Program Integration for Languages with Procedure Calls

DAVID BINKLEY, SUSAN HORWITZ, and THOMAS REPS
University of Wisconsin − Madison

Given a program *Base* and two variants, *A* and *B*, each created by modifying separate copies of *Base*, the goal of program integration is to determine whether the modifications interfere, and if they do not, to create an integrated program that incorporates both sets of changes as well as the portions of *Base* preserved in both variants. Text-based integration techniques, such as the one used by the UNIX *diff3* utility, are obviously unsatisfactory because one has no guarantees about how the execution behavior of the integrated program relates to the behaviors of *Base*, *A*, and *B*. The first program-integration algorithm to provide such guarantees was developed by Horwitz, Prins, and Reps. However, a limitation of that algorithm is that it only applies to programs written in a restricted language—in particular, the algorithm does *not* handle programs with procedures. This paper describes a generalization of the Horwitz-Prins-Reps algorithm that handles programs that consist of multiple (and possibly mutually recursive) procedures.

We show that two straightforward generalizations of the Horwitz-Prins-Reps algorithm yield unsatisfactory results. The key issue in developing a satisfactory algorithm is how to take into account different calling contexts when determining what has changed in the variants *A* and *B*. Our solution to this problem involves identifying two different kinds of affected components of *A* and *B*: those affected regardless of how the procedure is called, and those affected by a changed or new calling context. The algorithm makes use of interprocedural program slicing to identify these components, as well as components in *Base*, *A*, and *B* with the same behavior.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*programmer workbench*; D.2.3 [**Software Engineering**]: Coding—*program editors*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*enhancement, restructuring, version control*; D.2.9 [**Software Engineering**]: Management—*programming teams, software configuration management*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures, procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers, interpreters, optimization*; E.1 [**Data Structures**] *graphs*.

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, CCR-8958530, and CCR 91-00424, by the Defense Advanced Research Projects Agency under ARPA Order Nos. 6378 and 8856, monitored by the Office of Naval Research under contracts N00014-88-K-0590 and N00014-92-J-1937, as well as by grants from IBM, DEC, HP, Xerox, 3M, Eastman Kodak, and the Cray Research Foundation.

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

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Control dependence, data dependence, data-flow analysis, flow-insensitive summary information, program dependence graph, program slicing, semantics-based program integration.

---

1. INTRODUCTION

Program integration concerns the merging process necessary when a program's source code diverges into multiple variants. The need for program integration arises in many situations:

(1) When a system is "customized" by a user and simultaneously upgraded by a maintainer, and the user desires a customized, upgraded version.

(2) When a system is being developed by multiple programmers who may simultaneously work with separate copies of the source files.

(3) When several versions of a program exist and the same enhancement or bug-fix is to be made to all of them.

Having an integration tool to provide assistance when tacking such problems would obviously be useful.

Given a program *Base* and two variants, *A* and *B*, each created by modifying separate copies of *Base*, the goal of program integration is to determine whether the modifications interfere, and if they do not, to create an integrated program that includes both sets of changes as well as the portions of *Base* preserved in both variants.[1] At present, the only available tools for integration (*e.g.*, the UNIX[2] utility *diff3*) implement an operation for merging files as strings of text. This approach has the advantage that it is applicable to merging documents, data files, and other text objects as well as 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 how the execution behavior of the program produced by a purely *textual* merge relates to the execution behaviors of the programs that are the arguments to the merge. For example, if one variant contains changes only on lines 5–10, while the other variant contains changes only on lines 15–20, *diff3* would deem these changes to be interference-free; however, just because changes are made at different places in a program is no reason to believe that the changes are free of undesirable interactions. The merged program produced by a text-based integration tool must, therefore, be carefully checked for conflicts that might have been introduced by the merge.

---

[1] More generally, we may be interested in integrating an arbitrary number of variants with respect to *Base*; however, for the sake of exposition we consider the common case of two variants *A* and *B*.

[2] UNIX is a Trademark of AT&T Bell Laboratories.

In contrast, our goal is to create a *semantics-based* tool for program integration. A semantics-based program-integration tool has the following characteristics:

(1) The integration tool makes use of knowledge of the programming language to determine whether the changes made to the base program to create the two variants have undesirable semantic interactions; only if there is no such interference will the tool produce an integrated program.
(2) The integration tool provides guarantees about how the execution behavior of the integrated program relates to the execution behaviors of the base program and the two variants.

Determining any non-trivial property of a program's execution behavior is undecidable; thus, a semantics-based integration tool must use techniques that provide safe approximations to undecidable problems.

The first algorithm for semantics-based program integration was formulated by Horwitz, Prins, and Reps [7]. This algorithm will be referred to hereafter as the *HPR algorithm*. A limitation of the HPR algorithm is that it applies only to a restricted programming language—one that does not include procedures and procedure calls. This paper concerns how to integrate programs that consist of multiple (and possibly mutually recursive) procedures. The paper includes two major contributions:

(1) We define a model of program integration for programs with procedures. This model provides correctness criteria for integration algorithms.
(2) We present a multi-procedure integration algorithm that satisfies the model.

The remainder of the paper is organized as follows: Section 2 summarizes background material; in particular, it describes the HPR integration algorithm, system dependence graphs—the graph representation used by our multi-procedure integration algorithm—and interprocedural slicing. Section 3 presents our model for multi-procedure integration; Section 4 describes an algorithm that satisfies this model. Section 5 describes the relationship between this algorithm and the algorithm published in the first author's thesis. Section 6 describes an implementation of the multi-procedure integration algorithm.

2. BACKGROUND

In this section we review previous work that forms the basis for our new multi-procedure program-integration algorithm. In reviewing this work we introduce some notation and concepts that differ from those that have appeared in previous publications. These changes are introduced to simplify the presentation of the new algorithm in Section 4.

Throughout the paper, we use superscripts to distinguish between related quantities in the HPR algorithm and the multi-procedure integration algorithm. Those used in the former have the superscript "*hpr*"; those used in the latter have the superscript "*S*".

2.1. The Horwitz-Prins-Reps Algorithm for Program Integration

The HPR algorithm was designed to satisfy the model of semantics-based program integration given below. This model makes use of a non-standard notion of the (collecting) semantics of a program. Rather than the sequence of states that arise at a program component, we are concerned with the sequence of values *produced* by a program component. By a "program component" we mean an assignment statement, an output statement, or the predicate of a while loop or conditional statement. By "the sequence of values produced by a component" we mean: for an assignment statement, the sequence of values assigned to the target variable; for an output statement, the sequence of output values; and for a predicate, the sequence of boolean values to which the predicate evaluates.

Model of Program Integration for Programs Without Procedures

(1) Programs are written in a simplified programming language that has only assignment statements, conditional statements, while loops, and output statements. The language does not include input statements; however, a program can use a variable before assigning to it, in which case the variable's value comes from the initial state.

(2) When an integration algorithm is applied to base program *Base* and variant programs *A* and *B*, and if integration succeeds—producing program *M*—then for any initial state σ on which *Base*, *A*, and *B* all terminate normally,[3] the following properties concerning the executions of *Base*, *A*, *B*, and *M* on σ must hold:

   (i) *M* terminates normally.

   (ii) *M* captures the changed behavior of *A*: For any program component c_A in variant *A* for which there is no corresponding component of *Base* that produces the same sequence of values as c_A, there is a component of *M* that produces the same sequence of values as c_A.

   (iii) *M* captures the changed behavior of *B*: For any program component c_B in variant *B* for which there is no corresponding component of *Base* that produces the same sequence of values as c_B, there is a component of *M* that produces the same sequence of values as c_B.

   (iv) *M* captures the behavior of *Base* preserved in *A* and *B*: If there are corresponding program components c_{Base}, c_A, and c_B that all produce the same sequence of values then there is a component of *M* that produces the same sequence of values as c_{Base}, c_A, and c_B.

(3) Program *M* is created only from components that occur in programs *Base*, *A*, and *B*.

Properties (1) and (3) are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. An informal statement of Property (2) is "changes in the behavior of *A* and *B* with respect to *Base* must be incorporated in the integrated program, along with the unchanged behavior of all three." *Any* program *M* that satisfies Properties (1), (2), and (3) *integrates Base*, *A*, and *B*; if no such program exists then *A* and *B interfere* with respect to *Base*. However, the problem of whether a program exists that satisfies Property (2) is not decidable, even under the restrictions given by Properties (1) and (3) [19]; consequently, to be safe any program-integration algorithm must sometimes report interference—and consequently fail to produce an integrated program—even though there is actually *no* interference (*i.e.*, even when there is *some* program that meets the criteria given above).

Although determining whether a program modification actually leads to a change in program behavior is undecidable, the HPR algorithm is able to determine a safe approximation to the set of program elements with changed behavior by comparing each of the variants with the base program. To determine this information, the HPR algorithm employs a program representation that is similar to the *program dependence graphs* used previously in vectorizing and parallelizing compilers [6, 9]. The HPR algorithm also makes use of Weiser's notion of a *program slice* [12, 21] to find the statements of a program that contribute to the computation of the program elements with changed behavior.

The HPR algorithm assumes that program components are tagged so that corresponding components can be identified in *Base*, *A*, and *B*. Component tags can be provided by a special editor that obeys the following protocol:[4]

(1) When a copy of a program is made—*e.g.*, when a copy of *Base* is made in order to create a new variant—each component in the copy is given the same tag as the corresponding component in the original program.
(2) The operations on program components supported by the editor are insert, delete, and move. A newly inserted component is given a previously unused tag; the tag of a component that is deleted is never re-used; a component that is moved from one position to another retains its tag.
(3) The tags on components persist across different editing sessions and machines.
(4) Tags are allocated by a single server, so that two different editors cannot allocate the same new tag.

———————————————————

[3] There are two ways in which a program may fail to terminate normally on some initial state: (1) the program contains a nonterminating loop, or (2) a fault occurs, such as division by zero.

[4] 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 [5], GANDALF [11], and the Synthesizer Generator [13].

Component tags furnish the means for identifying how the program-dependence-graph vertices in different versions correspond. It is the tags that are used to determine "identical" vertices when operations are performed using vertices from different program dependence graphs.

2.1.1. Program dependence graphs

A program dependence graph (or PDG) is a directed graph.[5] The vertices of a program's PDG represent the predicates, assignment statements, and output statements of the program. In addition, there is a distinguished vertex called the *entry vertex*, and an *initial-definition vertex* for each variable that may be used before being defined. There are two kinds of edges: *control dependence edges* and *data dependence edges*. The source of a control dependence edge is either the entry vertex or a predicate vertex and each edge is labeled either **true** or **false**. A control dependence edge $v \longrightarrow_c w$ from vertex v to vertex w means that during execution, whenever the predicate represented by v is evaluated and its value matches the label on the edge to w, then the program component represented by w will eventually be executed if the program terminates normally.

There are two kinds of data dependence edges, *flow dependence edges* and *def-order dependence edges*. A flow dependence edge $v \longrightarrow_f w$ runs from a vertex v that represents an assignment to a variable x to a vertex w that represents a use of x reached by that assignment. Flow dependences are further classified as *loop-independent* or *loop-carried*. A flow dependence is loop-carried (denoted by $v \longrightarrow_{lc} w$) if there is an x-definition free path in the program's control-flow graph from v to w that includes a backedge; otherwise, the flow dependence is loop-independent (denoted by $v \longrightarrow_{li} w$). A def-order edge $v \longrightarrow_{do(u)} w$ runs between two vertices, v and w, where both v and w represent assignments to x, the definitions at v and w reach a common use of x at u (*i.e.*, $v \longrightarrow_f u$ and $w \longrightarrow_f u$), and v lexically precedes w.

Example. Figure 1 shows an example program and its PDG.

<< Insert Figure 1 >>

2.1.2. Slicing

Backward slicing

The vertices of the *backward slice* of PDG G with respect to a vertex v are all vertices on which v has a transitive flow or control dependence (*i.e.*, all vertices that can reach v via flow and/or control edges). The

---

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

vertices of the backward slice of G with respect to a set of vertices S are those that can reach some vertex in S. We define an operator b^{hpr} with signature $PDG \times vertex\text{-}set \rightarrow PDG \times vertex\text{-}set$ that pairs the graph being sliced with the vertices of the slice:

$$b^{hpr}(G, S) =_{df} (G, \{ w \in V(G) \mid w \longrightarrow^*_{c,f} v \text{ and } v \in S \}).$$

For a single vertex, we define $b^{hpr}(G, v) = b^{hpr}(G, \{ v \})$ and define $b^{hpr}(G, v) = (G, \varnothing)$ for any $v \notin V(G)$. As a notational shorthand, we sometimes use the name of a program to denote the corresponding PDG; for example, $b^{hpr}(Base, S)$ is sometimes used in place of $b^{hpr}(G_{Base}, S)$.

The backward slice of graph G with respect to vertex set S (which we often shorten to "slice"), is denoted by $Induce(b^{hpr}(G, S))$,[6] where $Induce : PDG \times vertex\text{-}set \rightarrow PDG$ is the function that returns the subgraph of the PDG induced by the vertex set. (A def-order edge $v \longrightarrow_{do(u)} w$ is only included in the induced edge set if u, v, and w are all included in the vertex set.) Formally, *Induce* is defined as follows:

$$Induce(G, V) =_{df} (V, E'), \text{ where } E' = \begin{array}{l} \{ v \longrightarrow_f w \in E(G) \mid v, w \in V \} \\ \cup \{ v \longrightarrow_c w \in E(G) \mid v, w \in V \} \\ \cup \{ v \longrightarrow_{do(u)} w \in E(G) \mid u, v, w \in V \}. \end{array}$$

Where there is no ambiguity, we omit *Induce*; for example, to compare the slices of G with respect to vertices v and u, we use $b^{hpr}(G, v) = b^{hpr}(G, u)$ in place of $Induce(b^{hpr}(G, v)) = Induce(b^{hpr}(G, u))$. Also, as a further notational shorthand, we omit the operator that selects the vertex-set component of a (graph, vertex-set) pair when intersecting the vertex sets of two slices or when testing whether a vertex is in a slice. For example, we use $u \in b^{hpr}(G, v)$ in place of $u \in VertexSet(b^{hpr}(G, v))$.

Example. Figure 2 shows the graph that results from slicing the PDG from Figure 1 with respect to the vertex for statement "$i := i + 1$" together with the program to which it corresponds.

<< Insert Figure 2 >>

Forward slicing

Forward slicing is the dual of backward slicing: whereas a backward slice identifies those program components that can reach a given component (along flow and/or control edges), a forward slice identifies those components that can be reached *from* a given component.

$$f^{hpr}(G, S) =_{df} (G, \{ w \in V(G) \mid v \longrightarrow^*_{c,f} w \text{ and } v \in S \}).$$

---

[6] In previous papers we used the notation G / v to denote this PDG.

2.1.3. Summary of the HPR algorithm

The first step of the HPR algorithm determines the slices of *A* and *B* that are changed from *Base*; the second step determines the slices of *Base* that are preserved in both *A* and *B*; the third step combines the changed and preserved slices to form the merged PDG G_M; the fourth step tests G_M for interference, and if there is no interference produces a program *M* whose PDG is G_M.

Step 1: Determining changed slices

If the slice of variant *A* with respect to vertex *v* differs from the slice of *Base* with respect to *v*, then *A* and *Base* may compute a different sequence of values at *v* [14]. In other words, vertex *v* is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of *A* with respect to *Base*, denoted by $AP^{hpr}(A, Base)$, to be the subset of vertices of *A* whose slices in *Base* and *A* differ.

$$AP^{hpr}(A, Base) =_{df} (A, \{ v \in V(A) \mid b^{hpr}(Base, v) \neq b^{hpr}(A, v) \}).$$

(As with the slicing operators, the various integration operators all return a (PDG, vertex-set) pair.) We define $AP^{hpr}(B, Base)$ similarly.

To compute $AP(A, Base)$ and $AP^{hpr}(B, Base)$ efficiently, we use special subsets of the affected points, called the *directly affected points*. First, define the sets of incoming edges for a vertex *v* in PDG *G* as follows:

$$IncomingControl^{hpr}(G, v) =_{df} \{ w \longrightarrow_{c} v \in E(G) \}$$
$$IncomingLoopCarriedFlow^{hpr}(G, v) =_{df} \{ w \longrightarrow_{lc} v \in E(G) \}$$
$$IncomingLoopIndependentFlow^{hpr}(G, v) =_{df} \{ w \longrightarrow_{li} v \in E(G) \}$$
$$IncomingDefOrder^{hpr}(G, v) =_{df} \{ w \longrightarrow_{do(v)} x \in E(G) \}.$$

The last of these definitions will be somewhat puzzling if a def-order edge $w \longrightarrow_{do(v)} x$ is thought of as an edge directed from *w* to *x* and labeled with *v*. However, it is often useful to think of def-order edge $w \longrightarrow_{do(v)} x$ as a hyper-edge directed from *w* to *x* to *v*; it is in this sense that a def-order edge is an incoming edge for *v*.

Vertex *v* is a *directly affected point* of *A* with respect to *Base* iff *v* is in *A* but not *Base*, or if *v* has different incoming edges in *A* and *Base*. Thus, the set of directly affected points of *A* with respect to *Base*, denoted by $DAP^{hpr}(A, Base)$, is defined as follows:

$$\begin{aligned} &DAP^{hpr}(A, Base) =_{df} \\ &\quad (A, \{ v \in V(A) \mid \quad v \notin V(Base) \\ &\qquad\qquad \vee\ IncomingControl^{hpr}(A, v) \neq IncomingControl^{hpr}(Base, v) \\ &\qquad\qquad \vee\ IncomingLoopCarriedFlow^{hpr}(A, v) \neq IncomingLoopCarriedFlow^{hpr}(Base, v) \\ &\qquad\qquad \vee\ IncomingLoopIndependentFlow^{hpr}(A, v) \neq IncomingLoopIndependentFlow^{hpr}(Base, v) \\ &\qquad\qquad \vee\ IncomingDefOrder^{hpr}(A, v) \neq IncomingDefOrder^{hpr}(Base, v) \}). \end{aligned}$$

Vertices in $DAP^{hpr}(A, Base)$ clearly have different backward slices in *A* and *Base*. We can identify *all* the vertices of *A* whose backward slices are different in *Base* (*i.e.*, the affected points, $AP^{hpr}(A, Base)$) by finding those vertices in the forward slice of *A* with respect to $DAP^{hpr}(A, Base)$. Thus, we have

$$AP^{hpr}(A, Base) = f^{hpr}(DAP^{hpr}(A, Base)).$$

The slices $b^{hpr}(AP^{hpr}(A, Base))$ and $b^{hpr}(AP^{hpr}(B, Base))$ capture all the slices of A and B (respectively) that differ from *Base*, and so we make the following definitions:

$$\Delta^{hpr}(A, Base) =_{df} b^{hpr}(AP^{hpr}(A, Base)) \quad \text{and} \quad \Delta^{hpr}(B, Base) =_{df} b^{hpr}(AP^{hpr}(B, Base)).$$

Step 2: Determining preserved slices

A vertex that has the same slice in all three programs is guaranteed to exhibit the same behavior [14]. Thus, we define the *preserved points* of *Base*, *A*, and *B*, denoted by $Pre^{hpr}(A, Base, B)$, to be those vertices of *Base* that have the same slice in *Base*, *A*, and *B*:

$$Pre^{hpr}(A, Base, B) =_{df} (Base, \{ v \in V(Base) \mid b^{hpr}(A, v) = b^{hpr}(Base, v) = b^{hpr}(B, v) \}).$$

Step 3: Forming the merged graph

The merged graph G_M is formed by taking the graph union[7] of the slices that characterize the changed behavior of *A*, the changed behavior of *B*, and the behavior of *Base* preserved in both *A* and *B*:

$$G_M =_{df} Induce(\Delta^{hpr}(A, Base)) \cup Induce(\Delta^{hpr}(B, Base)) \cup Induce(Pre^{hpr}(A, Base, B)).$$

Step 4: Testing for interference

There are two possible ways by which the graph G_M can fail to represent a satisfactory integrated program; we refer to them as "Type I interference" and "Type II interference." The test for Type I interference determines whether any of the changed slices of *A* and *B* (with respect to *Base*) were "corrupted" when they were combined to form G_M. Interference is reported when a vertex in $\Delta^{hpr}(A, Base)$ is also a directly affected point of G_M with respect to *A* (or if the analogous condition holds for *B*); that is, Type I interference exists iff

$$\Delta^{hpr}(A, Base) \cap DAP^{hpr}(G_M, A) \neq \emptyset \qquad \text{or} \qquad \Delta^{hpr}(B, Base) \cap DAP^{hpr}(G_M, B) \neq \emptyset.$$

(A different definition of Type I interference was used in [7]; the two definitions are shown to be equivalent in [15].)

The final step of the HPR algorithm involves reconstituting a program from G_M. However, it is possible that no such program exists—the merged graph can be an *infeasible* PDG; this is Type II interference. (The crux of program reconstitution is to determine whether it is possible to order the program components so as to preserve the dependences represented by the PDG edges; see [7] for a discussion of program

---

[7] Given directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the *graph union*, $G_1 \cup G_2$, is defined as: $(V_1 \cup V_2, E_1 \cup E_2)$.

reconstitution.[8]) If neither kind of interference occurs, a program whose PDG is G_M is returned as the result of the integration.

One of the most important aspects of the HPR algorithm is that it provides *semantic* guarantees about the behavior of the integrated program. It has been shown that the HPR algorithm satisfies the integration model of Section 2.1 (and, in particular, Property (2), the semantic criterion for integration) [22].

2.2. The System Dependence Graph and Interprocedural Slicing

Two important steps in extending the HPR algorithm to handle programs with procedures are (1) extending the graph representation of programs to include procedure calls, and (2) extending the algorithm for program slicing to handle slices that cross procedure boundaries. These extensions were defined in [8]; summaries are given below.

2.2.1. The system dependence graph

The system dependence graph (SDG) extends previous dependence representations to accommodate collections of procedures (with procedure calls) rather than just monolithic programs. Our definition of the SDG models a language with the following properties:

(1) A complete *system* consists of a single (main) procedure and a collection of auxiliary procedures.[9]

(2) Parameters are passed by value-result.[10]

(3) There are no call sites of the form $P(x, x)$ or $P(g)$, where g is a global variable. (The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, we do not discuss global variables explicitly.)

An SDG is made up of a collection of procedure dependence graphs connected by interprocedural control- and flow-dependence edges. Procedure dependence graphs are similar to the program dependence graphs described in Section 2.1.1 except that they include vertices and edges representing call statements, parameter passing, and transitive data dependences due to calls (we will abbreviate both procedure dependence graphs and program dependence graphs by "PDG"). A call statement is represented using a *call-site* vertex; parameter passing is represented using four kinds of *parameter* vertices: on the calling side, parameter passing is represented by two kinds of vertices, called *actual-in* and *actual-out* vertices, which are control

---

**A minor correction to the program-reconstitution algorithm is given in [1].

[9] Because the term "program dependence graph" is associated with graphs that represent single procedure programs, we introduced the term "system dependence graph" for the dependence graphs that represent multi-procedure programs [8]. Similarly, we use "system"—rather than "program"—to mean a program with multiple procedures.

[10] Techniques for handling parameters passed by reference and for dealing with aliasing are discussed in [8].

dependent on the call vertex; in the called procedure, parameter passing is represented by two kinds of vertices, called *formal-in* and *formal-out* vertices, which are control dependent on the procedure's entry vertex. Actual-in and formal-in vertices are included for every global variable that may be used or modified as a result of the call and for every parameter; formal-out and actual-out vertices are included only for global variables and parameters that may be modified as a result of the call. (Interprocedural side-effect analysis [4] is used to determine which global variables need to be represented by parameter vertices.)

Summary edges are added from actual-in vertices to actual-out vertices to represent transitive data dependences due to called procedures. These edges are defined as follows.

DEFINITION. $Call_Q$ denotes a call-site vertex that represents a call on procedure Q; $Call_Q.a_{in}$ denotes an actual-in vertex subordinate to $Call_Q$, and $Call_Q.a_{out}$ denotes an actual-out vertex subordinate to $Call_Q$. Similarly, $Enter_Q$ denotes the entry vertex for procedure Q; $Enter_Q.a_{in}$ denotes a formal-in vertex subordinate to $Enter_P$ and $Enter_Q.a_{out}$ denotes a formal-out vertex subordinate to $Enter_Q$.

DEFINITION. CF_P denotes the *control-flow dependence subgraph* of PDG G_P (*i.e.*, G_P with no def-order edges).

$$CF_p =_{df} (V(G_p), \{\, e \in E(G_p) \mid e \text{ is a flow or control edge} \,\}).$$

For each procedure P, the *augmented control-flow dependence subgraph*, denoted by ACF_P, is CF_P augmented with summary edges. The existence of a summary edge in ACF_P at a call-site on procedure Q depends on the existence of a path in ACF_Q from a formal-in vertex to a formal-out vertex. In the presence of recursion, this path may in turn depend on the existence of a path in ACF_P. This interdependence is captured by defining the *ACF* graphs recursively, as the least solution to the following set of equations (there is one equation for each procedure P in the system).

DEFINITION (ACF_P).

$$ACF_P =_{df} (V(CF_P), E(CF_P) \cup SE_P), \text{ where}$$
$$SE_P =_{df} \{\, Call_Q.a_{in} \longrightarrow_s Call_Q.b_{out} \mid Call_Q \in V(CF_P) \wedge Enter_Q.a_{in} \longrightarrow Enter_Q.b_{out} \in (E(ACF_Q))^+ \,\}.$$

The *summary edges* for procedure P are those edges in SE_P (*i.e.*, those edges in ACF_P but not in CF_P).

Note that the last term in the above definition refers to $(E(ACF_Q))^+$—the transitive closure of $E(ACF_Q)$—because the edges of ACF_P depend on the existence of *paths* in ACF_Q.

Summary edges can be computed using a variation on the technique used to compute the subordinate-characteristic graphs of an attribute grammar's nonterminals. The computation of summary edges is described in [8].

Procedure dependence graphs are connected to form an SDG using three new kinds of edges: (1) a *call* edge is added from each call-site vertex to the corresponding procedure-entry vertex; (2) a *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

Example. Figure 3 shows an example system and its SDG. (In Figure 3 as well as in the remaining figures of the paper, def-order edges are not shown and the edges representing control dependences are shown unlabeled; all such edges would be labeled **true**.)

<< Insert Figure 3 >>

2.2.2. Interprocedural slicing

Interprocedural slicing can be defined as a reachability problem using the SDG, just as in*tra*procedural slicing is defined as a reachability problem using the PDG. The slices obtained using this approach are the same as those obtained using Weiser's interprocedural-slicing method [21]. However, this approach does not produce slices that are as precise as possible, because it considers paths in the graph that are not possible execution paths. For example, there is a path in the graph shown in Figure 3 from the vertex of procedure *Main* labeled "$x_{in} := sum$" to the vertex of *Main* labeled "$i := y_{out}$." However, this path corresponds to procedure *Add* being called by procedure *A*, but returning to procedure *Increment*, which is not possible. The value of *i* after the call to procedure *A* in *Main* is independent of the value of *sum* before the call, and so the vertex labeled "$x_{in} := sum$" should not be included in the slice with respect to the vertex labeled "$i := y_{out}$".[11]

To determine more precise interprocedural slices, an interprocedural slice of an SDG *G* with respect to a set of vertices *S* is computed using two passes over the graph. The summary edges described in Section 2.2.1 are used to permit "moving across" a call site without having to descend into the called procedure; thus, there is no need to keep track of calling context explicitly to ensure that only legal execution paths are traversed. Both passes operate on the SDG, traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges. The traversal in Pass 1 starts from all vertices in *S* and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but *not* along def-order or parameter-out edges. The traversal in Pass 2 starts from all vertices reached in Pass 1 and goes backwards along flow edges, control edges, summary edges, and

[11] A similar concern with non-executable paths arises in the context of interprocedural dataflow analysis [10, 20].

parameter-out edges, but *not* along def-order, call, or parameter-in edges. The result of an interprocedural slice consists of the sets of vertices encountered during Pass 1 and Pass 2, and the set of edges induced by this vertex set.

In the remainder of the paper, we use the operators *b1* and *b2* to designate the individual passes of the slicing algorithm. Operators *b1* and *b2* are applied to an SDG and a set of vertices, and return a pair consisting of their first argument along with a set of vertices; thus, each has the signature $SDG \times vertex\text{-}set \rightarrow SDG \times vertex\text{-}set$.

The operator *b* is used to denote the composition $b2 \circ b1$. Thus, the (full backward) slice of graph *G* with respect to vertex set *S* is denoted by $Induce(b(G, S))$, where *Induce* is the extension to SDGs of the *Induce* operator defined in Section 2.1.2.

As with backward slicing, an (interprocedural) forward slice with respect to a given set of program components can be computed using two edge-traversal passes, where each pass traverses only certain classes of edges in the SDG; however, in a forward slice edges are traversed from source to target. The first pass of a forward slice, which we call an *f1* slice, ignores parameter-in, call, and def-order edges; the second pass, or *f2* slice, ignores parameter-out and def-order edges. Thus, the full forward slice of *G* with respect to *S*, denoted by $Induce(f(G, S))$, is defined as $Induce(f2(f1(G, S)))$.

3. A REVISED MODEL FOR MULTI-PROCEDURAL INTEGRATION

In this section we consider two multi-procedure integration algorithms that are straightforward extensions of the HPR algorithm; however, neither algorithm proves to be satisfactory. The first algorithm, discussed in Section 3.1, fails to satisfy Property (2) of the integration model of Section 2.1. The second algorithm, discussed in Section 3.2, does satisfy Property (2) of the integration model, but is unacceptable because it reports interference in cases where there is an intuitively acceptable integrated program. The latter example leads us to reformulate the integration model to better capture the goals of multi-procedure integration; the reformulated model is discussed in Section 3.3.

3.1. Integration of Separate Procedures

Our first candidate algorithm for multi-procedure integration applies the HPR algorithm separately to each procedure (*i.e.*, for each procedure *P* in one or more of *Base*, *A*, or *B*, variant *A*'s and variant *B*'s versions of *P* are integrated with respect to *Base*'s version of *P*). Unfortunately, this approach fails to satisfy Property (2) of the integration model of Section 2.1: it is possible for a merged program produced by this algorithm to terminate abnormally on an initial state on which *A*, *B*, and *Base* all terminate normally. This is illustrated by the example shown in Figure 4, where the program labeled "Integrated System" is the result when the HPR algorithm is used to integrate the three versions of *Main* and the three versions of *P*. Although programs *Base*, *A*, and *B* in Figure 4 all terminate normally, the integrated program terminates

abnormally with a division-by-zero error; this is the result of an interaction between the statement "$a := a - 1$" in procedure *P* (which was introduced in variant *A*) and the statement "$y := 1 / x$" in the *Main* procedure (which was introduced in variant *B*). This example motivates the need for a multi-procedure integration algorithm to take into account potential interactions between modifications that occur in different procedures of the different variants.

<< Insert Figure 4 >>

3.2. Direct-Extension Algorithm: A Straightforward Extension of the HPR Algorithm

The need to determine the potential effects of a change made in one procedure on components of other procedures suggests the use of interprocedural slicing. Thus, our second candidate algorithm for multi-procedure integration is a direct extension of the HPR algorithm: it performs the steps of the HPR algorithm exactly as given in Section 2.1.3, except that each in*tra*procedural slicing operation is reinterpreted as an in*ter*procedural slicing operation.

Although this reinterpretation does yield a multi-procedure integration algorithm that satisfies the integration model of Section 2.1, the algorithm obtained is unsatisfactory because it fails (*i.e.*, reports interference) on many examples for which, intuitively, integration should succeed. This is illustrated by the example shown in Figure 5, on which the direct extension of the HPR algorithm reports interference. Because the backward slices b(*Base*, { "$x := x + 1$" }), b(*A*, { "$x := x + 1$" }), and b(*B*, { "$x := x + 1$" }) are pairwise unequal, the statement "$x := x + 1$" is an affected point in both variants; therefore, the slices b(*A*, { "$x := x + 1$" }) and b(*B*, { "$x := x + 1$" }) are both included in the merged SDG G_M. However, because *both* slices are included in G_M, they are both "corrupted" in G_M. For example, the assignment statement "$b := 4$" is in the backward slice of G_B with respect to "$x := x + 1$", and therefore is in G_M. In G_A, the actual-in vertex associated with the second call to *Incr* is in the backward slice with respect to affected point "$x := x + 1$" and therefore is in Δ(*A*, *Base*). This actual-in vertex is also a directly affected point of G_M with respect to *A*; it has an incoming edge from "$b := 4$" in G_M but there is no such edge in G_A. When a vertex is in both Δ(*A*, *Base*) and *DAP* (G_M, *A*), it means that there is some affected point of *A* (in this case, "$x := x + 1$"), whose changed behavior may not be preserved in the merged program, and the integration algorithm reports Type I interference.

<< Insert Figure 5 >>

Further examination of the example in Figure 5 reveals that extending the programming language with procedures and call statements has introduced a new issue for the proper formulation of integration, namely, "What is the 'granularity' by which one should measure 'changed execution behavior'?" It is certainly true that statement "$x := x + 1$" exhibits three different behaviors (*i.e.*, produces three different

sequences of values) in *Base*, *A*, and *B*; however, statement "$x := x + 1$" in variant *A* exhibits different behavior from *Base* only on the *first* invocation of *Incr*; statement "$x := x + 1$" in *B* exhibits different behavior from *Base* only on the *second* invocation of *Incr*. Thus, for this example, it would seem desirable for the integration algorithm to succeed and return the program labeled "Proposed Integrated System" in Figure 5. When this program is executed, it exhibits the changed behavior from variant *A* during the first invocation of *Incr* and the changed behavior from variant *B* during the second invocation of *Incr*.

However, the program labeled "Proposed Integrated System" in Figure 5 fails to meet Property (2) of the integration model of Section 2.1 (because the sequences of values produced at statement "$x := x + 1$" in *Base*, *A*, and *B* are pairwise unequal). This example suggests that the integration model of Section 2.1, which was originally introduced as a model for the integration of *single-procedure* programs, needs to be revised to characterize better the goals of multi-procedure integration. In particular, the integration model should capture the notion of changed execution behavior at a finer level of granularity.

3.3. "Roll-out" and a Revised Model of Program Integration

In this section we define a more appropriate model of multi-procedure integration by relating multi-procedure integration to single-procedure integration through the concept of *roll-out*—the exhaustive in-line expansion of call statements to produce a program without procedure calls. Each expansion step replaces a call statement with a new *scope* statement that contains the body of the called procedure and is parameterized by assignment statements that make explicit the transfer of values between actual and formal parameters. We use the notation *roll-out*(*S*) to denote the set of (possibly infinite) programs produced by repeatedly expanding call-sites in each procedure of system *S*. In the presence of recursion, roll-out leads to an *infinite* program. (The meaning of an infinite program is defined by the least upper bound of the meanings of the finite programs that approximate it.) We wish to stress that our multi-procedure integration method does not actually *perform* any roll-outs; roll-out is simply a conceptual device introduced to formulate properly our algorithm's goal.

The roll-out concept leads us to a different perspective on the notion of the *behavior* of a program component (and consequently to a notion of finer granularity of changed execution behavior). A "rolled-out" program contains many occurrences—possibly an infinite number—of each statement in the original program. In such a program, different occurrences of a given component from procedure *P* correspond to invocations of *P* in different calling contexts. Consequently, one occurrence of a given component can have a different behavior in *roll-out*(*A*) than in *roll-out*(*Base*), while another occurrence of the same component has a different behavior in *roll-out*(*B*) than in *roll-out*(*Base*), without there being interference.

Example. For the systems listed in Figure 5, the (finite) *Main* programs obtained by applying roll-out to each system (these rolled-out programs are shown in Figure 6) contain two different occurrences of statement "$x := x + 1$", corresponding to the first and second invocations of procedure *Incr*. The change made in

variant *A* affects the behavior only at the first occurrence of "$x := x + 1$"; while the change made in variant *B* affects the behavior only at the second occurrence of "$x := x + 1$". When roll-out is applied to the system labeled "Proposed Integrated System" in Figure 5, the resulting program captures both changes: when the program is executed, it exhibits the changed behavior from variant *A* at the first occurrence of "$x := x + 1$" as well as the changed behavior from variant *B* at the second occurrence of "$x := x + 1$."

<< Insert Figure 6 >>

Our revised model of integration involves the concept of "integration" at two levels:

(1) The *conceptual* level concerns the integration of rolled-out (and hence possibly infinite) programs. In what follows, I^{∞} denotes an operation that combines three rolled-out programs.

(2) The *concrete* level concerns an actual *algorithm* for multi-procedure integration (*i.e.*, an operation that applies to finite representations of programs, such as SDGs). In what follows, I^{S} denotes an algorithm that combines three programs.

Operation I^{∞} provides a standard against which we measure I^{S}.

We can now state the revised, two-level model of integration for programs with procedures:

Revised Model of Program Integration

(1) At the conceptual level, we modify the requirements of the integration model of Section 2.1 to cover sets of rolled-out procedures: let $M^{\infty} = I^{\infty}(\textit{roll-out}(\textit{Base}), \textit{roll-out}(A), \textit{roll-out}(B))$. Then for every (rolled-out) procedure that exists in one or more of *roll-out*(*Base*), *roll-out*(*A*), *roll-out*(*B*), and M^{∞}, the three properties of the integration model of Section 2.1 (with infinite programs now permitted) must hold.

(2) At the concrete level, we impose the following conditions:

(i) Programs are finite and are written in the simplified programming language described in Property (1) of the integration model of Section 2.1, extended with procedures, call statements, and value-result parameter passing.

(ii) When I^S (applied to *Base*, *A*, and *B*) succeeds—producing system *M*—then I^{∞} (applied to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*)) succeeds—producing set of programs M^{∞}—and for every initial state σ on which *Base*, *A*, and *B* terminate, the following must hold for the execution of M^{∞}, *roll-out*(*M*), *roll-out*(*A*), and *roll-out*(*B*) on σ:

(a) For every program point $p_{M^{\infty}}$ in M^{∞}, there exists a component of *roll-out*(*M*) that produces the same sequence of values as $p_{M^{\infty}}$.

(b) For every program point $p_{\textit{roll-out}(M)}$ in *roll-out*(*M*), there exists a component in *roll-out*(*A*), *roll-out*(*B*), or both that produces the same sequence of values as $p_{\textit{roll-out}(M)}$.

(iii) Program *M* is created only from components that occur in programs *Base*, *A*, and *B*.

Property (2.ii) states that the system produced by algorithm I^S must be consistent with the (possibly infinite) program produced by operation I^{∞}.

The use of rolled-out programs in the model has the effect of weakening Property (2) of the integration model of Section 2.1, since the new model directly relates the computations of only the rolled-out programs. In a rolled-out program, different occurrences of a given component of some procedure *P* correspond to different invocations of *P* in different calling contexts. Thus, there is not necessarily interference when one occurrence of a given component has a different behavior in *roll-out*(*A*) than in *roll-out*(*Base*), while another occurrence of the component has a different behavior in *roll-out*(*B*) than in *roll-out*(*Base*).

As with the original integration model of Section 2.1, the questions of whether there exists an M^{∞} that satisfies Property (1) and whether there exists an *M* that satisfies Property (2.ii) are undecidable (even under the restrictions given by Properties (2.i) and (2.iii)). Consequently, we must use techniques that provide safe approximations to these properties, and so our program-integration algorithm will sometime

report interference—and hence fail to produce an integrated program—in cases when there is *some* program that meets the conditions given above.

Note that our model of program integration can have many different "instantiations." The model relates an operation I^{∞} that operates on rolled-out (and potentially infinite) programs, to an operation I^{S} that operates on finite program representations. Consequently, a particular instantiation of the model involves defining two operations, say $\overline{\mathbf{I}}^{\infty}$ and $\overline{\mathbf{I}}^{S}$, although only $\overline{\mathbf{I}}^{S}$ is of practical significance for the management of actual program variants.

The remainder of this paper concerns one particular instantiation of the program-integration model. The operation that plays the role of I^{∞} is called *Integrate*$^{\infty}$ and is defined as follows. Let *HPR*$^{\infty}$ be the natural extension of the HPR algorithm that operates on the infinite PDGs of infinite programs. Let *roll-out*$(A)|_P$ denote procedure P of *roll-out*(A). *Integrate*$^{\infty}$ is the operation that applies *HPR*$^{\infty}$ to every (rolled-out) procedure in one or more of *roll-out*(*Base*), *roll-out*(A), and *roll-out*(B):

$$\textit{Integrate}^{\infty}(\textit{roll-out}(\textit{Base}), \textit{roll-out}(A), \textit{roll-out}(B)) =_{df} \bigcup_{P \in (\textit{Base} \cup A \cup B)} \textit{HPR}^{\infty}(\textit{roll-out}(\textit{Base})|_P, \textit{roll-out}(A)|_P, \textit{roll-out}(B)|_P).$$

The operation that plays the role of I^{S} is called *Integrate*^{S} and is defined in Section 4. We call the pair of operations (*Integrate*$^{\infty}$, *Integrate*^{S}) the "Standard Instantiation (of the Revised Model of Program Integration)."

A few words about the Standard Instantiation are in order at this point:

(i) The pair (*Integrate*$^{\infty}$, *Integrate*^{S}) satisfies the Revised Model of Program Integration [3]. The proof follows from three results outlined below. First, the integration and slicing theorems for single-procedure programs extend to sets of infinite programs. Thus, M^{∞} satisfies Part 1 of the model. Second, starting with *Base*, *A*, and *B*, the set of programs obtained by applying roll-out and then *Integrate*$^{\infty}$ is a slice of the set of programs obtained by applying *Integrate*^{S} and then roll-out (see Figure 7). Thus, *roll-out*(M) satisfies Part 2.ii.a of the model. Third, *roll-out*(M) is the union of uncorrupted slices from *roll-out*(A) and *roll-out*(B) (*i.e.*, for each vertex v in *roll-out*(M), the slice of *roll-out*(M) with respect to v is equal to the slice of *roll-out*(A) with respect to v or the slice of *roll-out*(B) with respect to v). Thus, M satisfies Part 2.ii.b of the model.

<< Insert Figure 7 >>

(ii) *Integrate*^{S} generalizes the HPR algorithm in the following way: when the HPR algorithm and *Integrate*^{S} are applied to the same collection of single-procedure programs, either both report interference, or both produce the same integrated program. This follows from the idempotence of b^{hpr} and f^{hpr}, and the fact that on single-procedure programs, the operations *b1* and *b2* degenerate to b^{hpr}, and the operations *f1* and *f2* degenerate to f^{hpr}.

4. AN ALGORITHM FOR MULTI-PROCEDURE INTEGRATION

This section presents *Integrate*^{S}, an algorithm for multi-procedure program integration that meets the requirements of the Revised Model of Program Integration presented in Section 3. There are two aspects of the multi-procedure integration problem that complicate the definition of *Integrate*^{S}:

(1) *Integrate*^{S} must take into account different calling contexts when determining what was changed in variants *A* and *B*. (For instance, the candidate algorithm proposed in Section 3.1.2 reports interference, and hence fails to produce the system labeled "Proposed Integrated System" for the example in Figure 5 because it does not take into account different calling contexts.) In our work on interprocedural slicing [8], a somewhat similar calling-context problems is handled by exploiting the SDGs summary edges and by using two separate traversals over the SDG. For multi-procedure integration, the calling-context problem is handled by again exploiting the summary edges, this time in conjunction with a number of backward and forward slicing operations.

(2) One or more of the arguments to *Integrate*^{S} can contain dead code (that is, code whose execution has no effect on the program's output). Note that the model of program integration given in Section 2.1 (as well as the revised model for multi-procedure integration) requires that every program component in variant *A* or *B* whose behavior differs from the corresponding component of *Base* be included in the integrated program *M* (regardless of whether that component's changed behavior also causes the program's output to change). This is important for several reasons. One reason is to ensure that semantics-preserving changes (which by definition cause no change to the program's output) introduced into variant *A* or *B* are included in the integrated program. A second reason is that, during program development, dead code may be introduced temporarily, with the programmer intending to turn it into live code at a later time. It is an important practical concern that an integration algorithm preserve *all* changes made to a variant, including the introduction of, and modifications to, dead code.

Including semantics-preserving changes in an integrated multi-procedure program is not a problem; however, to properly handle systems that contain dead code, the first step of the multi-procedure integration algorithm augments the SDGs of *A* and *B* with some additional vertices that act as representatives for dead code, and additional edges that summarize paths to dead code (see Section 4.2). This information is used by the integration algorithm to identify changed behaviors in variants *A* and *B* even in the presence of dead code.

The presentation of the multi-procedure integration algorithm is divided into three subsections: Section 4.1 describes the basic algorithm, temporarily ignoring the problem of systems that contain dead code. Section 4.2 describes the modifications needed to handle dead code. Section 4.3 contains a recap of the multi-procedure integration algorithm *Integrate*^{S} and discusses the complexity of the algorithm.

4.1. The Basic Multi-Procedure Integration Algorithm

As in the HPR algorithm, multi-procedure integration involves 4 steps: Step 1 identifies slices that represent changed behaviors ($\Delta^S(A, Base)$ and $\Delta^S(B, Base)$); Step 2 identifies slices that represent preserved behaviors ($Pre^S(A, Base, B)$); Step 3 combines these slices to form the merged SDG; and Step 4 tests for interference.

4.1.1. Step 1: Determining changed slices

Directly affected points **(**DAP^S**)**

As in the HPR algorithm, the *directly affected points* of *A* and *B* (with respect to *Base*) are used to identify the *affected points*, which are in turn used to identify the slices of *A* and *B* that represent their changed behaviors.

Recall that in the case of *single* procedure integration (the HPR algorithm) $DAP^{hpr}(A, Base)$ consists of the vertices of *A* that are not in *Base* and those whose incoming edges are different in *A* and *Base*. For the purposes of identifying directly affected points in SDGs, it is crucial to *exclude* differences in parameter-in and call edges from the definition of $DAP^S(A, Base)$, and it is convenient also to exclude differences in parameter-out and summary edges. Thus, the definition of directly affected points is essentially the same for multi-procedure and for single-procedure integration:

$$
\begin{aligned}
&DAP^S(A, Base) =_{df} \\
&\quad (A, \{\, v \in V(A) \mid \quad v \notin V(Base) \\
&\qquad\qquad \vee\ IncomingControl^S(A, v) \neq IncomingControl^S(Base, v) \\
&\qquad\qquad \vee\ IncomingLoopCarriedFlow^S(A, v) \neq IncomingLoopCarriedFlow^S(Base, v) \\
&\qquad\qquad \vee\ IncomingLoopIndependentFlow^S(A, v) \neq IncomingLoopIndependentFlow^S(Base, v) \\
&\qquad\qquad \vee\ IncomingDefOrder^S(A, v) \neq IncomingDefOrder^S(Base, v)\ \}).
\end{aligned}
$$

The reason for excluding differences in parameter-in and call edges from the definition of $DAP^S(A, Base)$ has to do with the construction of $\Delta^S(A, Base)$ from $DAP^S(A, Base)$. This construction (discussed shortly) involves *b* (*i.e.*, "full backward") slices with respect to each directly affected point *v* of *A*.[12] The use of *b* slices includes *all* of *v*'s possible calling contexts in $\Delta^S(A, Base)$. Thus, if changes in the incoming parameter-in or call edges were to cause formal-in and entry vertices, respectively, in procedure *P* to be classified as directly affected points, then *all* call-sites on *P* (not just the one containing the change) would be part of $\Delta^S(A, Base)$.

In particular, this problem occurs when *A* adds or deletes a call on *P* or replaces an actual parameter at an existing call-site on *P*, and *B* adds or deletes a call on *P* or changes the value of an actual parameter at

———————————————————

[12] The definition of $\Delta^S(A, Base)$ also involves *b* and *b2* slices taken with respect to some other vertices.

an existing call-site on *P*. For example, Figure 8 illustrates the case where *A* deletes a call-site on procedure *P*, and *B* changes the value of an actual parameter at a different call-site on *P*. (For this example, the goal is to produce the system labeled "Proposed M".) In *A*, the deletion of call statement "**call** $P(a)$" changes the incoming parameter-in edges of the formal-in vertex for *x*, and also changes the incoming call edges of the entry vertex $Enter_P$. We do not want this change to cause the actual-in and call-site vertices of the remaining call-site on *P* to be incorporated into $\Delta^S(A, Base)$. If they are, then "$b := 0$" will also be included in $\Delta^S(A, Base)$ and there would be a conflict with the change made in variant *B*.

<< Insert Figure 8 >>

It is not crucial to the correctness of $Integrate^S$ to exclude differences in parameter-out and summary edges from the definition of $DAP^S(A, Base)$: doing so does not affect the success of $Integrate^S$ nor does it change the merged program produced by $Integrate^S$. It does, however, simplify the argument, given in [3], that $Integrate^S$ satisfies the Revised Model of Program Integration.

As discussed in Section 2.1.3, determining the directly affected points (as well as the other steps of $Integrate^{hpr}$) requires program components to be tagged so that corresponding components can be identified in all three versions (*Base*, *A*, and *B*). Section 2.1.3 did not discuss tags for call statements. Because a call statement is represented by multiple vertices in an SDG it has multiple tags: one for each vertex. For example, **call** $P(a)$ has three tags: a tag for the call-site vertex, for *a*'s actual-in vertex, and for *a*'s actual-out vertex. Furthermore, in addition to the conditions imposed on tags in Section 2.1.3, the tags on parameter vertices (for both actual and formal parameters) must encode the position of the parameter in the parameter list.

To understand why this encoding is necessary, consider an edit that interchanges two actual parameters. This interchange is likely to change the computation carried out by the system and therefore should be captured in Δ^S. At the same time, what is captured in Δ^S should be limited to those calling-contexts that contain the call-site where the interchange took place (not all calling-contexts of the called procedure). We accomplish this by including the parameter-list position of actual parameters as part of the tags of their actual-in (and actual-out) vertices; thus, interchanging two parameters creates directly affected points only at the call-site where the interchange took place. (A similar encoding is performed for formal-in and formal-out vertices.)

Affected points **(**AP^S**)**

As in the HPR algorithm, the affected points of *A* and *B* with respect to *Base* are defined to be the vertices that are in the forward slices of *A* and *B* with respect to their directly affected points:

$$AP^S(A, Base) =_{df} f(DAP^S(A, Base)).$$

Changed slices **(Δ^S)**

In the HPR algorithm, $\Delta^{hpr}(A, Base)$ is defined to be the backward slice of A with respect to its affected points. For multi-procedure integration it is necessary to partition the affected points into two subsets: the *strongly affected points* and the *weakly affected points*. A strongly affected point potentially exhibits changed behavior in *all* calling contexts while a weakly affected point only potentially exhibits changed behavior in *some* calling contexts. The strongly affected points of A with respect to *Base*, denoted by $SAP^S(A, Base)$, are those vertices of A that are in the *f1* slice with respect to a directly affected point:

$$SAP^S(A, Base) =_{df} f1(DAP^S(A, Base)).$$

The affected points that are not strongly affected are the weakly affected points. Strongly and weakly affected points contribute differently to the definition of Δ^S; thus, there are two parts to the definition of Δ^S. The first contains the contribution of the strongly affected points. Because the execution behavior at each strongly affected point v is potentially modified in *every* calling context in which it is executed, it is necessary to incorporate *all* of v's possible calling contexts in Δ^S. This is accomplished by taking a *b* (*i.e.*, "full backward") slice with respect to v.

Example. Variant A shown in Figure 9 contains two directly affected points: the statement "$t := 2$" and the actual-in vertex for t at the second call-site on Q in P (which has different incoming edges in A and *Base*); these two points are also the only strongly affected points. The *b* slice of A taken with respect to these points includes all of the calling contexts for P (the two call-sites on P in this example). This part of $\Delta^S(A, Base)$ is shown in the third column of Figure 9.

<< Insert Figure 9 >>

The second part of the definition of Δ^S contains the contribution of the weakly affected points. Because the execution behavior of each weakly affected point v is potentially modified only in *some* calling contexts in which it is executed, it is necessary to incorporate only *some* of v's possible calling contexts in Δ^S. This can be done using *b2* slices with respect to weakly affected points despite the fact that a *b2* slice includes none of a procedure's calling contexts. Suppose v is a weakly affected point in procedure P. A *b2* slice with respect to v includes only vertices from P and procedures called by P; it does not include any vertices from procedures that call P. Initially this seems incorrect because for a weakly affected point *some* calling context must have changed. However, at least one of the call-site, actual-in, or actual-out vertices associated with the changed calling context must also be an affected point; thus, any changed calling context for P in A with respect to *Base* will also be included in one of the two parts of $\Delta^S(A, Base)$, as desired.

Because $b2(SAP^S(A, Base)) \subseteq b(SAP^S(A, Base))$, the second part of Δ^S can be defined to be the *b2* slice taken with respect to all affected points rather than only the weakly affected points.

Example. Variant *A* shown in Figure 9 contains two weakly affected points: the formal-in vertex for *z* in procedure *Q* and the vertex labeled "*t2* := *t3*+*z*." The *b2* slice with respect these vertices includes *Q*'s entry vertex and the vertex labeled "*t3* := 1". (See the fourth column of Figure 9.) It does not include any of *Q*'s affected calling contexts (those that contain the second call-site on *Q* in *P*). However, the vertices at the second call-site on *Q* are also affected points and are therefore correctly included in $\Delta^S(A, Base)$ (shown in the fifth column of Figure 9).

Putting the two parts of Δ^S together, we define

$$\Delta^S(A, Base) =_{df} b(SAP^S(A, Base)) \cup b2(AP^S(A, Base)).$$

Example. In Figure 5, $\Delta^S(A, Base)$ does not contain the second call-site on *Incr* in variant *A*. Thus, *B*'s change can affect this call-site (as it does in Figure 5) without causing interference.

4.1.2. Step 2: Determining preserved slices

Intuitively, a vertex *v* in procedure *Q* should be in Pre^S only if both of the following hold:

(1) Vertex *v* is in *Base*, *A*, and *B*.

(2) If the versions of *Q* in *Base*, *A*, and *B* are called with the same argument values, the program component represented by *v* produces the same sequence of values in all three versions.

This intuitive definition of Pre^S requires that *v* be in Pre^S even when there is *no* calling context in *Base*, *A*, and *B* such that *Q* is called with the same argument values in all three programs. Thus, the definition of Pre^S should ignore the contexts in which *Q* is called, while still correctly accounting for calling context in the calls made (transitively) from *Q*. This is accomplished using *b2* slices; a vertex with the same *b2* slice exhibits the same behavior when the procedure containing the vertex is called with the same initial argument values in *Base*, *A*, and *B* [3]. Thus, we define Pre^S as follows:

$$Pre^S(Base, A, B) =_{df} (Base, \{ v \in V(Base) \mid b2(A, v) = b2(Base, v) = b2(B, v) \}).$$

4.1.3. Step 3: Forming the merged graph

The merged graph is formed as follows:

$$G_M = \Delta^S(A, Base) \cup \Delta^S(B, Base) \cup Pre^S(Base, A, B).$$

In [3] it is shown that this construction is a true generalization of the one used in the HPR algorithm; that is, for systems that consist of only a single procedure (*i.e.*, with no calls to auxiliary procedures), G_M is identical to the merged graph constructed by the HPR algorithm.

4.1.4. Step 4: Testing for interference

As in the HPR algorithm, the merged graph must be tested for both Type I interference (was any changed slice of a variant "corrupted" when the merged graph was formed?) and Type II interference (does the

merged graph correspond to some program?). In addition, there is an additional interference test (for Type III interference) that has no counterpart in the HPR algorithm. Note that we explicitly use G_M in the definitions of the Type I and Type II interference tests to emphasize that a system M whose SDG is G_M exists only if there is no Type II interference. Even if M exists, it may be semantically unsuitable (*i.e.* there may be Type I or Type III interference).

Type I Interference

The test for Type I interference is a straightforward extension of the corresponding test in the HPR algorithm; there is Type I interference if a vertex in $\Delta^S(A, Base)$ is also a directly affected point of G_M with respect to A (or if the analogous condition holds for B):

$$\Delta^S(A, Base) \cap DAP^S(G_M, A) \neq \emptyset \qquad \text{or} \qquad \Delta^S(B, Base) \cap DAP^S(G_M, B) \neq \emptyset.$$

Type II interference

The next step of *Integrate*^{S} is to determine whether the merged graph G_M is feasible (*i.e.*, corresponds to some system), and if it is, to return a system whose SDG is G_M. There are two possible reasons why this may not be possible (in both cases Type II interference is reported). First, G_M may be PDG-infeasible (*i.e.*, one of the PDGs may be infeasible). Second, even if all of the SDG's PDGs are feasible, G_M may be SDG-infeasible (*i.e.*, G_M may still be infeasible due to procedure linkage constraints).

PDG-infeasibility is tested for by applying the program-reconstitution algorithm from [7] to each PDG in G_M (a few straightforward modifications to the reconstitution algorithm are needed to accommodate the additional kinds of vertices that represent call statements). The program-reconstitution algorithm may determine that a PDG is infeasible (and thus that G_M is infeasible); in this case, there is Type II interference and *Integrate*^{S} reports failure.

Assuming each PDG in G_M is PDG-feasible, SDG-feasibility is tested for in two steps. The first checks that the actual-in and actual-out vertices for each call-site in G_M on procedure P match up with the formal-in and formal-out vertices of P's PDG. The second step is to form a system from the reconstituted PDGs of G_M. If G_M is this system's SDG then G_M is SDG-feasible.

Type III Interference

Informally, Type III interference exists when a component c is added to a procedure P in variant A, and a (transitive) call on procedure P is added in variant B. This situation would produce, in M, an execution of component c in a calling context that exists in B but not in A. We want *Integrate*^{S} to fail in such cases because M would not satisfy Property (2.ii) of the Revised Model of Program Integration, which requires that M compute only sequences of values computed by A, B, or both.

The Type III interference test uses the Δ^S operator; however, unlike other uses of Δ^S in algorithm *Integrate*^{S}, the Type III interference test applies Δ^S to M and A. Intuitively, $\Delta^S(M, A)$ includes the components of M necessary to capture the computations of M that are different from A. In order to satisfy the Revised Model of Program Integration, these computations must occur in B. If some component from $\Delta^S(M, A)$ is also in $DAP^S(M, B)$ then some computation from M that is not in A is also not in B. Thus, there is Type III interference iff

$$\Delta^S(M, A) \cap DAP^S(M, B) \neq \emptyset.$$

(It can be shown that it is unnecessary to test the symmetric condition $\Delta^S(M, B) \cap DAP^S(M, A) \neq \emptyset$; that is, $\Delta^S(M, A) \cap DAP^S(M, B) \neq \emptyset$ iff $\Delta^S(M, B) \cap DAP^S(M, A) \neq \emptyset$.)

If there is no Type I, Type II, or Type III interference, then a system M whose SDG is G_M is returned as the result of the integration.

4.2. Handling Dead Code

The construction of Δ^S described above is not quite correct because, although it handles the calling-context problem, it does not correctly handle systems with dead code. We now describe how to overcome this limitation. To handle dead code correctly, it is necessary to add some additional vertices and edges to the SDGs for A and B before computing $\Delta^S(A, Base)$ and $\Delta^S(B, Base)$, respectively. These additional *meeting-point* vertices and edges are used to include the correct actual parameters in the presence of dead code. After the addition of these vertices and edges the computation of Δ^S proceeds exactly as described above (except that the slicing operators *b1*, *b2*, *f1*, and *f2* are extended to also traverse meeting-point edges). First we describe the specific problem meeting-point vertices and edges solve and why this problem exists only in the presence of dead code. Then we present a formal definition of meeting-point vertices and edges. An algorithm for the efficient computation of meeting-point vertices and edges can be found in [2]

The problem meeting-point vertices solve arises when the value of more than one actual parameter affects a "dead" computation in a (transitively) called procedure. When computing Δ^S, if the actual-in vertex of one of these parameters is an affected point then the dead computation is also affected. To incorporate this affected computation, it is necessary to include in Δ^S *all* of the actual parameters that affect the dead computation. This is accomplished by connecting the actual-in vertices for these actual parameters to meeting-point vertices.

Example. For the example shown in Figure 10, if $\Delta^S(A, Base)$ is computed without meeting-point vertices then it includes the *b* slice with respect to the actual-in vertex for parameter "*a*" (a strongly affected point) and the *b2* slice with respect to the vertex representing "*t1* := *x* + *y*" (a weakly affected point). Neither of these slices includes the actual-in vertex for actual parameter "1". However, in this example the

actual parameters "*a*" and "1" at the first call-site on procedure *P* in variant *A* both affect the computation of local variable *t1* in procedure *P*. To cause the actual parameter "1" to be included in Δ^S, both actual-in vertices are connected to a meeting-point vertex *mp*. Because *mp*, like the actual-in vertex for parameter "*a*", is a strongly affected point, the backward slice of *A* taken with respect to *mp* as part of the computation of $\Delta^S(A, Base)$ includes the actual-in vertex for "1".

<< Insert Figure 10 >>

The problem solved by meeting-point vertices and edges exists only in the presence of dead code—in the absence of dead code there is always an actual-out vertex that subsumes the role of the meeting-point vertex. To see that this is the case, suppose that there are two formal parameters whose values both affect the computation represented at vertex *v* (*e.g.*, in Figure 10, *v* is the vertex labeled "*t1* := *x*+*y*" where the values of *x* and *y* are both used in the expression "*x*+*y*"). In the absence of dead code, a path connects *v* to some formal-out vertex (*i.e.*, *v* is not dead). Each actual-out vertex associated with this formal-out vertex is therefore the target of (at least) two summary edges—one from each of the actual-in vertices associated with the formal-in vertices for the two formal parameters. Thus the meeting at *v* is witnessed at each call-site by an actual-out vertex, which obviates the need for a meeting-point vertex.

We now formally define meeting-point vertices and edges by building on the definition for ACF_P, given in Section 2.2.1. Meeting-point vertices can be obtained as the least solution to the set of equations given below. These equations define, for each procedure *P*, the control-flow dependence subgraph of G_P augmented with summary edges and meeting-point vertices and edges, which we denote by $MpACF_P$. (In the equation for *V*, the notation "$<Call_Q.a_{in}, Call_Q.b_{in}>$" denotes a new meeting-point vertex and, in the equation for *E*, the expression "$Call_Q.a_{in} \longrightarrow v$, $Call_Q.b_{in} \longrightarrow v \mid \cdots$" denotes a pair of meeting point edges from actual-in vertices $Call_Q.a_{in}$ and $Call_Q.b_{in}$ to meeting-point vertex *v*.)

DEFINITION. (Meeting-Point Vertices).

$MpACF_P = ACF_P \cup (V, E)$, where
$V = \{ <Call_Q.a_{in}, Call_Q.b_{in}> \mid Call_Q \in V(G_P) \wedge Call_Q.a_{in} \neq Call_Q.b_{in} \}$
$E = \{ Call_Q.a_{in} \longrightarrow v, Call_Q.b_{in} \longrightarrow v \mid$
$\quad Call_Q \in V(G_P)$
$\quad \wedge\ v \in V \wedge v = <Call_Q.a_{in}, Call_Q.b_{in}>$
$\quad \wedge\ \exists x \in V(MpACF_Q)\ s.t.\ \{ Enter_Q.a_{in} \longrightarrow x, Enter_Q.b_{in} \longrightarrow x \} \subseteq (E(MpACF_Q))^+$
$\quad \wedge\ \nexists u \in V(ACF_P)\ s.t.\ \{ Call_Q.a_{in} \longrightarrow_s u, Call_Q.b_{in} \longrightarrow_s u \} \subseteq E(ACF_P) \}$.

Thus, at each call-site, there is one meeting-point vertex for every pair of actual-in vertices at the call-site. Each meeting-point vertex is the target of either no edges or exactly two edges. These meeting-point edges originate from (different) actual-in vertices at the call-site and represent two paths in the called procedure's PDG that meet at a common vertex (*x* in the definition). The reason for the final clause in the equation for the set of edges *E* is to avoid the addition of meeting-point edges if there is an actual-out vertex and

appropriate summary edges that subsume the need for a meeting-point vertex and its edges. This clause is included to allow an optimization in the algorithm for computing meeting-point vertices and edges: the algorithm need not materialize those meeting-point vertices and edges that are subsumed by actual-out vertices and summary edges.

Example. Figure 10 illustrates the addition of meeting-point vertices and edges. In the PDG for *P* of variant *A*, paths connect the formal-in vertices labeled "$x := x_{in}$" and "$y := y_{in}$" to the vertex labeled "$t1 := x + y$." Consequently, meeting-point vertices and edges are added to the SDG for variant *A* at both call-sites on *P*. As noted above, when computing $\Delta^S(A, Base)$, the vertex labeled "$a := 1$" is directly affected, hence the vertex labeled "meeting point" is strongly affected, and the backward slice with respect to this vertex (taken when computing $\Delta^S(A, Base)$) includes both of the actual-in vertices associated with the first call on *P* (in particular the one labeled "$y_{in} := 1$"). Figure 11 shows the merged system created when $Integrate^S$ is applied to the three systems shown in Figure 10 when meeting-point vertices are used to compute $\Delta^S(A, Base)$ and $\Delta^S(B, Base)$.

<< Insert Figure 11 >>

After the addition of meeting-point vertices and edges the construction of $\Delta^S(A, Base)$ proceeds exactly as described in Section 4.1 (except that the slicing operators *b1*, *b2*, *f1*, and *f2* are each extended to traverse meeting-point edges).

4.3. Recap of $Integrate^S$

Putting all the pieces together, a complete algorithm for multi-procedure integration appears in Figure 12. This algorithm has been implemented in a prototype system, called the Wisconsin Program Integration System [16, 17]. This system can be obtained by contacting the second or third authors. It is being distributed under license by the Computer Sciences Department at the University of Wisconsin–Madison. The system is written in C and SSL (the specification language of the Synthesizer Generator) and runs under UNIX on a variety of workstations. Further information about configuration requirements is available on request.

The cost of integration is the sum of the costs of building SDGs for *Base*, *A*, and *B*, and the cost of applying $Integrate_S$. These costs can be expressed in terms of the following parameters:

<< Insert Figure 12 >>

P	The number of procedures in the system.
CE	The maximum number of edges in any procedure's control-flow graph.
Assign	The maximum number of assignment statements in any procedure.
Params	The maximum number of formal-in vertices in any procedure's PDG. (This is bounded by the maximum sum of the number of formal parameters in a procedure and the number of global variables accessed—directly or indirectly—by the procedure.)
Sites	The maximum number of call-sites in any procedure.
TotalSites	The total number of call sites in the system. (This is bounded by $P \times Sites$.)
E	The maximum number of edges in any procedure's PDG.

The cost of building the PDGs for a system's procedures is bounded by $O(P \times CE \times Assign)$. The cost of adding call, parameter-in, and parameter-out edges is bounded by $O(TotalSites \times Params)$. The cost of adding summary edges is bounded by $O((E \times TotalSites \times Params) + (TotalSites \times Params^3))$ [18]. Thus, the total cost of building the SDG is bounded by $O((P \times CE \times Assign) + (E \times TotalSites \times Params) + (TotalSites \times Params^3))$.

The cost of applying $Integrate^S$ is the sum of the costs of adding meeting-point vertices and edges, constructing G_M, testing for interference, and reconstituting a system from G_M. With one exception, each phase of $Integrate^S$ has complexity polynomial in the sum of the sizes of *Base*, *A*, and *B*. The exception is the reconstitution of a procedure from a PDG. This is in general an NP-complete problem; however, in practice we do not expect this phase to exhibit exponential worst-case behavior [7], and so we expect that the cost of $Integrate^S$ will be polynomial in the sizes of *Base*, *A*, and *B*.

The cost of adding meeting point vertices and edges is bounded by $O(P \times E \times TotalSites \times Params^4)$. The construction of G_M is linear in the size of the largest SDG. For example, Δ^S can be computed in four passes: expanding the right-hand side of the definition of Δ^S by the definitions of *b*, *f*, SAP^S, and AP^S yields

$$\Delta^S(A, Base) = b2(b1(f1(DAP^S(A, Base)))) \cup b2(f2(f1(DAP^S(A, Base)))).$$

Only four passes are required because for both terms the initial pass is $f1(DAP^S(A, Base))$ and the union of $b1(f1((DAP^S(A, Base)))$ and $f2(f1(DAP^S(A, Base)))$ can be performed before a final *b2* pass is made.

The tests for Type I and Type III interference are linear in the size of G_M. The Type II interference test involves reconstituting a procedure from each PDG in G_M, which is an NP-complete problem. The difficulty is ordering multiple definitions to the same variable when there are no dependence edges that force a particular order. Although this can require considering all possible orderings, it is expected that in practice there will be few unforced choices, and thus a simple backtracking algorithm will suffice. If no backtracking is required, the cost can be stated in terms of the parameters given in the table below (all of these parameters refer to quantities in G_M). In the table, a *span* is the definition of a variable and all the uses reached by the definition, and a *region* is all the control successors of a PDG vertex that have the same label (*i.e.*, *true* or *false*) on their incoming control edge.

Parameters that measure the cost of reconstituting a procedure from a PDG	
N	The maximum number of vertices in a region.
Var	The maximum number of accessed variables in a procedure.
G	The maximum number of spans of which any vertex is a member.
SP	The maximum size of a span.
R	The maximum number of regions in any procedure's PDG.

If no backtracking is required, the cost of reconstituting a single procedure from a PDG is $O(R \times N^2 \times (N^2 + Var \times G^2 \times SP))$ [7]. The cost of reconstituting M is the cost of reconstituting the PDGs in G_M (*i.e.*, $O(P \times (R \times N^2 \times (N^2 + Var \times G^2 \times SP)))$). This together with the $O(P \times E \times TotalSites \times Parms^4)$ cost of adding meeting-point vertices and edges dominate the cost of $Integrate^S$.

ACKNOWLEDGMENTS

We wish to thank Genevieve Rosay and Thomas J. A. Bricker for their role in the implementation of $Integrate^S$ as part of the Wisconsin Program Integration System. In addition, we wish to acknowledge Genevieve Rosay for devising the Type III interference test, which replaces a related, but more complicated test presented in [2].

REFERENCES

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

2. Binkley, D., "Multi-procedure program integration," Ph.D. dissertation and Technical Report TR-1038, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

3. Binkley, D., Horwitz, S., and Reps, T., "The Theory of Multi-Procedure Integration," Technical Report (in preparation), Computer Sciences Department, University of Wisconsin, Madison, WI (1993).

4. Cooper, K.D. and Kennedy, K., "Interprocedural side-effect analysis in linear time," *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, (Atlanta, GA, June 22-24, 1988), *ACM SIGPLAN Notices* **23**(7) pp. 57-66 (July 1988).

5. Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., "Programming environments based on structured editors: The MENTOR experience," pp. 128-140 in *Interactive Programming Environments*, ed.

D. Barstow, E. Sandewall, and H. Shrobe,McGraw-Hill, New York, NY (1984).

6. 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).

7. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

8. Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.* **12**(1) pp. 26-60 (January 1990).

9. 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, NY (1981).

10. Landi, W. and Ryder, B., "Pointer-induced aliasing: A problem classification," pp. 93-103 in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 21-23, 1991), ACM, New York, NY (1991).

11. 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, *Journal of Systems and Software* **5**(2)(May 1985).

12. Ottenstein, K.J. and Ottenstein, L.M., "The program dependence graph in a software development environment," *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. 177-184 (May 1984).

13. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

14. Reps, T. and Yang, W., "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

15. Reps, T. and Bricker, T., "Illustrating interference in interfering versions of programs," *Proceedings of the 2nd International Workshop on Software Configuration Management*, (Princeton, NJ, Oct. 24-27, 1989), *ACM SIGSOFT Software Engineering Notes* **17**(7) pp. 46-55 (November 1989).

16. Reps, T., "Demonstration of a prototype tool for program integration," TR-819, Computer Sciences Department, University of Wisconsin, Madison, WI (January 1989).

17. Reps, T., "The Wisconsin program-integration system reference manual: Release 2.0," Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI (July 1993).

18. Reps, T., Sagiv, M., and Horwitz, S., “Interprocedural dataflow via graph reachability,” Unpublished report, Datalogisk Institut, University of Coopenhagen, Copenhagen, Denmark (April 1994).

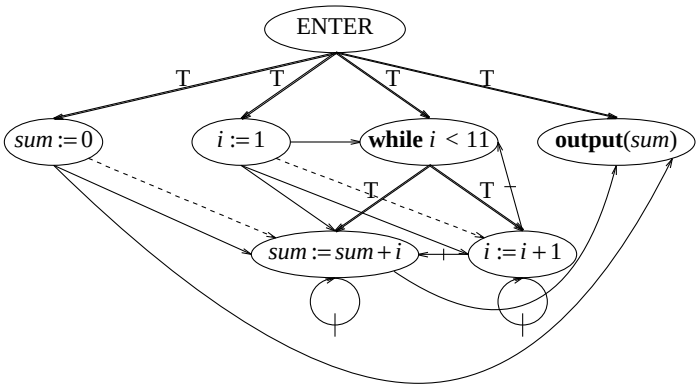
19. Rice, H.G., “Classes of recursively enumerable sets and their decision problems,” *Trans. AMS* **89** pp. 25-59 (1953). (as cited in Hopcroft and Ullman, Introduction to Automata theory, Languages, and Computation, Addison-Wesley, 1979)

20. Sharir, M. and Pnueli, A., “Two approaches to interprocedural data flow analysis,” pp. 189-233 in *Program Flow Analysis: Theory and Applications*, ed. S.S. Muchnick and N.D. Jones,Prentice-Hall, Englewood Cliffs, NJ (1981).

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

22. Yang, W., “A new algorithm for semantics-based program integration,” Ph.D. dissertation and Tech. Rep. TR-962, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1990).

```
program
   sum := 0
   i := 1
   while i < 11 do
      sum := sum + i
      i := i + 1
   od
   output(sum)
end
```



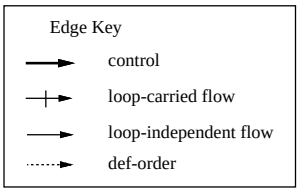


Figure 1. An example program, which sums the integers from 1 to 10, and its PDG.

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

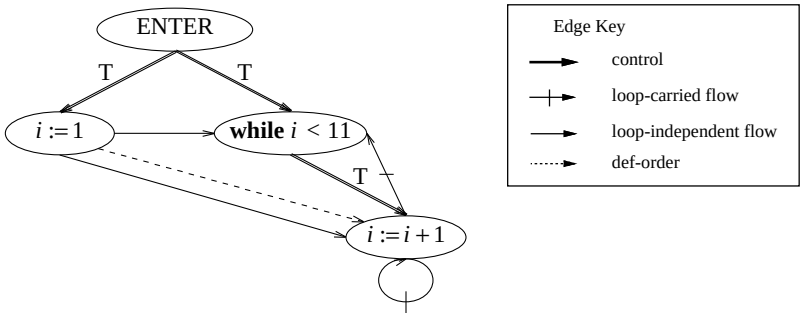


Figure 2. The PDG that results from slicing the example from Figure 1 with respect to the vertex for statement "$i := i + 1$" together with the program to which it corresponds.

```
procedure Main
   sum := 0
   i := 1
   while i < 11 do
       call A(sum, i)
   od
   output(sum)
end

procedure A(x, y)
   call Add(x, y)
   call Increment(y)
return

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

procedure Increment(z)
   call Add(z, 1)
return
```

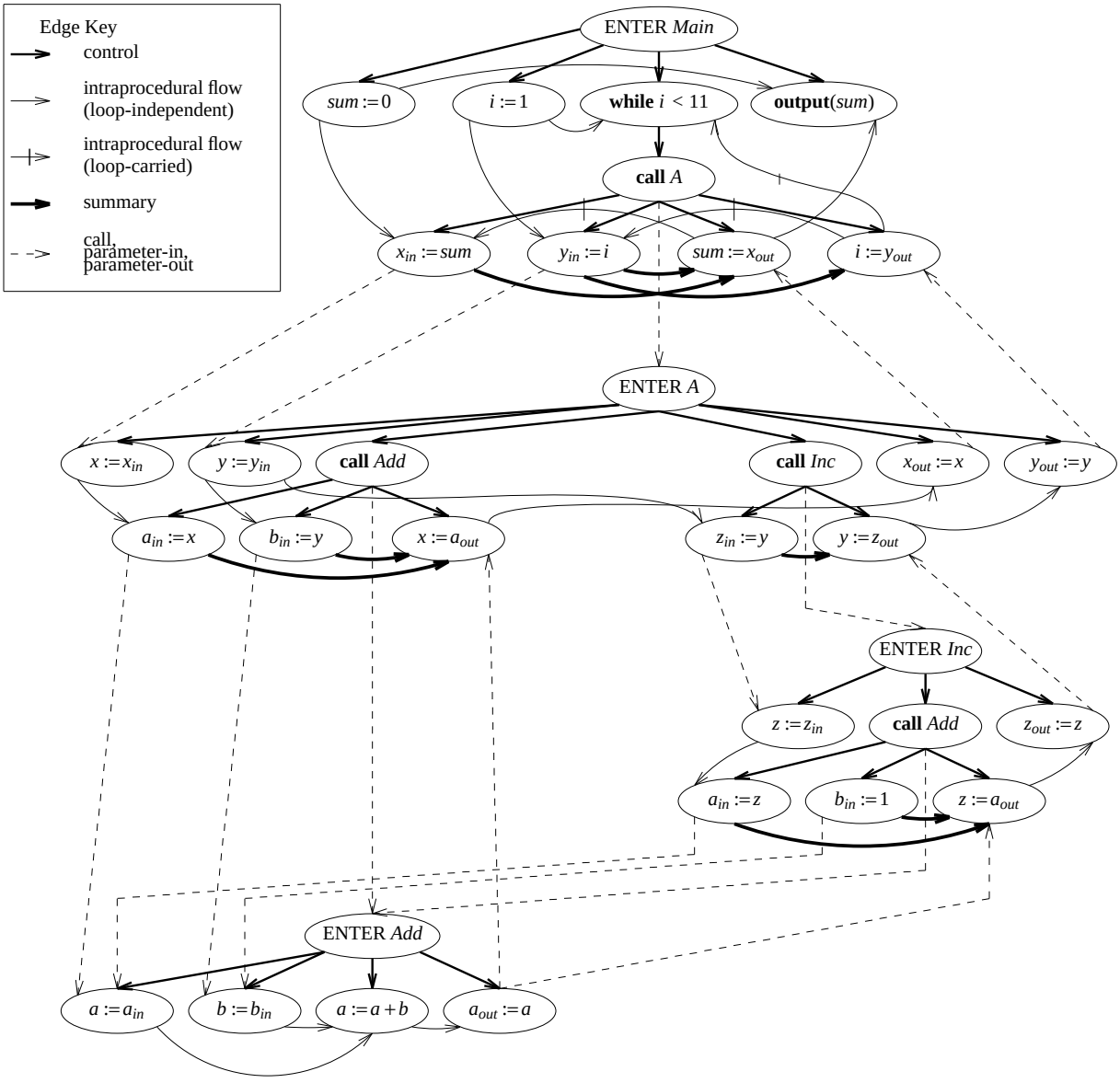


Figure 3. Example system and its SDG.

Base	Variant *A*	Variant *B*	Integrated System
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*
$x := 1$	$x := 1$	$x := 1$	$x := 1$
call $P(x)$	**call** $P(x)$	**call** $P(x)$	**call** $P(x)$
		$\boxed{y := 1 / x}$	$y := 1 / x$
output(x)	**output**(x)	**output**(x)	**output**(x)
end	**end**	**end**	**end**
procedure $P(a)$	**procedure** $P(a)$	**procedure** $P(a)$	**procedure** $P(a)$
$a := a + 1$	$\boxed{a := a - 1}$	$a := a + 1$	$a := a - 1$
return	**return**	**return**	**return**

Figure 4. This example illustrates that when the HPR algorithm is used to integrate separately the individual procedures that make up a system, the integrated program can fail to satisfy Property (2) of the integration model of Section 2.1. (The boxes indicate the modifications made to variants *A* and *B*.)

Base	Variant *A*	Variant *B*	Proposed Integrated System
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*
$a := 1$	$\boxed{a := 3}$	$a := 1$	$a := 3$
$b := 2$	$b := 2$	$\boxed{b := 4}$	$b := 4$
call *Incr* (a)	**call** *Incr* (a)	**call** *Incr* (a)	**call** *Incr* (a)
call *Incr* (b)	**call** *Incr* (b)	**call** *Incr* (b)	**call** *Incr* (b)
output(a)	**output**(a)	**output**(a)	**output**(a)
output(b)	**output**(b)	**output**(b)	**output**(b)
end	**end**	**end**	**end**
procedure *Incr* (x)	**procedure** *Incr* (x)	**procedure** *Incr* (x)	**procedure** *Incr* (x)
$x := x + 1$	$x := x + 1$	$x := x + 1$	$x := x + 1$
return	**return**	**return**	**return**

Figure 5. The program shown on the right incorporates the changed behavior of both *A* and *B* as well as the preserved behavior of *Base*, *A*, and *B*. However, the direct extension of the HPR algorithm reports interference on this example. (The boxes indicate the modifications made to variants *A* and *B*.)

Main program of *roll-out*(*Base*)	*Main* program of *roll-out*(*A*)	*Main* program of *roll-out*(*B*)	*Main* program of *roll-out*(*Proposed Integrated System*)
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*
$a := 1$	$a := 3$	$a := 1$	$a := 3$
$b := 2$	$b := 2$	$b := 4$	$b := 4$
scope($x := a$; $a := x$)	**scope**($x := a$; $a := x$)	**scope**($x := a$; $a := x$)	**scope**($x := a$; $a := x$)
$x := x + 1$	$x := x + 1$	$x := x + 1$	$x := x + 1$
end scope	**end scope**	**end scope**	**end scope**
scope($x := b$; $b := x$)	**scope**($x := b$; $b := x$)	**scope**($x := b$; $b := x$)	**scope**($x := b$; $b := x$)
$x := x + 1$	$x := x + 1$	$x := x + 1$	$x := x + 1$
end scope	**end scope**	**end scope**	**end scope**
output(a)	**output**(a)	**output**(a)	**output**(a)
output(b)	**output**(b)	**output**(b)	**output**(b)
end	**end**	**end**	**end**

Figure 6. The main programs that result from applying roll-out to the four systems shown in Figure 5.

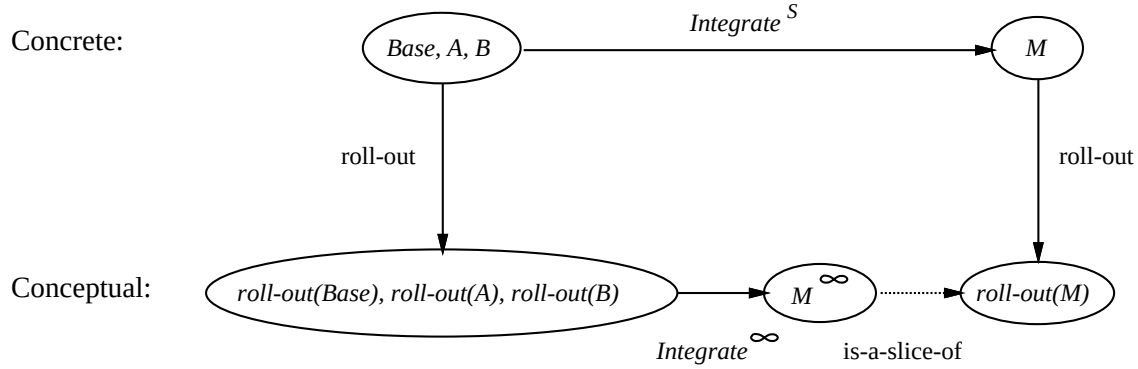


Figure 7. Commutative diagram for the Revised Model of Program Integration.

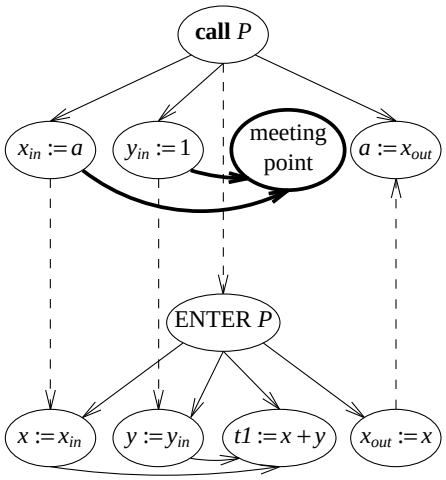
Base	Variant *A*	Variant *B*	Proposed *M*	Common Procedure *P*
procedure *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** *Main*	**procedure** $P(x)$
call $P(a)$		**call** $P(a)$		$x := x + 1$
$b := 0$	$b := 0$	$\boxed{b := 1}$	$b := 1$	**return**
call $P(b)$	**call** $P(b)$	**call** $P(b)$	**call** $P(b)$	
end	**end**	**end**	**end**	

Figure 8. Motivation for the definition of $DAP^S(A, Base)$. (The boxes indicate the modifications made to variants *A* and *B*.)

Base	Variant *A*	$b(SAP^S(A, Base))$	$b2(AP^S(A, Base))$	$\Delta^S(A, Base)$
procedure *Main*	**procedure** *Main*	**procedure** *Main*		**procedure** *Main*
$a := 1$	$a := 1$			
$b := 2$	$b := 2$			
call $P(a)$	**call** $P(a)$	**call** $P()$		**call** $P()$
call $P(b)$	**call** $P(b)$	**call** $P()$		**call** $P()$
output(a)	**output**(a)			
output(b)	**output**(b)			
end	**end**	**end**		**end**
procedure $P(x)$	**procedure** $P(x)$	**procedure** $P()$	**procedure** $P()$	**procedure** $P()$
call $Q(x)$	**call** $Q(x)$			
$t := 1$	$\boxed{t := 2}$	$t := 2$	$t := 2$	$t := 2$
call $Q(t)$	**call** $Q(t)$	**call** $Q(t)$	**call** $Q(t)$	**call** $Q(t)$
$x := 2$	$x := 2$			
return	**return**	**return**	**return**	**return**
procedure $Q(z)$	**procedure** $Q(z)$		**procedure** $Q(z)$	**procedure** $Q(z)$
$t3 := 1$	$t3 := 1$		$t3 := 1$	$t3 := 1$
$t2 := t3+z$	$t2 := t3+z$		$t2 := t3+z$	$t2 := t3+z$
return	**return**		**return**	**return**

Figure 9. The third and fourth columns of this figure show the two parts of $\Delta^S(A, Base)$ computed from systems *Base* and *A* shown in the first two columns. The union of these two parts (really their SDGs) yields a program (SDG) that captures the changed computations of *A* with respect to *Base*. This union is shown in the rightmost column. (The box indicates the modification made to variant *A*.)

Base	Variant *A*	Variant *B*	*Main*'s first call on *P* in Variant *A*

procedure *Main*
  $a := 2$
  $b := 1$
  call $P(a, 1)$
  call $P(b, 1)$
  output(a)
end
	procedure *Main*
	  $\boxed{a := 1}$
	  $b := 1$
	  call $P(a, 1)$
	  call $P(b, 1)$
	  output(a)
	end
		procedure *Main*
		  $a := 2$
		  $b := 1$
		  call $P(a, 1)$
		  call $P(b, \boxed{2})$
		  output(a)
		end
			

procedure $P(x, y)$
  $t1 := x + y$
  call $Incr(x)$
 	
  $x := 2$
return
	procedure $P(x, y)$
	  $t1 := x + y$
	  call $Incr(x)$
	  $\boxed{t3 := x}$
	  $x := 2$
	return
		procedure $P(x, y)$
		  $t1 := x + y$
		  $\boxed{}$
		
		  $x := 2$
		return	

procedure $Incr(z)$
  $t2 := 1/z$
  $z := z + 1$
return
	procedure $Incr(z)$
	  $t2 := 1/z$
	  $z := z + 1$
	return
		procedure $Incr(z)$
		  $t2 := 1/z$
		  $z := z + 1$
		return	

Figure 10. This example shows three systems that contain dead code (assignments to the temporary local variables *t1*, *t2*, and *t3* and the call to *Incr* in *P*). The right-hand column shows a fragment of the SDG for *A* augmented with a meeting-point vertex (shown in bold). (The boxes indicate the modifications made to variants *A* and *B*.)

```
procedure Main
  a := 1
  b := 1
  call P(a, 1)
  call P(b, 2)
  output(a)
end

procedure P(x, y)
  t1 := x + y
  call Incr(x)
  t3 := x
  x := 2
return

procedure Incr(z)
  t2 := 1/z
  z := z + 1
return
```

Figure 11. The result of applying $Integrate^S$ to the three systems shown in Figure 10.

```
function Integrate^S(Base, A, B: system) returns a system or Failure declare
  M : system
  G : SDG begin
  G := Δ^S(A, Base) ∪ Δ^S(B, Base) ∪ Pre^S(Base, A, B)
  if Type_I_Interference(G, Base, A, B) then return (Failure) fi
  if Type_II_Interference(G) then return (Failure) fi
  if Type_III_Interference(G, Base, A, B) then return (Failure) fi
  M := ReconstituteSystem(G) return (M)
```

Figure 12. The function $Integrate^S$ takes as input three systems *Base*, *A*, and *B*, where *A* and *B* are variants of *Base*. Whenever the changes made to *Base* to create *A* and *B* do not interfere, $Integrate^S$ produces a system *M* that integrates *A* and *B* (the absence of Type II interference ensures the existence of a system *M* such that $G_M = G$).