

THEOREM. (INTEGRATION THEOREM). *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 on which the final states of A and Base agree, then the final state of M agrees with the final state of B on x, and (3) if y is a variable on which the final states of B and Base agree, then the final state of M agrees with the final state of A on y.

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.

4. WILL LANGUAGE-BASED PROGRAM INTEGRATION SCALE UP?

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 determining the program elements that may exhibit changed behavior and the test for interference had to be made *safe* rather than *exact*. Consequently, the integration algorithm will report interference in some cases where no real conflict exists. The issue of whether fully automatic integration turns out to be a realistic proposition will only be resolved once we have built the components for creating program dependence graphs, slicing them, and testing the merged graph for interference.

In any case, some integrations will report interference. For these situations it is not enough merely to detect interference; one needs a tool for *semi-automatic, interactive integration* so that the user can guide the integration process to a successful completion. Through such a tool, it will be possible for the user to examine sites of potential conflicts, which may or may not represent actual conflicts. The tool will make use of the merged program dependence graph that is built by the integration algorithm; this graph is a product of interfering as well as non-interfering integrations.

In many respects the interactive integration tool will be similar to PTOOL, an interactive tool developed by Kennedy's group at Rice [3]. PTOOL makes use of a program dependence representation to provide programmers with diagnostic information about potential problems that arise in parallelization.

In a similar fashion, the integration tool we are building will display information about individual program elements and program slices that indicate potential integration conflicts. By highlighting collections of program elements, it will indicate such information as: the affected points in the two variant programs, the slices with respect to the affected points, and slices of the two variants that become "intertwined" in the merged graph. The tool will also provide capabilities for the user to resolve conflicts and create a satisfactory merged program.

A preliminary implementation of a program-integration tool has been embedded in a program editor created using the Synthesizer Generator [17, 18]. Data-flow analysis on programs is carried out according to the editor's defining attribute grammar and used to construct their program dependence graphs. A slice command added to the editor makes it possible to highlight the elements of program slices. An integration command invokes the integration algorithm on the pro-

gram dependence graphs and reports whether the variant programs interfere.

We plan to experiment with ways of incorporating other integration facilities in this system. In particular, it may be desirable for users to be able to supply pragmas to the program-integration system. For instance, 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.

As to the practicality of our techniques, the basic techniques used in the integration algorithm (dependence analysis and slicing) are ones that have been applied successfully to programs of substantial size. Many of the techniques that are needed in order to adapt the integration algorithm to handle realistic languages (see below) have also been used in production-quality systems; these techniques include analysis of array index expressions to provide sharper information about actual dependencies among array references [2, 4, 22] and interprocedural data-flow analysis to determine "may-summary" information [5, 6].

4.1. Applicability to Realistic Languages

We have recently made progress towards extending the integration technique to handle languages with procedure calls and pointer-valued variables. The major stumbling block when making such extensions is devising a suitable extension of the program dependence representation. Our recent work in this direction is summarized below.

4.1.1. Interprocedural slicing using dependence graphs

As a first step toward extending our integration algorithm to handle languages with procedures, we have devised a multi-procedure dependence representation and have developed a new algorithm for interprocedural slicing that uses this representation [9]. 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 [21].

The method described in [21] 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 A              procedure B(y)         procedure C(z)
   x := 0;                y := y + 1             call B(z);
   w := 0;             return                    z := z + 1
   call B(x);                                 return
   call C(w);
end(w, x)
```

Using the algorithm from [21] to slice this system with respect to variable x at the end of program A, we obtain the following:

```
program A                  procedure B (y)          procedure C (z)
    x := 0;                    y := y + 1               call B (z);
    w := 0;                return                   return
    call B (x)
    call C (w);
end(x)
```

However, further inspection shows that the value of x at the end of program A is not affected by the initialization of w in A, nor by the call on C in A, nor by procedure C. The reason these components are included in the slice is (roughly) the following: The statement "**call** $B(x)$" in program A causes the slice to "descend" into procedure B. When the slice reaches the beginning of B it "ascends" to *all* sites that call B, both the site in A at which it "descended" and the (irrelevant) site in C. Similarly, the slice in C ascends to the call site in A.

By contrast, our algorithm for interprocedural slicing correctly accounts for the calling context of a called procedure; on the example given above, procedure C is left out of the slice. 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. The context free part of the linkage grammar models the system's procedure-call structure; it includes one nonterminal and one production for each procedure in the system. If procedure P contains no calls, the right-hand side of the production for P is ε; otherwise, there is one right-hand side nonterminal for each call site in P. The attributes in the linkage grammar represent the parameters, globals, and return values that are transferred across procedure boundaries. Attribute dependencies, expressed via the attribute equations of the linkage-grammar's productions are used to model intraprocedural dependencies among these elements.

Transitive dependencies between attributes due to procedure calls are then determined using a standard attribute-grammar construction: the computation of the nonterminals' *subordinate characteristic graphs*. These dependencies are the key to the slicing algorithm; they permit the algorithm to "come back up" from a procedure call (*e.g.* from procedure B 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.

4.1.2. Dependence analysis for pointer variables

We have devised a method for determining data dependencies between program statements for programming languages that have pointer-valued variables (*e.g.* Lisp and Pascal). The method determines data dependencies 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.

For scalar variables, one way to compute *flow dependencies* is in the form of *use-definition chains*. To do this one first computes a more general piece of information: the set of *reaching definitions* for each program point [1, 14]. A definition of variable x at some program point p *reaches* point q if there is an execution path from p to q such that no other definition of x appears on the path. The set of reaching definitions for a program point q is the set of program points that generate definitions that reach q. Program point q is *flow dependent* on all members of the reaching-definition set that define variables used at q.

To extend the concept of flow dependence for languages that manipulate heap-allocated storage, it is necessary to rephrase the definition in terms of *memory locations* rather than *variables*.

> Program point q is flow dependent on point p if p writes to a memory location that may be read by q.

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 dependencies between constructs that manipulate and access heap-allocated storage, our starting point is the method described in [13], which, for each program point q, determines a set of structures that approximate 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 dependencies 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.

ACKNOWLEDGEMENTS

Several others have participated in the development of the ideas described above. We are indebted to J. Prins for many pleasurable discussions and for his contributions to the development of the program-integration algorithm. Subsequent work to extend the integration algorithm's range of applicability has been carried out in collaboration with D. Binkley, P. Pfeiffer, and W. Yang.

REFERENCES

1. Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).
2. Allen, J.R., "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Dept. of Math. Sciences, Rice Univ., Houston, TX (April 1983).
3. Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A., "PTOOL: A semi-automatic parallel programming assistant," Tech Rep. COMP TR86-31, Dept. of Computer Science, Rice Univ., Houston, TX (January 1986).
4. Bannerjee, U., "Speedup of ordinary programs," Ph.D. dissertation and Tech. Rep. R-79-989, Dept. of Computer Science, University of Illinois, Urbana, IL (October 1979).

5. Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York (1979).

6. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," Tech. Rep. COMP TR87-62, Dept. of Computer Science, Rice Univ., Houston, TX (October 1987).

7. Ferrante, J., Ottenstein, K., and Warren, J., "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems* 9(3) pp. 319-349 (July 1987).

8. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," TR-690, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1987).

9. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," Extended abstract, Computer Sciences Department, University of Wisconsin, Madison, WI (November 1987).

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

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

12. Horwitz, S., Prins, J., and Reps, T., "Support for integrating program variants in an environment for programming in the large," in *Proceedings of the International Workshop on Software Version and Configuration Control 88,* (Grassau, W. Germany, Jan. 28-29, 1988), (1988).

13. Jones, N.D. and Muchnick, S.S., "Flow analysis and optimization of Lisp-like structures," in *Program Flow Analysis: Theory and Applications,* ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

14. Kennedy, K., "A survey of data flow analysis techniques," in *Program Flow Analysis: Theory and Applications,* ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

15. Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).

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

17. Reps, T. and Teitelbaum, T., "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* (Pittsburgh, PA, Apr. 23-25, 1984), *ACM SIGPLAN Notices* **19**(5) pp. 42-48 (May 1984).

18. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: Reference Manual,* Dept. of Computer Science, Cornell Univ., Ithaca, NY (August 1985, Second Edition: July 1987).

19. Reps, T. and Yang, W., "The semantics of program slicing," Tech. Rep. in preparation, Computer Sciences Department, University of Wisconsin, Madison, WI ().

20. Tichy, W.F., "RCS: A system for version control," *Software – Practice & Experience* **15**(7) pp. 637-654 (July 1985).

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

22. Wolfe, M.J., "Optimizing supercompilers for supercomputers," Ph.D. dissertation and Tech. Rep. R-82-1105, Dept. of Computer Science, University of Illinois, Urbana, IL (October 1982).