MULTI-PROCEDURE PROGRAM INTEGRATION

by

David W. Binkley

Computer Sciences Technical Report #1038

August 1991

MULTI-PROCEDURE PROGRAM INTEGRATION

by

DAVID W. BINKLEY

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN—MADISON

1991

Abstract

The magnitude and complexity of existing and future software systems heightens the need for tools that assist programmers with the task of maintaining and developing software. One recurring problem that arises in system development is the need to reconcile multiple divergent lines of program development. When solved by hand, this reconciliation or integration process is often tedious and error prone. A better solution is the use of a *program integration tool*—a tool that takes as input several variants of a *base* program, automatically determines the changes in each variant with respect to the *base* program, and incorporates these changes, along with the portion of the *base* program preserved in all the variants, into a merged program.

Previous algorithms that solve the program-integration problem include text-based approaches, such as that used by the UNIX *diff3* utility. However, the text-based approach is unsatisfactory since it fails to guarantee any relationship between the behavior of the integrated program and the behaviors of the *base* program and its variants. In contrast, semantics-based approaches to program integration can exploit knowledge of a language's semantics to provide such a guarantee.

Previous semantics-based algorithms, however, have been limited to single-procedure programs. This dissertation extends one of these algorithms to handle programs that consist of multiple (possibly mutually recursive) procedures. In doing so it makes the following three major contributions.

(1) *Definition of the system dependence graph.* This graph extends previous dependence representations to incorporate collections of procedures and the interconnections between them.

(2) *New algorithms for interprocedural slicing.* These algorithms compute several kinds of interprocedural slices (*i.e.*, slices that cross the boundaries between procedures). The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. By taking calling context into account the algorithms described in this dissertation produce smaller (*i.e.*, better) slices than previous algorithms.

(3) *A semantics-based algorithm for multi-procedure integration.* As with slicing, the chief difficulty in multi-procedure integration is correctly accounting for calling context. The algorithm makes use of the system dependence graph, interprocedural slicing, and some additional preprocessing of the system dependence graph to correctly account for calling context.

Acknowledgements

To begin with, I must thank my advisor, Thomas Reps. He set high standards in the best way possible: by example. I really should thank my advisors; although she got none of the official credit, Susan Horwitz was an invaluable part of my graduate education.

When I took a days troubles home, it was my wife Judy, and later Judy and our daughter Erin, and still later, Judy, Erin, and our son Christopher who had to put up with me. I owe them a great deal. Judy always seemed to know when to let me work and when to make me play—Erin (and Chris) were always willing to do either.

There are a number of graduate students, past and present, that shaped my stay at UW–Madison. However, since any attempt to recall all of them would be plagued by omissions, let me simply give you all one big thanks.

Finally, I wish to thank Cathy from the McBurney center and Mary from the writing lab. Their willingness to help got me through my last year of graduate school.

Table of Contents

List of Figures

CHAPTER 1

PROGRAM INTEGRATION

Program integration concerns the merging process that becomes necessary when a program's source code diverges into multiple variants. A program integration tool takes as input several variants of a *base* program and after determining the changes in each variant (with respect to the *base* program), incorporates these changes, along with the portion of *base* that is preserved in all the variants, into a merged program. In addition to assembling a merged program the integration tool also determines if two modification interfere (the notion of "interference" is formalized later). In the absence of interference the integration tool must provide guarantees about how the execution behavior of the merged program relates to the execution behaviors of the variants. This dissertation extends previous semantics-based integration tools by considering programs that include procedures and procedure calls.

An example of the use of an integration tool is to provide a kind of optimistic concurrency control. In this scheme multiple programmers can update separate copies of a program in parallel *without* placing restrictions (*i.e.*, locks) on the software modules they are modifying (such restrictions essentially constrain the activities of programmers and hence reduce their productivity). For example, consider a compiler being upgraded by two programmers: the first is specializing the code generator to produce code for a particular architecture and the second is implementing a new register allocation scheme. Knowing that they have a program-integration tool, these two programmers could work independently (without hampering each others progress). The result of their work is two versions of the compiler, each containing one of the upgrades. When both programmers have completed their work, an integration tool would be invoked to produce a version of the compiler for the particular architecture that includes the new register allocator.

It is easy to imagine a variety of uses for an integration tool in areas including, for example, program development and program maintenance. More specifically, the need for program integration arises in the following 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 tackling such problems would obviously be useful.

At present, the only available tools for integration (*e.g.*, the UNIX[1] utility *diff3*) implement an operation for merging files as strings of text. This approach has the advantage that it is applicable to merging

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

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 that results from 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 checked carefully for conflicts that might have been introduced by the merge.

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 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 variants.

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

The remainder of this chapter is divided into three subsections. The first describes the model for semantics-based program integration discussed in [Horwitz89] and also introduces the first algorithm that satisfies this model (a detailed summary of this algorithm is given in Chapter 2). The second subsection describes the major contributions of this dissertation, which extends the approach in [Horwitz89] to programs that contain procedures and procedure calls. This chapter concludes with an outline of the remaining chapters of the dissertation.

1.1. The Horwitz, Prins, and Reps Semantics-Based Program Integration Algorithm

Horwitz, Prins, and Reps first formalized the problem of semantics-based integration [Horwitz87, Horwitz89]. To make this problem amenable to theoretical study they studied the problem using a simplified model. This model, referred to hereafter as the HPR model, possesses the essential requirements for an integration algorithm; thus, it permits the study of the program-integration problem to be performed in the absence of inessential detail.

The Horwitz-Prins-Reps Model of Program Integration

Informally, given a program *Base* and two variants *A* and *B* (each created by editing separate copies of *Base*), the goal of program integration is to determine whether the modifications in *A* and *B* interfere, and if they do not, to create a merged program *M* that incorporates the changed behavior of *A* with respect to *Base*, the changed behavior of *B* with respect to *Base*, and the unchanged behavior common to *Base*, *A*, and *B*.[2]

[2] 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*.

HPR Model of Program Integration

(1) Programs must be written in a simplified programming language that has only scalar variables and constants, assignment statements, conditional statements, while loops, and final output statements (called *end* statements); by definition, only those variables listed in the end statement have values in the final state. 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* in variant *A* that produces different sequences of values in *A* and *Base*, component *c* is in *M* and produces the same sequence of values as in *A* (*i.e.*, *M* agrees with *A* at component *c*).

  (iii) *M* captures the changed behavior of *B*: for any program component *c* in variant *B* that produces different sequences of values in *B* and *Base*, component *c* is in *M* and produces the same sequence of values as in *B* (*i.e.*, *M* agrees with *B* at component *c*).

  (iv) *M* captures the behavior of *Base* preserved in *A* and *B*: for any program component *c* that produces the same sequence of values in *Base*, *A*, and *B*, component *c* is in *M* and produces the same sequence of values as in *Base* (*i.e.*, *M* agrees with *Base*, *A*, and *B* at component *c*).

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

A "program component" means an assignment statement, the predicate of a while loop or conditional statement, or the occurrence of a variable in the program's end statement. "The sequence of values produced by a program component," means the following: for an assignment statement, the sequence of values assigned to the target variable; for a predicate, the sequence of boolean values to which the predicate evaluates; and for a variable named in the end statement, the singleton sequence containing the variable's final value. In addition, a program is assumed to have a special initial-definition component for each variable that can be used before being assigned to. Such components compute a singleton sequence containing the variable's initial value, as taken from the initial state.

Properties (1) and (3) of the model are syntactic restrictions that limit the scope of the integration problem. Property (2) defines the model's *semantic* criterion for integration and interference. A more 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."[4] *Any* program *M* that satisfies

[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] Originally, the changed behavior of a variant was defined in [Horwitz89] in terms of the *final* values of its variables rather than in terms of the sequences of values produced by all program components. However, that definition of changed behavior was unsatisfactory for several reasons. One problem is that, in a program development environment, dead code (code that has no effect on the final value of any variable) may be introduced temporarily; it is important that an integration algorithm preserve *all* changes made to a variant, including the introduction of dead code (which the programmer may intend to turn into live code at a later time). Another problem with defining Property (2) in terms of final values is that it restricts the way output statements can be handled. While Property (1) permits only final output statements, it is clearly desirable to relax that restriction to permit intermediate output statements. If Property (2) deals only with final values, the only way to preserve changes to intermediate output is to treat output as a single value, with every output statement concatenating onto the end of that value. With this treatment there is interference essentially every time both variants include any change to the program's output. An alternative philosophy for treating output statements, which reduces the number of cases in which interference is reported, is to treat output statements as producing independent sequences of values; in this case, Property (2) must be defined in terms of the sequence of values produced at intermediate program components in order to ensure that changes

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, Property (2) is not decidable, even under the restrictions given by Properties (1) and (3); consequently, any program-integration algorithm will 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).

The Horwitz-Prins-Reps Algorithm for Program Integration

The first algorithm that meets the requirements of the HPR model was formulated by Horwitz, Prins, and Reps [Horwitz87, Horwitz89]. That algorithm, referred to hereafter as the *HPR algorithm*, was the first algorithm for semantics-based program integration. The HPR algorithm represents a fundamental advance over text-based program integration algorithms (such as *diff3*), and provides the first step towards the creation of a semantics-based tool for program integration. The HPR algorithm is able to detect changes in *behavior*—rather than just changes in text; the algorithm either incorporates all behavioral changes in the integrated program or reports interference.

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 similar to the *program dependence graphs* used previously in vectorizing and parallelizing compilers [Kuck81, Ferrante87]. The HPR algorithm also makes use of Weiser's notion of a *program slice* [Weiser84, Ottenstein84] to find the statements of a program that contribute to the computation of the program elements with changed behavior. Program dependence graphs, program slicing, and the HPR algorithm are described in Chapter 2.

1.2. Contributions

A necessary and important step in extending the HPR algorithm to a full-fledged programming language is the capability to integrate programs that contain procedures and procedure calls. In designing an algorithm for integrating such programs, we would like to make use of the HPR algorithm (and its proof of correctness). An example of an algorithm that does this is the following three-step process: (1) transform three programs with procedures into equivalent programs without procedures, (2) apply the HPR algorithm, and (if the HPR algorithm does not report interference) (3) reverse the transformation from step (1) to produce a program with procedures. The transformation used in step (1) is accomplished by *roll-out*: the exhaustive in-line substitution of call statements. Unfortunately, the roll-out of a recursive program is infinite. Thus, for languages with recursion, the above three-step process cannot be used to implement multi-procedure integration.[5] However, the idea of roll-out is used in Chapter 4 to formulate a criterion against which our algorithm is measured. The algorithm does not actually perform any in-line expansions, it operates on finite representations of programs (called system dependence graphs).

The primary contribution of this dissertation is an algorithm for semantics-based multi-procedure program integration that meets the criterion discussed above. That is, we present an algorithm that succeeds in integrating three programs with procedures whenever the three-step process defined above succeeds in producing a satisfactory integrated program. Two of the important building blocks developed as part of this algorithm are the definition of the system dependence graph and an algorithm for precise interprocedural

---

to non-final output statements are preserved.

[5] Even for programs without recursion, the three-step process is not practical because the number of copies of a procedure introduced by *roll-out* may be exponential in the number of call-sites in the program.

slicing.

1.2.1. The System Dependence Graph

The system dependence graph, described in Chapter 3, extends previous dependence graphs in two ways: it explicitly represents multiple procedures and both the *interprocedural control dependence* between a call-site and the called procedure and the *interprocedural data dependences* between actual and formal parameters. Another key feature of the system dependence graph is its inclusion of transitive dependence edges at call-sites (in addition to conventional direct-dependence edges). These edges summarize paths of edges through called procedures, thus permitting information about a called procedure to be obtained *without* examining the called procedure. This capability significantly reduces the complexity of algorithms (*e.g.*, interprocedural slicing) that must keep track of the *calling context* of a called procedure.

1.2.2. Precise Interprocedural Slicing

The purpose of any slicing algorithm is to identify a reduced program that captures some part of the computation of the original program; therefore, the usefulness of a slicing algorithm is inversely proportional to the size of the slices it produces. Hence a more precise slicing algorithm (*i.e.*, one that produces smaller slices) is of greater utility. Chapter 3 discusses an interprocedural slicing algorithm that is more precise than its predecessor [Weiser84].

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. As discussed in Chapter 3, the transitive dependence edges in a system dependence graph are used in conjunction with a two-pass algorithm to solve the calling-context problem as it arises in interprocedural slicing. This approach leads to more precise slices than the original interprocedural slicing algorithm given by Weiser [Weiser84] and provides greater efficiency, particularly when more than one slice of the same program is desired, than the algorithm given by Hwang et. al. [Hwang88], which has similar precision.

1.2.3. A Multi-Procedure Integration Algorithm

As discussed in Chapter 5, the primary contribution of this dissertation is the first algorithm for semantics-based multi-procedure program integration; this algorithm extends previous semantics-based integration algorithms, which apply only to single-procedure programs. As with interprocedural slicing the chief difficulty in integrating multi-procedure programs is correctly accounting for calling context (an algorithm that does not account for calling context is used in Chapter 4 as a strawman to motivate the need for more precise techniques).

To handle the calling-context problem as it arises in multi-procedure integration, the algorithm makes use of the transitive dependence edges in a system dependence graph, precise interprocedural slicing, and an additional kind of transitive dependence edge that summarizes paths to dead code inside a procedure's body. By accounting for calling context, the algorithm can more precisely capture the changed and preserved components of the variants; thus, it can succeed in returning an integrated program in cases where a less precise algorithm (*e.g.*, the one developed in Chapter 4) reports interference and therefore fails to return an integrated program.

In addition to integrating multi-procedure programs, some of the intermediate results from the multi-procedure integration algorithm have uses outside program integration. For example, the term $\Delta^S(P, Q)$, which captures the differences between programs P and Q, can be used to determine a reduced program from P that computes all the changed behavior of P when compared with the behavior of Q. Such information has obvious uses in software development and software maintenance. For example, if a program maintainer has produced P from Q as the result of fixing a bug in Q, then, rather than retesting all of Q's

functionality, only that portion represented in $\Delta^S(P, Q)$ must be retested because the remainder of P is computationally equivalent to Q. This is particularly beneficial when $\Delta^S(P, Q)$ is small and P and Q are large.

1.3. Dissertation Overview

The remainder of this dissertation is divided into eight chapters. Chapter 2 provides necessary background material on program dependence graphs (for single procedure programs), intraprocedural slicing, and the HPR integration algorithm. System dependence graphs and interprocedural slicing are the topic of Chapter 3. Chapters 4 and 5 discuss multi-procedure integration: Chapter 4 identifies shortcomings (when integrating multi-procedure programs) in the HPR integration model and develops a new model more appropriate for multi-procedure integration; Chapter 5 describes an algorithm that satisfies this model. Chapters 6, 7, and 8 are devoted to proving that the algorithm from Chapter 5 satisfies the model from Chapter 4. Finally, Chapter 9 summarizes the work described in the dissertation and describes possible future work on program integration.

CHAPTER 2

BACKGROUND

This chapter summarizes the HPR program-integration algorithm. It first describes two key components of the algorithm: program dependence graphs and program slicing, before summarizing the steps of the HPR algorithm itself.

2.1. PROGRAM DEPENDENCE GRAPHS

Different definitions of program dependence representations have been given, depending on the intended application; they are all variations on a theme introduced in [Kuck72], and share the common feature of having an explicit representation of data dependences (see below). The "program dependence graphs" defined in [Ferrante87] introduced the additional feature of an explicit representation for control dependences (see below). The definition of program dependence graph given below differs from [Ferrante87] in two ways. First, it covers only a restricted language with scalar variables, assignment statements, conditional statements, while loops, and a restricted kind of "output statement" called an *end statement*,[1] and hence is less general than the one given in [Ferrante87]. Second, we omit certain classes of data dependence edges and make use of a class introduced in [Horwitz87]. Despite these differences, the structures defined below and those defined in [Ferrante87] share the feature of explicitly representing both control and data dependences. (See [Pfeiffer91] for a survey and history of dependence graphs.)

The program dependence graph for program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges.[2] The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

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

(2) For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined (see [Aho86]), there is a vertex called the *initial definition of* x, which represents an assignment to x from the initial state and is labeled "x := InitialState(x)."

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

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

[1] As described in the HPR model in Chapter 1, an end statement, which can only appear at the end of a program, names one or more of the variables used in the program; when execution terminates, only those variables will have values in the final state; the variables named by the end statement are those whose final values are of interest to the programmer.

[2] A *directed graph* G is a pair consisting of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(s, t) \in E(G)$ is directed from s to t, where s is the *source* and t the *target* of the edge. The graph having vertex set V and edge set E is denoted by (V, E).

source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \rightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will eventually be executed if the program terminates normally. A method for determining control dependence edges for arbitrary programs is given in [Ferrante87]; however, because we are assuming that programs include only assignment, conditional, while, and end statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under consideration here, the control dependences reflect a program's nesting structure.

DEFINITION. (Control Dependence Edges). Program dependence graph G_P contains a *control dependence edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

(1) v_1 is the entry vertex, and v_2 represents a component of P that is not nested within any loop or conditional; these edges are labeled **true**.

(2) v_1 represents a control predicate, and v_2 represents a component of P immediately nested within the loop or conditional whose predicate is represented by v_1. If v_1 is the predicate of a while-loop, the edge $v_1 \rightarrow_c v_2$ is labeled **true**; if v_1 is the predicate of a conditional statement, the edge $v_1 \rightarrow_c v_2$ is labeled **true** or **false** according to whether v_2 occurs in the **then** branch or the **else** branch, respectively.[3]

A data dependence edge from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. In this dissertation, program dependence graphs contain two kinds of data-dependence edges, representing *flow dependences* and *def-order dependences*.[4] The data-dependence edges of a program dependence graph are computed using data-flow analysis. For the restricted languages considered in this dissertation, the necessary computations can be defined in a syntax-directed manner.

DEFINITION. (Flow Dependence Edges). A program dependence graph contains a *flow dependence edge* from vertex v_1 to vertex v_2 iff all of the following hold:

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

(2) v_2 is a vertex that uses x.

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

A flow dependence from vertex v_1 to vertex v_2 will be denoted by $v_1 \rightarrow_f v_2$.

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

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

---

[3]In other definitions that have been given for control dependence edges, there is an additional edge from each predicate of a **while** statement to itself labeled **true**. This kind of control edge is left out of our definition because it is not necessary for our purposes.

[4]For a complete discussion of the need for these edges and a comparison of def-order dependences with anti- and output dependences see [Horwitz88].

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

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

DEFINITION. (Def-order Dependence Edges) A program dependence graph contains a *def-order dependence edge* from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 and v_2 both define the same variable.

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

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

(4) The program component represented by v_1 occurs before the program component represented by v_2 in a preorder traversal of the program's abstract-syntax tree.

A def-order dependence from v_1 to v_2 with "witness" v_3 is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$. However, it is often useful to think of this edge as a hyper-edge from source v_1 to source v_2 and then to target v_3. For example, only when def-order edges are thought of in this way the following statement is true for all kinds of edges "the evaluation of the component (components) represented by the source (sources) of an edge affect the evaluation of the component represented by the target of the edge."

Note that a program dependence graph is a multi-graph (*i.e.*, it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependence edge between two vertices, each is labeled by a different loop that carries the dependence. When there is more than one def-order edge between two vertices, each has a different witness.

The adequacy of the program dependence graph as a representation for programs has been addressed in [Horwitz88], where is it shown that two programs with isomorphic program dependence graphs are strongly equivalent. In other words, if P and Q have isomorphic program dependence graphs then, when run on the same initial state, P and Q either both terminate in the same final state or they both fail to terminate normally.

Example. Figure 2.1 shows an example program and its program dependence graph.

2.2. Program Slices (of Single-Procedure Programs)

The *slice* of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p. This concept, originally discussed by Mark Weiser in [Weiser84], can be used to isolate individual computation threads within a program. Slicing can help a programmer understand complicated code, can aid in debugging [Lyle86], and can be used for automatic parallelization [Weiser83, Badger88]. Finally, program slicing, as described in Section 2.2.1 is used in the HPR program integration algorithm, which is the topic of Section 2.3.

In Weiser's terminology, a *slicing criterion* is a pair <p, V>, where p is a program point and V is a subset of the program's variables. In his work, a slice consists of all statements and predicates of the program that might affect the values of variables in V at point p. This is a more general kind of slice than is often needed: rather than a slice taken with respect to program point p and an *arbitrary* variable, one is often interested in a slice taken with respect to a variable x that is *defined* or *used* at p. The value of a variable x defined at p is directly affected by the values of the variables used at p and by the loops and conditionals that enclose p. The value of a variable y *used* at p is directly affected by assignments to y that reach p and by the loops and conditionals that enclose p. When slicing a program that consists of a single monolithic procedure, a slice can be determined from the closure of the directly-affects relation. Ottenstein and Ottenstein pointed out how well-suited *program dependence graphs* are for this kind of slicing [Ottenstein84];

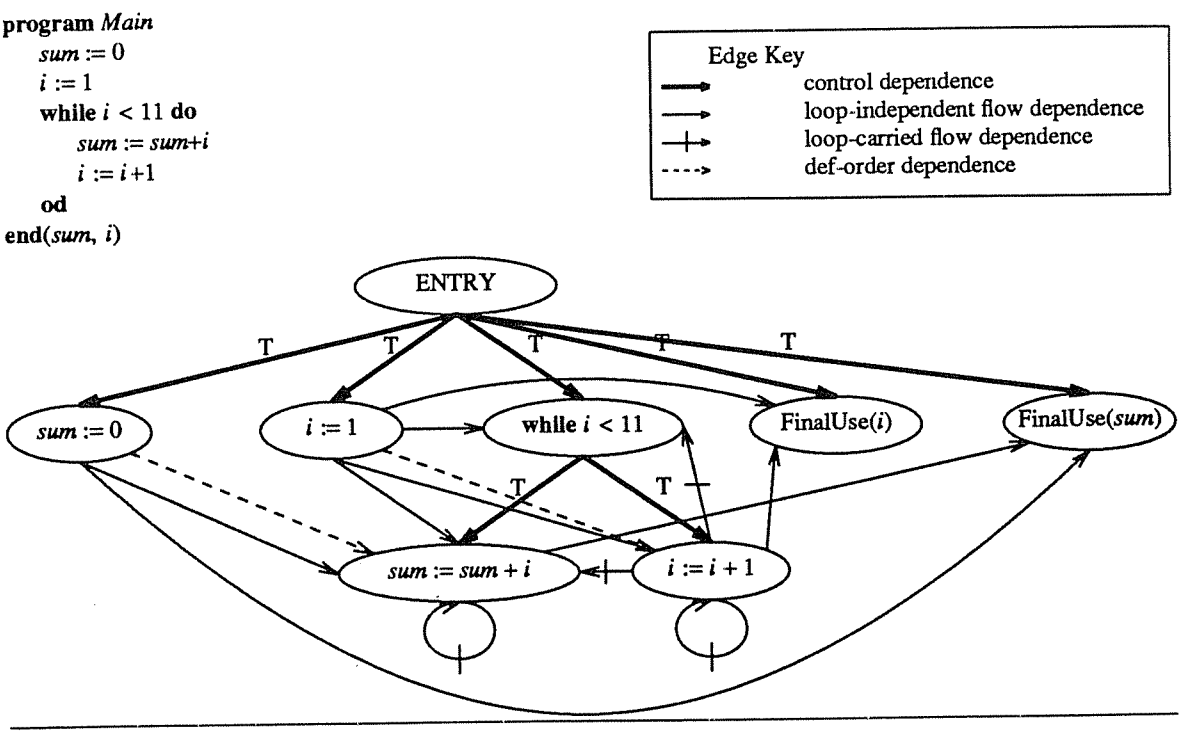


Figure 2.1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. (The boldface arrows represent control dependence edges, solid arrows represent loop-independent flow dependence edges, solid arrows with a hash mark represent loop-carried flow dependence edges, and dashed arrows represent def-order dependence edges.)

once a program is represented by its program dependence graph, the slicing problem is simply a vertex-reachability problem, and thus slices may be computed in linear time.

2.2.1. HPR Program Slices

This section introduces the two kinds of *HPR* slices:[5] backward and forward. The original formulation of the HPR algorithm, given in Section 2.3, uses only backward slices; we include the definition of forward slices at the end of this section for two reasons: first, Chapter 3 addresses both backward and forward interprocedural slicing and second, forward HPR slices are used in Chapter 5 in a reformulation of the HPR algorithm. Since the predominate use of slicing in the following chapters is the use of backward slicing, we use the term "slice" to refer to a "backward slice" (occasionally adding an explicit "backward" only for emphasis); a forward slice is always referred to as a "forward slice".

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

[5] In previous papers we have referred to an "HPR slice" as an "in*tra*procedural slice"; however, in this dissertation, the term in*tra*procedural slice is reserved for an extended HPR slice, defined in Chapter 5.

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

For a single vertex, we define $b^{hpr}(G, v) = b^{hpr}(G, \{ v \})$ and define $b^{hpr}(G, v) = (G, \emptyset)$ for any $v \notin V(G)$. Figure 2.2 gives a simple worklist algorithm for computing the vertices of a slice using a program dependence graph.

As a notational shorthand, when taking the slice of a program's program dependence graph, we sometimes use the name of a program to denote the corresponding program dependence graph, for example $b^{hpr}(Base, S)$ in place of $b^{hpr}(G_{Base}, S)$.

The slice of graph G with respect to vertex set S 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 program dependence graph induced by the vertex set. (A def-order edge $v \longrightarrow_{do(u)} w$ is included in the induced edge set iff u, v, and w are all included in the vertex set.) Formally, *Induce* is defined as follows:

$$\begin{aligned} Induce(G, S) \triangleq (S, E'), \text{ where } E' = \quad & \{ v \longrightarrow_f w \in E(G) \mid v, w \in S \} \\ & \cup \{ v \longrightarrow_c w \in E(G) \mid v, w \in S \} \\ & \cup \{ v \longrightarrow_{do(u)} w \in E(G) \mid u, v, w \in S \}. \end{aligned}$$

The relationship between a program's dependence graph and a slice of the graph has been addressed in [Reps88]. We say that G is a *feasible* program dependence graph iff G is the program dependence graph of some program P. For any $S \subseteq V(G)$, if G is a feasible program dependence graph, the slice $Induce(b^{hpr}(G, S))$ is also a feasible program dependence graph; it corresponds to the program P' obtained by restricting the syntax tree of P to the statements and predicates whose vertices are in $Induce(b^{hpr}(G, S))$ [Reps88].

```
procedure MarkVerticesOfSlice(G, S)
declare
  G: a program dependence graph
  S: a set of vertices in G
  WorkList: a set of vertices in G
  v, w: vertices in G
begin
  WorkList := S
  while WorkList ≠ ∅ do
    Select and remove vertex v from WorkList
    Mark v
    for each unmarked vertex w such that edge w →f v or edge w →c v is in E(G) do
      Insert w into WorkList
    od
  od
end
```

Figure 2.2. A worklist algorithm that marks the vertices in G / S. Vertex v is in G / S if there is a path along flow and/or control edges from v to some vertex in S.

---

[6] In previous papers the notation G / v has been used to denote the backward slice of G with respect to v. Because of the variety of different kinds of slices used in multi-procedure integration we have introduced names for some of the lower-level operations used to generate slices (*e.g.*, b^{hpr} and *Induce*).

Example. Figure 2.3 shows the graph that results from taking a slice of the program dependence graph from Figure 2.1 with respect to the final-use vertex for i, together with the one program to which it corresponds.

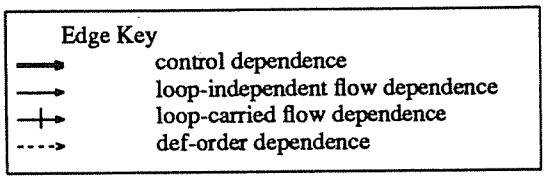
The significance of a slice is that it captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each component of the slice (*i.e.*, each program dependence graph vertex in the slice) [Reps88]. Recall from Chapter 1 that a program point is (1) an assignment statement, (2) a control predicate, (3) an initial-definition point (for a variable that may be referenced before being assigned to), or (4) a final use of a variable in an end statement. And that by "computing the same sequence of values for each component of (point in) the slice" we mean: (1) for any assignment statement the same *sequence* of values is assigned to the target variable; (2) for a predicate the same *sequence* of boolean values is produced; (3) for an initial-definition point the same value is assigned to the target variable (this values comes from the initial state); and (4) for each final use the same value for the variable is produced.

Forward Slicing

Forward slicing is the dual of backward slicing: whereas the *backward slice* of a program with respect to a program point p and variable x consists of all statements and predicates of the program that potentially *affect* the value of x at point p, the *forward slice* of a program with respect to a program point p and variable x consists of all statements and predicates of the program that are potentially *affected by* the value of x at point p.

---

program *Main*
  $i := 1$
  while $i < 11$ **do**
    $i := i + 1$
  od
end(i)



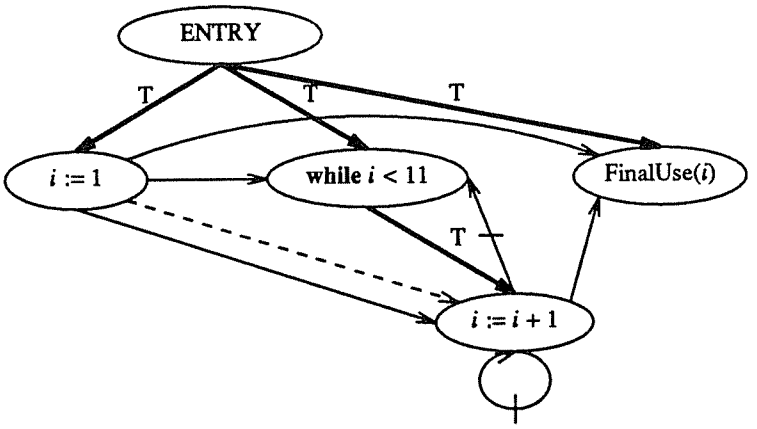


Figure 2.3. The graph and the corresponding program that result from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for i.

For a vertex v of program dependence graph (PDG) G, the vertices of a forward slice of G with respect to v are all vertices that are transitively flow or control dependent on v. We define an operator f^{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:

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

For a single vertex, we define $f^{hpr}(G, v) = f^{hpr}(G, \{ v \})$ and define $f^{hpr}(G, v) = (G, \varnothing)$ for any $v \notin V(G)$. The forward slice of G with respect to v is denoted by $Induce(f^{hpr}(G, S))$.

Finally, unlike a backward slice, the forward slice of a feasible program dependence graph (*i.e.*, a graph that is the program dependence graph of some program) does not correspond to a feasible program dependence graph. For example, the forward slice with respect to the statement "*sum* := 0" in Figure 2.1 consists of the vertices labeled "*sum* := 0", "*sum* := *sum*+*i*", and *FinalUse*(*sum*) together with the flow and def-order edges that connect them. This slice does not include the entry vertex or the control predecessor of "*sum* := *sum*+*i*"; either of these omissions implies that the slice is infeasible.

2.3. Summary of the HPR Integration Algorithm

This section provides an overview of the HPR integration algorithm; a more detailed description can be found in reference [Horwitz89]. In particular, [Horwitz89] contains a running example that illustrates the different steps of the algorithm; the reader may find it helpful to refer to Figures 1–8 of [Horwitz89], which depict the results of the various graph manipulations that the algorithm performs (an abbreviated version of this example is given at the end of this section).

As outlined in Chapter 1, given a program *Base* and two variants *A* and *B*, each created by editing separate copies of *Base*, the HPR algorithm determines whether the changes made to *Base* to produce *A* and *B* interfere; if there is no interference, the algorithm produces a merged program *M* that incorporates the changed behavior of *A* with respect to *Base*, the changed behavior of *B* with respect to *Base*, and the unchanged behavior common to *Base*, *A*, and *B*.

The HPR algorithm requires that program components (*i.e.*, statements and predicates) be tagged so that corresponding components can be identified in all three versions. Component tags can be provided by a special editor that obeys the following conventions:[7]

(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 deleted component is never re-used; a component 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.

Component tags are used to determine *corresponding* vertices when performing operations that use vertices from different program dependence graphs; two vertices are *corresponding* if they have the same tag.

The HPR algorithm uses backward slicing to determine a safe approximation to the changed and preserved behaviors of variants *A* and *B* with respect to *Base*, and to determine whether the modifications

[7] 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 [Donzeau-Gouge84], GANDALF [Notkin85], and the Synthesizer Generator [Reps88a].

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

Step 1: Determining changed and preserved 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 [Reps88]. In other words, vertex v is a site that potentially exhibits changed behavior in A when compared with *Base*. 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 (as with the slicing operators, the various integration operators all return (program dependence graph, vertex-set) pairs):

$$AP^{hpr}(A, Base) \triangleq (A, \{ v \in V(A) \mid Induce(b^{hpr}(Base, v)) \neq Induce(b^{hpr}(A, v)) \}).$$

We define $AP^{hpr}(B, Base)$ similarly. It follows that 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:

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

A vertex that has the same slice in all three programs is guaranteed to exhibit the same behavior [Reps88]. 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:

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

Step 2: Forming the merged graph

The merged graph G_M is formed by taking the graph union[8] 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:

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

Step 3: 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 criterion for Type I interference is based on a comparison of slices of A, B, and G_M. The slices $b^{hpr}(AP^{hpr}(A, Base))$ and $b^{hpr}(AP^{hpr}(B, Base))$ capture the changed slices of A and B, respectively. There is Type I interference if G_M does not preserve these slices; that is, there is Type I interference if either of the following is true:

there exists a vertex v in $AP^{hpr}(A, Base)$ such that $Induce(b^{hpr}(G_M, v)) \neq Induce(b^{hpr}(A, v))$
there exists a vertex u in $AP^{hpr}(B, Base)$ such that $Induce(b^{hpr}(G_M, u)) \neq Induce(b^{hpr}(B, u))$.

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* program dependence graph; this is Type II interference. (See [Horwitz89] or [Ball90] for a discussion of how to determine whether G_M is feasible, and if so how to reconstitute a program from G_M.) If neither kind of interference occurs, a program whose program dependence graph is G_M is returned as the result of the integration.

[8] 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: $(V_1 \cup V_2, E_1 \cup E_2)$.

One of the most important aspects of the HPR algorithm is that it provides *semantic* guarantees about the behavior of the integrated program. In particular, it has been shown that the HPR algorithm satisfies Property (2) of the HPR integration model discussed in Section 1.1 [Reps88].

2.3.1. HPR Integration Example

This section presents an abbreviated version of the running example given in [Horwitz89]. Figure 2.1 shows a program that sums the integers from 1 to 10 and its corresponding program dependence graph. We now consider two variants of this program, shown in Figure 2.4 with their program dependence graphs:

1) In variant A two statements have been added to the original program to compute the product of the integers from 1 to 10.

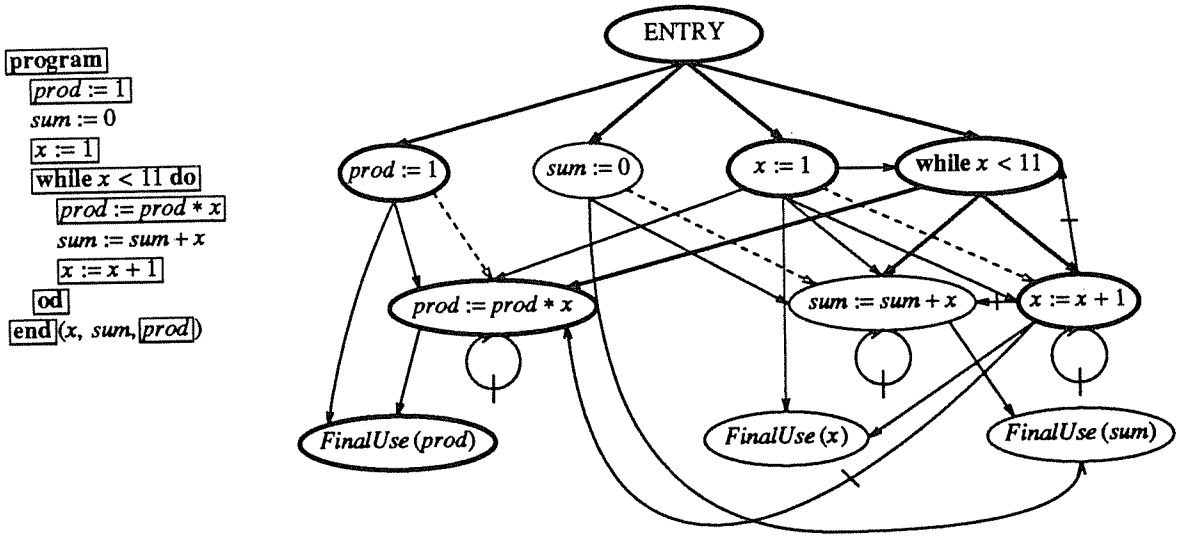
2) In variant B one statement has been added to compute the mean of the sequence.

These two programs represent non-interfering enhancements of the original summation program. The affected points of A with respect to *Base* are the assignment vertices labeled "*prod* := 1" and "*prod* := *prod* * *x*" as well as the final-use vertex for *prod*. Similarly, the affected points of B with respect to *Base* are the assignment vertex labeled "*mean* := *sum* / 10" and the final-use vertex for *mean*.

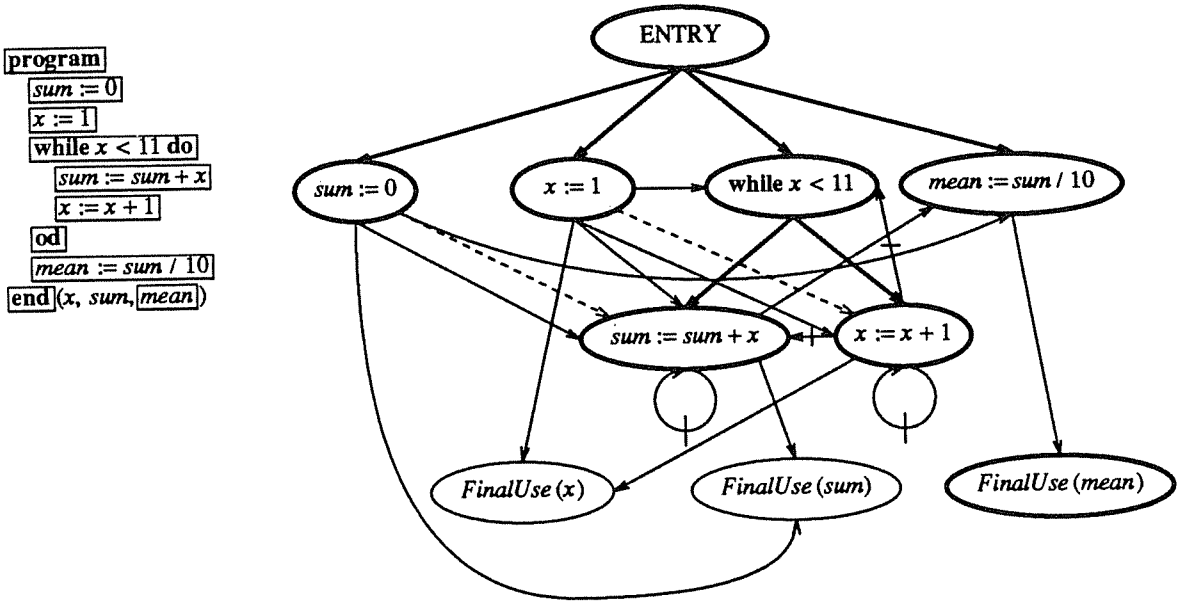
Figure 2.4 also shows the vertex sets for two of the three graphs that form G_M: the vertices of $\Delta^{hpr}(A, Base)$ (*i.e.*, the vertices in $b^{hpr}(AP^{hpr}(A, Base))$) and the vertices of $\Delta^{hpr}(B, Base)$ (*i.e.*, the vertices in $b^{hpr}(AP^{hpr}(B, Base))$); the subgraphs of G_A and G_B induced by these vertices capture the changed behaviors of A and B, respectively. The third graph, $Induce(Base, Pre^{hpr}(A, Base, B))$, which captures the behavior of *Base* preserved in both A and B, is identical to the program dependence graph shown in Figure 2.1 because every vertex in G_{Base} has the same slice in G_{Base}, G_A, and G_B (*i.e.*, $Pre^{hpr}(A, Base, B)$ contains all the vertices of G_{Base}).

The merged graph G_M, shown in Figure 2.5, is formed by taking the union of the sub-graphs of G_A and G_B induced by the bold vertices shown in Figure 2.4(a) and Figure 2.4(b), respectively, and the graph shown in Figure 2.1. An inspection of the merged graph shown in Figure 2.5 reveals that there is no Type I interference: the slices of G_M and A with respect to the affected points of A (the bold vertices in Figures 2.4(a)) are the same and the slices of G_M and B with respect to the affected points of B (the bold vertices in Figures 2.4(b)) are the same. Finally, this graph is feasible; one of the programs that has G_M as its program dependence graph is shown below:

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



(a) Variant *A* and its program dependence graph.



(b) Variant *B* and its program dependence graph.

Figure 2.4. Variants *A* and *B* of the base program shown in Figure 1, and their program dependence graphs. (Boxed statements correspond to the vertices in $\Delta^{hpr}(A, Base)$ and $\Delta^{hpr}(B, Base)$, respectively, which are shown in bold.)

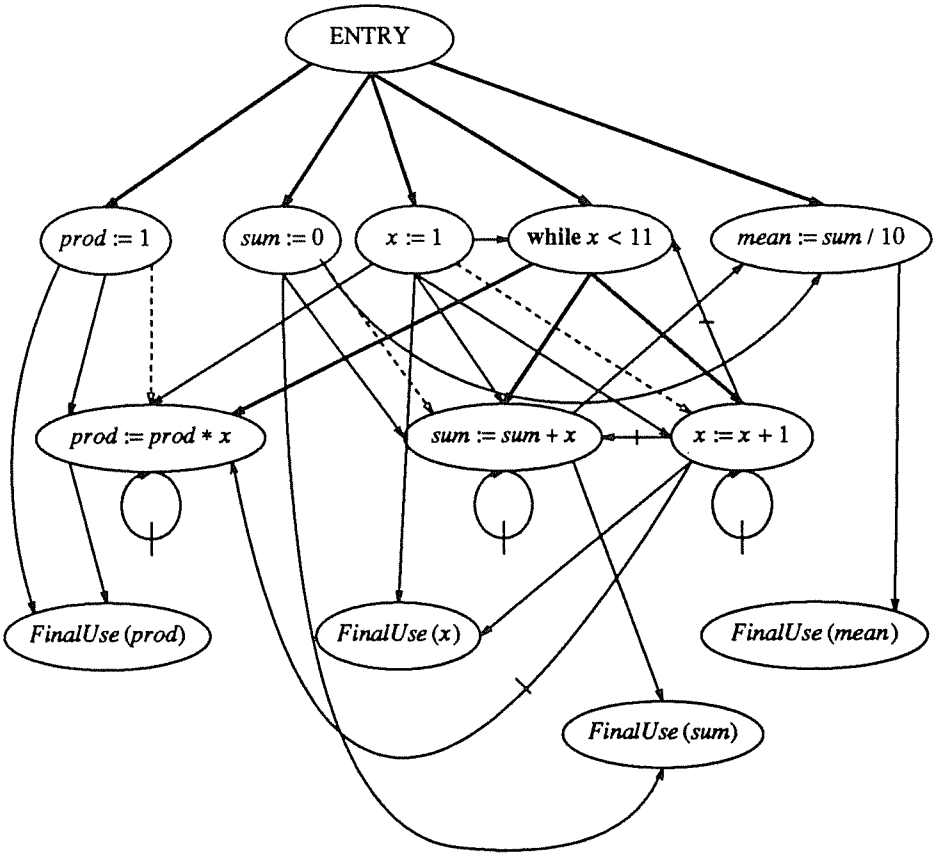


Figure 2.5. G_M is created by taking the union of the induced graphs computed from the bold vertices shown in Figures 2.4(a) and 2.4(b), and the graph shown in Figure 2.1.

CHAPTER 3

THE SYSTEM DEPENDENCE GRAPH AND INTERPROCEDURAL SLICING

This chapter lays the foundation for the remainder of the dissertation. It begins by informally discussing the need for more precise interprocedure slicing. Section 3.2 then describes the extensions to the programming language introduced in the previous chapter to include procedures and procedure calls; Section 3.2 also introduces the *system dependence graph*, the dependence graph representation used to represent programs containing procedures and procedure calls. The main result of this chapter is the precise interprocedural slicing algorithm discussed in Section 3.3. The chapter concludes with a discussion of related work.

3.1. Interprocedural Slicing: Informal Discussion

As described in the beginning of Chapter 2, the *backward slice* of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p. Our main concern in this chapter is the problem of in*ter*procedural slicing—generating a slice of an entire program, where the slice crosses the boundaries between procedures. The algorithm for interprocedural slicing discussed in Section 3.3 produces a more precise answer than that produced by the interprocedural slicing algorithm given by Weiser in [Weiser84]. Precision is important because the quality of the multi-procedure integration algorithm presented in Chapter 5 depends to a large extent on the precision of the interprocedural slicing algorithm upon which it a based.

The algorithm in Section 3.3, like the algorithm used in the HPR integration algorithm, follows the example of Ottenstein and Ottenstein by computing a slice of a program as an operation on a dependence graph representation for the program [Ottenstein84]; however, in [Ottenstein84] Ottenstein and Ottenstein only discuss the case of programs that consist of a single monolithic procedure and do not discuss the more general case where slices cross procedure boundaries. As discussed below, the chief difficulty in computing such a slice is correctly accounting for the calling context of a called procedure.

It is important to understand the distinction between two different but related "slicing problems:"

Version (1)

> The slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p.

Version (2)

> The slice of a program with respect to program point p and variable x consists of a reduced program that computes the same sequence of values for x at p. That is, at point p the behavior of the reduced program with respect to variable x is indistinguishable from that of the original program.

When slicing single-procedure programs, a solution to Version (1) provides a solution to Version (2), since the "reduced program" required in Version (2) can be obtained by restricting the original program to just

the statements and predicates found in the solution for Version (1) [Reps89].

When slicing multi-procedure programs, restricting the original program to just the statements and predicates found for Version (1) may yield a program that is syntactically incorrect (and thus certainly not a solution to Version (2)). The reason behind this phenomenon has to do with multiple calls to the same procedure: it is possible that the program elements found by an algorithm for Version (1) will include more than one such call, each passing a different subset of the procedure's parameters.

This chapter addresses Version (1) of the interprocedural slicing problem (with the further restriction—mentioned in Chapter 2—that a slice can only be taken with respect to program point p and variable x if x is defined or used at p). The algorithm given in this chapter identifies a subgraph of the program's system dependence graph whose components might affect the sequence of values for x at p. A solution to Version (2) requires either that the slice be extended or that it be transformed by duplicating code to specialize procedure bodies for particular parameter-usage patterns. (It should be noted that, although it is imprecise, Weiser's algorithm produces a solution to Version (2).)

Recall that as discussed in Chapter 2, Weiser described a slice using a *slicing criterion*, a pair $<p, V>$, where p is a program point and V is a subset of the program's variables. Weiser's method for interprocedural slicing is described in [Weiser84] as follows:

> For each criterion C for a procedure P, there is a set of criteria $\mathrm{UP}_0(C)$ which are those needed to slice callers of P, and a set of criteria $\mathrm{DOWN}_0(C)$ which are those needed to slice procedures called by P. . . . $\mathrm{UP}_0(C)$ and $\mathrm{DOWN}_0(C)$ can be extended to functions UP and DOWN which map sets of criteria into sets of criteria. Let CC be any set of criteria. Then
>
> $$\mathrm{UP}(CC) = \bigcup_{C \in CC} \mathrm{UP}_0(C)$$
>
> $$\mathrm{DOWN}(CC) = \bigcup_{C \in CC} \mathrm{DOWN}_0(C)$$
>
> The union and transitive closure of UP and DOWN are defined in the usual way for relations. $(\mathrm{UP} \cup \mathrm{DOWN})^*$ will map any set of criteria into all those criteria necessary to complete the corresponding slices through all calling and called routines. The complete interprocedural slice for a criterion C is then just the union of the intraprocedural slices for each criterion in $(\mathrm{UP} \cup \mathrm{DOWN})^*(C)$.

This method does not produce as precise a slice as possible because the transitive-closure operation fails to account for the calling context of a called procedure.[1]

Example. To illustrate this problem, and the shortcomings of Weiser's algorithm, consider the following example program, which sums the integers from 1 to 10. (Except in Section 3.3.3, where call-by-reference parameter passing is discussed, parameters are passed by value-result.)

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

Using Weiser's algorithm to slice this program with respect to variable z and the **return** statement of

[1]For example, the relation $(\mathrm{UP} \cup \mathrm{DOWN})^*(<p, V>)$ includes the relation $\mathrm{UP}(\mathrm{DOWN}(<p, V>))$. $\mathrm{UP}(\mathrm{DOWN}(<p, V>))$ includes all call sites that call procedures containing the program points in $\mathrm{DOWN}(<p, V>)$, not just the procedure that contains p. This fails to account for the calling context, namely the procedure that contains p.

procedure *Increment*, we obtain everything from the original program. However, a closer inspection reveals that computations involving the variable *sum* do not contribute to the value of *z* at the end of procedure *Increment*; in particular, neither the initialization of *sum*, nor the first actual parameter of the call on procedure *A* in *Main*, nor the call on *Add* in *A* (which adds the current value of *i* to *sum*) should be included in the slice. The reason these components are included in the slice computed by Weiser's algorithm is as follows: the initial slicing criterion "<end of procedure *Increment*, *z*>," is mapped by the DOWN relation to a slicing criterion "<end of procedure *Add*, *a*>." The latter criterion is then mapped by the UP relation to *two* slicing criteria—corresponding to *all* sites that call *Add*—the criterion "<call on *Add* in *Increment*, *z*>" and the (irrelevant) criterion "<call on *Add* in *A*, *x*>." Weiser's algorithm does not produce as precise a slice as possible because transitive closure fails to account for the calling context (*Increment*) of a called procedure (*Add*), and thus generates a spurious criterion (<call on *Add* in *A*, *x*>).

A more precise slice consists of the following elements:

```
program Main
  i := 1
  while i < 11 do
     call A(i)
  od
end

procedure A(y)
  call Increment(y)
return

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

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

This set of program elements is computed by the slicing algorithm described in Section 3.3.

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To address the calling-context problem, system dependence graphs include some data-dependence edges that represent *transitive* dependences due to the effects of procedure calls, in addition to the conventional edges for direct dependences.

The cornerstone of the construction of the system dependence graph is the use of an attribute grammar to represent calling and parameter-linkage relationships among procedures. The step of computing the required transitive-dependence edges is reduced to the construction of the subordinate characteristic graphs for the grammar's nonterminals (an introduction to the terminology and concepts from attribute-grammar theory that are used in this chapter may be found in the Appendix to Chapter 3). The need to express this step in this fashion (rather than, for example, with transitive closure) is discussed further in Section 3.3.2.

The remainder of the chapter is organized as follows: Section 3.2 introduces procedures and procedure calls into the language used in Chapter 2 and discusses the system dependence graph. Section 3.3 presents an efficient, precise interprocedural slicing algorithm, which operates on the system dependence graphs and correctly accounts for the calling context of a called procedure. Section 3.3 also describes four related topics: how to improve the precision of interprocedural slicing using interprocedural summary information in the construction of system dependence graphs, how to handle programs with aliasing, how to compute precise interprocedural forward slices, and how to slice incomplete programs.

3.2. The System Dependence Graph: A Multi-procedural Dependence Graph Representation With Explicit Interprocedural Dependences

We now turn to the definition of the *system dependence graph*. The system dependence graph, an extension of the program dependence graphs defined in Chapter 2, represents programs in a language that includes procedures and procedure calls. In particular, our definition of the system dependence graph models a language with the following properties:

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

(2) Auxiliary procedures end with **return** statements instead of **end** statements (as defined in Chapter 2). A **return** statement does not include a list of variables.

(3) Parameters are passed by value-result.

We make the further assumptions that 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 in the remainder of the dissertation, except in Section 3.3.2, which uses interprocedural summary information to determine which parameters and global variables are defined and used by a procedure. Finally, in order to simplify the presentation, we assume there are no calls on the main procedure. This assumption is easily removed by including as the entry point for the system an invisible (uncallable) procedure that simply calls the existing main procedure.

It should become clear that our approach is not tied to the particular language features enumerated above. Modeling different features will require some adaptation; however, the basic approach is applicable to languages that allow nested scopes and languages that use different parameter-passing mechanisms. Section 3.3.3 discusses how to deal with systems that use call-by-reference parameter passing and contain aliasing.

A system dependence graph 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 Chapter 2, 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"). The definition of a system dependence graph is motivated by the following concerns:

(1) The graph representation should be correct in the sense that inequivalent programs (systems) should have non-isomorphic graph representations. The correctness of the system dependence graphs developed in Section 3.3.2 is shown in [Binkley89]

(2) A (correct) representation that allows some class of equivalent programs to have isomorphic dependence graphs should be chosen over one that distinguishes among members of the class by giving them non-isomorphic dependence graphs. As discussed in Section 3.2.2, this concern motivates the use of interprocedural data-flow information in the definition of a system dependence graph.

(3) The graph representation should support efficient, precise interprocedural slicing. An algorithm for slicing a system dependence graph is presented in Section 3.3.

Section 3.2.1 discusses how procedure calls and procedure entry are represented in procedure dependence graphs and how *procedure linkage edges* (the edges representing dependences between a call site and the called procedure) are added to connect these graphs together. Section 3.2.2 defines the *linkage grammar*, an attribute grammar used to represent the call structure of a system. Transitive dependences due to procedure calls are computed using the linkage grammar and are added as the final step of building a system dependence graph.

3.2.1. Procedure Calls and Parameter Passing

Extending the definition of dependence graphs to handle procedure calls requires representing the passing of values between procedures. In designing the representation of parameter passing we have three goals:

(1) It should be possible to build an individual procedure's procedure dependence graph (including the computation of data dependences) with minimal knowledge of other system components.

(2) The system dependence graph should consist of a straightforward connection of the procedure dependence graph and procedure dependence graphs.

(3) It should be possible to extract a precise interprocedural slice efficiently by traversing the graph via a procedure analogous to the procedure MarkVerticesOfSlice given in Figure 2.2.

To meet the goals outlined above, our graphs model the following slightly non-standard, two-stage mechanism for run-time parameter passing: when procedure P calls procedure Q, values are transferred from P to Q by means of intermediate temporary variables, one for each parameter. A different set of temporary variables is used when Q returns to transfer values back to P. Before the call, P copies the values of the actual parameters into the call temporaries; Q then initializes local variables from these temporaries. Before returning, Q copies return values into the return temporaries, from which P retrieves them.

This model of parameter passing is represented in procedure dependence graphs through the use of five new kinds of vertices. The locus of control for a call is represented using a *call-site* vertex; information transfer is represented using four kinds of *parameter* vertices. On the calling side, information transfer is represented by a set of *actual-in* and *actual-out* vertices. These vertices, which are control dependent on the call-site vertex, represent assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively. Similarly, information transfer in the called procedure is represented by a set of *formal-in* and *formal-out* vertices. These vertices, which are control dependent on the procedure's entry vertex, represent assignment statements that copy the initial values of the formal parameters from the call temporaries and the final values to the return temporaries, respectively.

Using this model, data dependences between procedures are limited to dependences from actual-in vertices to formal-in vertices, and from formal-out vertices to actual-out vertices. Connecting procedure dependence graphs to form a system dependence graph is straightforward, involving the addition of 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. (Call edges are a new kind of control dependence edge; parameter-in and parameter-out edges are new kinds of data dependence edges.)

Another advantage of this model is that flow dependences can be computed in the usual way, using data flow analysis on the procedure's control-flow graph. The control-flow graph for a procedure includes nodes analogous to the actual-in, actual-out, formal-in and formal-out vertices of the procedure dependence graph. A procedure's control-flow graph starts with a sequence of assignments that copy values from call temporaries to formal parameters, and ends with a sequence of assignments that copy values from formal parameters to return temporaries. Each call statement within the procedure is represented in the procedure's control-flow graph by a sequence of assignments that copy values from actual parameters to call temporaries, followed by a sequence of assignments that copy values from return temporaries to actual parameters.

An important question is *which* values are transferred from a call site to the called procedure and back again. This point is discussed further in Section 3.3.2, which presents a strategy in which the results of interprocedural data flow analysis are used to omit some parameter vertices from procedure dependence graphs. For now, we will assume that all actual parameters are copied into the call temporaries and retrieved from the return temporaries. Thus, the parameter vertices associated with a call from procedure P to procedure Q are defined as follows (G_P denotes the procedure dependence graph for P):

> In G_P, subordinate to the call-site vertex that represents the call to Q, there is an actual-in vertex and an actual-out vertex for each actual parameter a of the call to Q. These vertices are labeled "$r_{in} := a$" and "$a := r_{out}$", respectively, where r is the corresponding formal parameter name.

The parameter vertices associated with the entry to procedure Q and the return from procedure Q are defined as follows (G_Q denotes the procedure dependence graph for Q):

> For each formal parameter r of Q, G_Q contains a formal-in vertex and a formal-out vertex. These vertices are labeled "$r := r_{in}$", and "$r_{out} := r$", respectively.

Example. Figure 3.1 repeats the example system from Section 3.1 and shows the corresponding procedure dependence graphs connected with parameter-in edges, parameter-out edges, and call edges. (In Figure 3.1, as well as in the remaining figures of the dissertation, def-order edges are not shown. Edges representing control dependences are shown unlabeled; all such edges in this example would be labeled **true**.)

3.2.2. The Linkage Grammar: An Attribute Grammar that Models Procedure-Call Structure

Using the graph structure defined in the previous section, interprocedural slicing could be defined as a graph-reachability problem, and the slices obtained would be the same as those obtained using Weiser's slicing method. As explained in Section 3.1, Weiser's method does not produce as precise a slice as possible because it fails to account for the calling context of a called procedure.

Example. The problem with Weiser's method can be illustrated using the graph shown in Figure 3.1. In the graph-reachability vocabulary, the problem is that there is a path from the vertex of procedure *Main* labeled "$x_{in} := sum$" to the vertex of *Main* labeled "$i := y_{out}$", even though the value of i after the call to procedure A is independent of the value of *sum* before the call. The path is as follows:

$$
\begin{array}{llllll}
Main\text{: } "x_{in} := sum" & \rightarrow & A\text{: } "x := x_{in}" & \rightarrow & A\text{: } "a_{in} := x" & \rightarrow \quad Add\text{: } "a := a_{in}" \\
& \rightarrow & Add\text{: } "a := a + b" & \rightarrow & Add\text{: } "a_{out} := a" & \rightarrow \quad Inc\text{: } "z := a_{out}" \\
& \rightarrow & Inc\text{: } "z_{out} := z" & \rightarrow & A\text{: } "y := z_{out}" & \rightarrow \quad A\text{: } "y_{out} := y" \\
& \rightarrow & Main\text{: } "i := y_{out}" & & &
\end{array}
$$

The source of this problem is that not all paths in the graph correspond to possible execution paths (*e.g.*, the path from vertex "$x_{in} := sum$" of *Main* to vertex "$i := y_{out}$" of *Main* corresponds to procedure *Add* being called by procedure A, but returning to procedure *Increment*).

To overcome this problem, we add an additional kind of edge, called *summary* edges, to the system dependence graph to represent transitive dependences due to the effects of procedure calls. The presence of summary edges permits interprocedural slices to be computed in two passes, each of which is cast as a reachability problem. Thus, the next step in the construction of the system dependence graph is to determine such transitive dependences. For example, for the graph shown in Figure 3.1, we need an algorithm that can discover the transitive dependence from vertex "$x_{in} := sum$" of *Main* to vertex "$sum := x_{out}$" of *Main*. This dependence exists because the value of *sum* after the call to A depends on the value of *sum* before the call to A.

One's first impulse might be to compute transitive dependences due to calls by taking the transitive closure of the graph's control, flow, parameter, and call edges. However, this technique is imprecise for the same reason that transitive closure (or, equivalently, reachability) is imprecise for interprocedural slicing, namely that not all paths in the system dependence graph correspond to possible execution paths. Using transitive closure to compute the dependence edges that represent the effects of procedure calls would put in a (spurious) edge from vertex "$x_{in} := sum$" of *Main* to vertex "$i := y_{out}$" of *Main*.

For a language without recursion, this problem could be eliminated by using a separate copy of a procedure dependence graph for each call site; however, to handle a language *with* recursion, a more powerful technique is required. The technique we use involves defining an attribute grammar, called the *linkage*

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

```
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
```

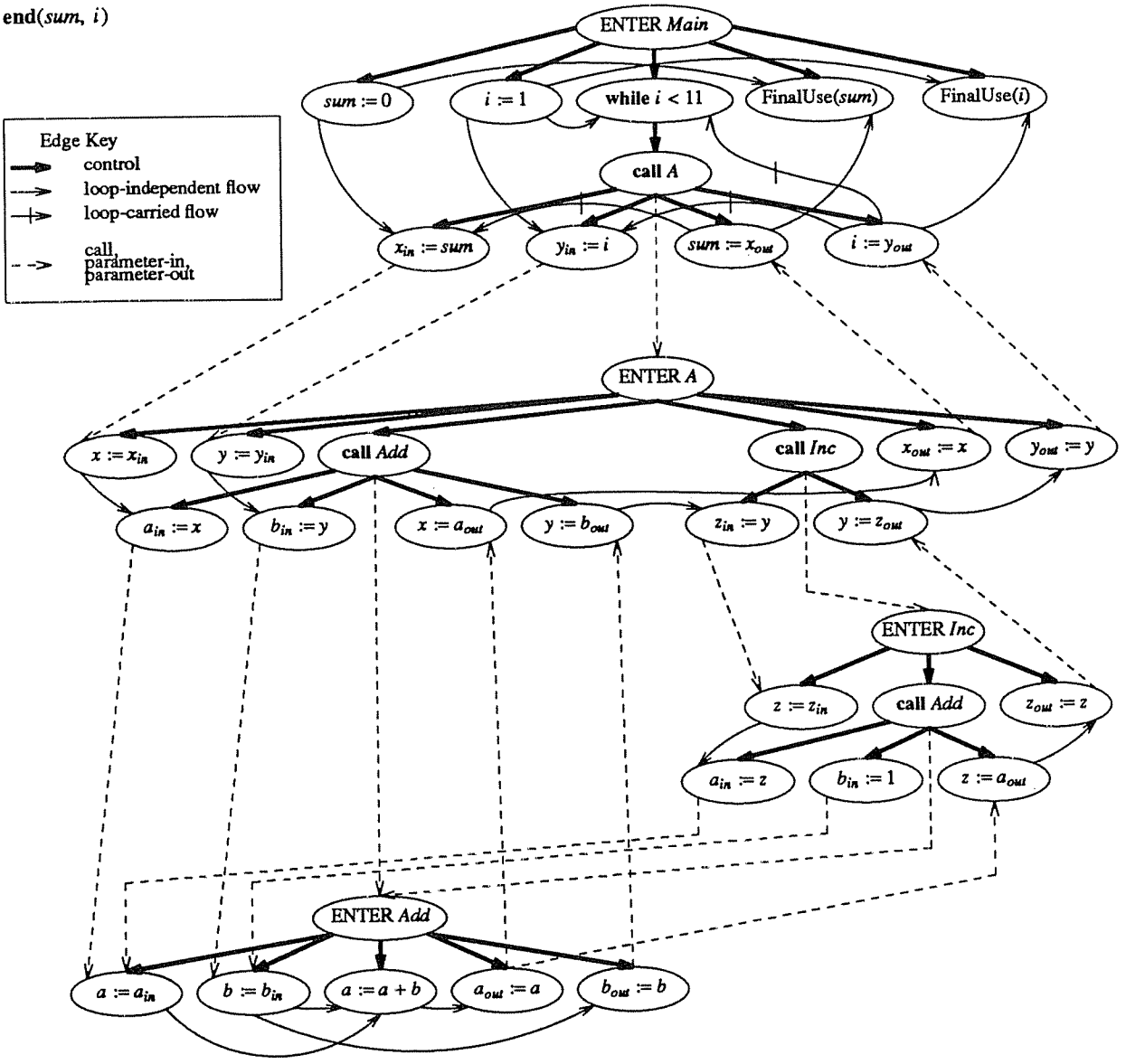


Figure 3.1. Example system and corresponding program and procedure dependence graphs connected with parameter-in, parameter-out, and call edges. Edges representing control dependences are shown (unlabeled) in boldface; edges representing intraprocedural flow dependences are shown using arcs; parameter-in edges, parameter-out edges, and call edges are shown using dashed lines.

grammar, to model the call structure of each procedure as well as the in*tra*procedural transitive flow

dependences among the procedure's parameter vertices. Interprocedural transitive flow dependences among a system dependence graph's parameter vertices are determined from the linkage grammar using a standard attribute-grammar construction: the computation of the *subordinate characteristic graphs* of the linkage grammar's nonterminals.[2]

In this section, we describe the construction of the linkage grammar and the computation of its subordinate characteristic graphs. It should be understood that the linkage grammar is used *only* to compute transitive dependences due to calls; we are not interested in the language defined by the grammar, nor in actual attribute values.

The context-free part of the linkage grammar models the system's procedure-call structure. The grammar includes one nonterminal and one production for each procedure in the system. If procedure P contains no calls, the right-hand side of the production for P is ε; otherwise, there is one right-hand-side nonterminal for each call site in P.

Example. For the example system shown in Figure 3.1, the productions of the linkage grammar are as follows:

$Main \rightarrow A$ $\qquad$ $A \rightarrow Add\ \ Increment$ $\qquad$ $Add \rightarrow \varepsilon$ $\qquad$ $Increment \rightarrow Add$

The attributes in the linkage grammar correspond to the parameters of the procedures. Procedure inputs are modeled as inherited attributes; procedure outputs are modeled as synthesized attributes. For example, the productions shown above are repeated in Figure 3.2, this time in tree form. In Figure 3.2, each nonterminal is annotated with its attributes; a nonterminal's inherited attributes are placed to its left; its synthesized attributes are placed to its right.

More formally, the program's linkage grammar has the following elements:

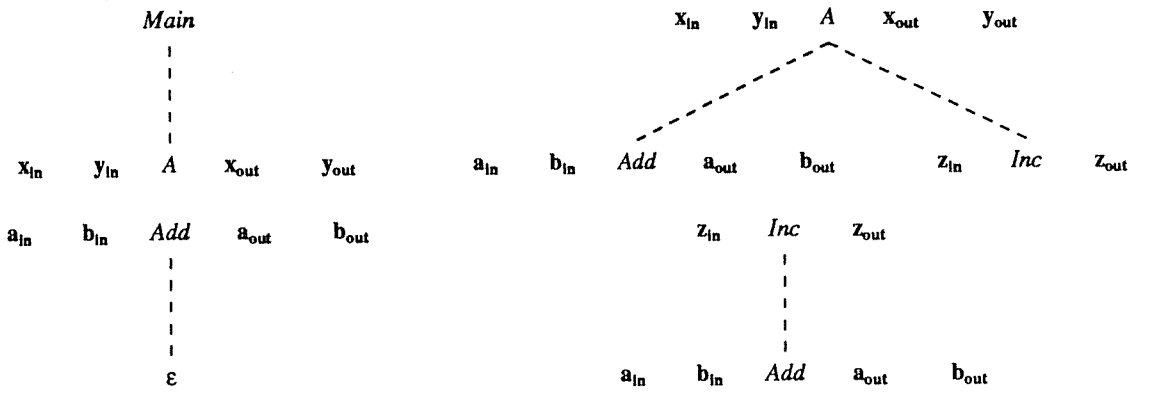


Figure 3.2. The productions of the example linkage grammar shown in tree form. Each nonterminal is annotated with its attributes; a nonterminal's inherited attributes are placed to its left; its synthesized attributes are placed to its right.

[2]A summary of attribute-grammar terminology can be found in the Appendix to Chapter 3.

(1) For each procedure P, the linkage grammar contains a nonterminal P.

(2) For each procedure P, there is a production $p: P \rightarrow \beta$, where for each call-site representing a call on procedure Q in P there is a distinct occurrence of Q in β.

(3) For each actual-in vertex of P, there is an inherited attribute of nonterminal P.

(4) For each actual-out vertex of P, there is a synthesized attribute of nonterminal P.

Attribute a of nonterminal X is denoted by "$X.a$".

Dependences among the attributes of a linkage-grammar production are used to model the (possibly transitive) intraprocedural dependences among the parameter vertices of the corresponding procedure. These dependences are computed using HPR slices of the procedure's procedure dependence graph as described in Chapter 2. For each grammar production, attribute equations are introduced to represent the intraprocedural dependences among the parameter vertices of the corresponding procedure dependence graph. For each attribute occurrence a, the procedure dependence graph is sliced with respect to the vertex that corresponds to a. An attribute equation is introduced for a so that a depends on the attribute occurrences that correspond to the parameter vertices included in the slice. More formally:

> For each attribute occurrence $X.a$ of a production p, let v be the vertex of the procedure dependence graph G_P that corresponds to $X.a$. Associate with p an attribute equation of the form $X.a = f(\ldots, Y.b, \ldots)$ where the arguments $Y.b$ to the equation consist of the attribute occurrences of p that correspond to the parameter vertices in $b^{hpr}(G_P, v)$.

Note that the actual function f on the right-hand side of the equation is completely irrelevant because the attribute grammar is *never* used for evaluation; all we need is that the equation induce the dependences described above.

Example. Figure 3.3 shows the productions of the grammar from Figure 3.2, augmented with attribute dependences. The dependences for production $Main \rightarrow A$, for instance, correspond to the attribute-definition equations

$$A.x_{in} = f1(A.x_{out}, A.y_{out}) \qquad A.y_{in} = f2(A.y_{out}) \qquad A.x_{out} = f3(A.y_{out}) \qquad A.y_{out} = f4(A.y_{out}).$$

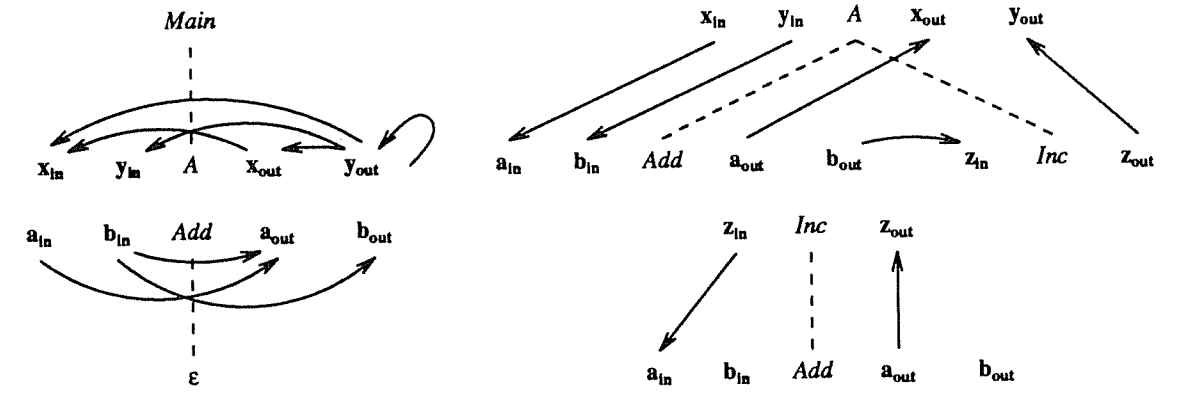


Figure 3.3. The productions of Figure 3.2, augmented with attribute dependences.

It is entirely possible that a linkage grammar will be a circular attribute grammar (*i.e.*, there may be attributes in some derivation tree of the grammar that depend on themselves); additionally, the grammar may not be well-formed (*e.g.*, a production may have equations for synthesized attribute occurrences of right-hand-side symbols). This does not create any difficulties as the linkage grammar is used *only* to compute transitive dependences and *not* for attribute evaluation.

Example. The equation $A.y_{out} = f4(A.y_{out})$ makes the example attribute grammar both circular and not well-formed. This equation is added to the attribute grammar because of the following (cyclic) path in the graph shown in Figure 3.1:

$$Main: \text{“}i := y_{out}\text{”} \quad \rightarrow \quad Main: \text{“}\textbf{while } i < 11\text{”} \quad \rightarrow \quad Main: \text{“}\textbf{call } A\text{”} \quad \rightarrow \quad Main: \text{“}i := y_{out}\text{”}$$

Transitive dependences from a call site's actual-in vertices to its actual-out vertices are computed from the linkage grammar by constructing the subordinate characteristic graphs for the grammar's nonterminals. The algorithm we give exploits the special structure of linkage grammars to compute these graphs more efficiently than can be done for attribute grammars in general. For general attribute grammars, computing the sets of possible subordinate characteristic graphs for the grammar's nonterminals may require time exponential in the number of attributes attached to some nonterminal. However, a linkage grammar is an attribute grammar of a restricted nature: for each nonterminal X in the linkage grammar, there is only one production with X on the left-hand side. Because linkage grammars are restricted in this fashion, for each nonterminal of a linkage grammar there is one subordinate characteristic graph that covers all of the nonterminal's other possible subordinate characteristic graphs. For such grammars, it is possible to give a polynomial-time algorithm for constructing the (covering) subordinate characteristic graphs.

The computation is performed by the algorithm ConstructSubCGraphs, which is a slight modification of an algorithm originally developed by Kastens to construct approximations to a grammar's transitive dependence relations [Kastens80]. The covering subordinate characteristic graph of a nonterminal X of the linkage grammar is captured in the graph $TDS(X)$ (standing for "Transitive Dependences among a Symbol's attributes"). Initially, all the *TDS* graphs are empty. The construction that builds them up involves the auxiliary graph $TDP(p)$ (standing for "Transitive Dependences in a Production"), which expresses dependences among the attributes of a production's nonterminal occurrences.

The basic operation used in ConstructSubCGraphs is the procedure "AddEdgeAndInduce($TDP(p)$, (a, b))", whose first argument is the *TDP* graph of some production p and whose second argument is a pair of attribute occurrences in p. AddEdgeAndInduce carries out three actions:

(1) The edge (a, b) is inserted into the graph $TDP(p)$.

(2) Any additional edges needed to transitively close $TDP(p)$ are inserted into $TDP(p)$.

(3) In addition, for each edge added to $TDP(p)$ by (1) or (2), (*i.e.*, either the edge (a, b) itself or some other edge (c, d) added to reclose $TDP(p)$), AddEdgeAndInduce may add an edge to one of the *TDS* graphs. In particular, for each edge added to $TDP(p)$ of the form $(X_0.m, X_0.n)$, where X_0 is the left-hand-side occurrence of nonterminal X in production p and $(X.m, X.n) \notin TDS(X)$, an edge $(X.m, X.n)$ is added to $TDS(X)$.

An edge in one of the *TDS* graphs can be *marked* or *unmarked*; the edges that AddEdgeAndInduce adds to the *TDS* graphs are unmarked.

The *TDS* graphs are generated by the procedure ConstructSubCGraphs, given in Figure 3.4, which is a slight modification of the first two steps of Kastens's algorithm for constructing a set of evaluation plans

for an attribute grammar [Kastens80]. ConstructSubCGraphs performs a kind of closure operation on the *TDP* and *TDS* graphs. Step 1 of the algorithm—the first two for-loops of ConstructSubCGraphs—initializes the grammar's *TDP* and *TDS* graphs; when these loops terminate, the *TDP* graphs contain edges representing all direct dependences that exist between the grammar's attribute occurrences, and the *TDS* graphs contain unmarked edges corresponding to direct left-hand-side-to-left-hand-side dependences in the linkage grammar's productions. Our construction of attribute equations for the linkage grammar ensures that the graph of direct attribute dependences is transitively closed; thus, at the end of Step 1, $TDP(p)$ is a transitively closed graph.

In Step 2 of ConstructSubCGraphs, the invariant for the **while**-loop is:

> If a graph $TDP(p)$ contains an edge e' that corresponds to a marked edge e in one of the *TDS* graphs, then e has been induced in all of the other graphs $TDP(q)$.

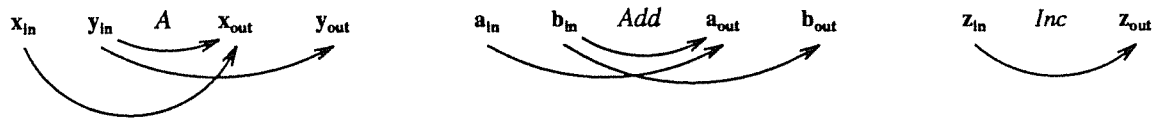
When all edges in all *TDS* graphs have received marks, the effects of all dependences have been induced in the *TDP* and *TDS* graphs. Thus, the $TDS(X)$ graphs computed by ConstructSubCGraphs are guaranteed to cover the transitive dependences among the attributes of X that exist at any occurrence of X in any derivation tree.

```
procedure ConstructSubCGraphs(L)
declare
   L: a linkage grammar
   p: a production in L
   X_i, X_j, X̂: nonterminal occurrences in L
   a, b: attributes of nonterminals in L
   X: a nonterminal in L
begin
   /* Step 1: Initialize the TDS and TDP graphs */
      for each nonterminal X in L do
         TDS(X) := the graph containing a vertex for each attribute X.b but no edges
      od
      for each production p in L do
         TDP(p) := the graph containing a vertex for each attribute occurrence X_j.b of p but no edges
         for each attribute occurrence X_j.b of p do
            for each argument X_i.a of the equation that defines X_j.b do
               Insert edge (X_i.a, X_j.b) into TDP(p)
               let X be the nonterminal corresponding to nonterminal occurrence X_j in
                  if i = 0 and j = 0 and (X.a, X.b) ∉ TDS(X) then Insert an unmarked edge (X.a, X.b) into TDS(X) fi
               ni
            od
         od
      od
   /* Step 2: Determine the sets of induced transitive dependences */
      while there is an unmarked edge (X.a, X.b) in one of the TDS graphs do
         Mark (X.a, X.b)
         for each occurrence X̂ of X in any production p do
            if (X̂.a, X̂.b) ∉ TDP(p) then AddEdgeAndInduce(TDP(p), (X̂.a, X̂.b)) fi
         od
      od
end
```

Figure 3.4. Computation of a linkage grammar's sets of *TDP* and *TDS* graphs.

Put more simply, because for each nonterminal X in a linkage grammar there is only a single production that has X on the left-hand side, the grammar only derives one tree (for a recursive grammar it will be an infinite tree). All marked edges in TDS represent transitive dependences in this tree, and thus the $TDS(X)$ graph computed by ConstructSubCGraphs represents a subordinate characteristic graph of X that covers the subordinate characteristic graph of any partial derivation tree derived from X, as desired.

Example. The nonterminals of our example grammar are shown below annotated with their attributes and their subordinate characteristic graphs.



3.2.3. Recap of the Construction of the System Dependence Graph

The system dependence graph is constructed by the following steps:

(1) For each procedure of the system, construct its procedure dependence graph.

(2) For each call site, introduce a call edge from the call-site vertex to the corresponding procedure-entry vertex.

(3) For each actual-in vertex v at a call site, introduce a parameter-in edge from v to the corresponding formal-in vertex in the called procedure.

(4) For each actual-out vertex v at a call site, introduce a parameter-out edge to v from the corresponding formal-out vertex in the called procedure.

(5) Construct the linkage grammar corresponding to the system.

(6) Compute the subordinate characteristic graphs of the linkage grammar's nonterminals.

(7) At all call sites that call procedure P, introduce summary edges corresponding to the edges in the subordinate characteristic graph for P.

Example. Figure 3.5 shows the complete system dependence graph for our example system.

3.3. Interprocedural Slicing

This section describes how to perform an interprocedural slice using the system dependence graph defined in Section 3.2. It then discusses modifications to the definition of the system dependence graph that permit more precise slicing and extend the slicing algorithm's range of applicability.

3.3.1. An Algorithm for Interprocedural Slicing

As discussed in the Section 3.1.1, the algorithm presented in [Weiser84], while safe, is not as precise as possible. The difficult aspect of interprocedural slicing is keeping track of the calling context when a slice "descends" into a called procedure.

The key element of our approach is the use of the linkage grammar's characteristic graph edges in the system dependence graph. These edges represent transitive data dependences from actual-in vertices to actual-out vertices due to procedure calls. The presence of such edges permits us to sidestep the "calling context" problem; the slicing operation can move "across" a call without having to descend into it; thus, there is no need to keep track of calling context to ensure that only legal execution paths are traversed.

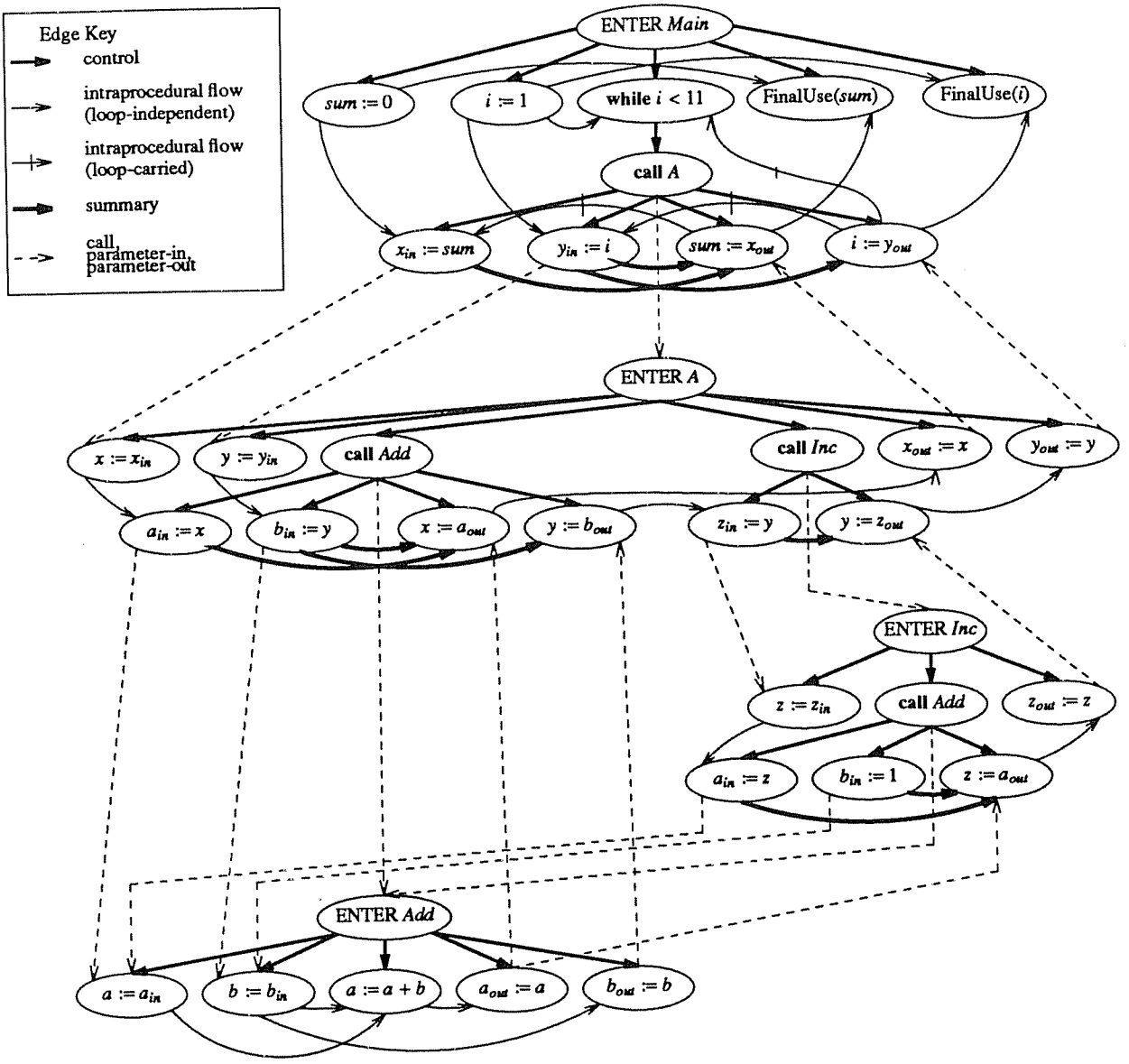


Figure 3.5. Example system's system dependence graph. Control dependences, shown unlabeled, are represented using medium-bold arrows; intraprocedural flow dependences are represented using arcs; summary edges (*i.e.*, transitive interprocedural flow dependences that correspond to subordinate characteristic graph edges) are represented using heavy bold arcs; call edges, parameter-in edges, and parameter-out edges (which connect program and procedure dependence graphs together) are represented using dashed arrows.

Our algorithm for interprocedural slicing is given in Figure 3.6, in which the computation of the slice of system dependence graph G with respect to vertex set S is performed in two linear-time passes over G. Both passes operate on the system dependence graph using essentially the method presented in Chapter 2 for performing an HPR slice—the graph is traversed 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-

```
procedure MarkVerticesOfSlice(G, S)
declare
  G: a system dependence graph
  S, S': sets of vertices in G
begin
  /* Pass 1: Slice without descending into called procedures */
    S' := MarkReachingVertices(G, S, {def-order, parameter-out})
  /* Pass 2: Slice called procedures without ascending to call sites */
    MarkReachingVertices(G, S', {def-order, parameter-in, call})
end

function MarkReachingVertices(G, V, Kinds) returns a set of vertices
declare
  G: a system dependence graph
  V: a set of vertices in G
  Kinds: a set of kinds of edges
  v, w: vertices in G
  WorkList: a set of vertices in G
begin
  WorkList := V
  while WorkList ≠ ∅ do
    Select and remove a vertex v from WorkList
    Mark v
    for each unmarked vertex w such that there is an edge w → v whose kind is not in Kinds do
      Insert w into WorkList
    od
  od
  return (the set of marked vertices in G)
end
```

Figure 3.6. The procedure MarkVerticesOfSlice marks the vertices of the interprocedural slice of G taken with respect to S. The auxiliary procedure MarkReachingVertices marks all vertices in G from which there is a path to a vertex in V along edges of kinds other than those in the set *Kinds*.

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 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 identified by Pass 1 and Pass 2, and the set of edges induced by this vertex set.

Suppose the goal is to slice system dependence graph G with respect to some vertex s in procedure P; Passes 1 and 2 can be characterized as follows:

Pass 1

Pass 1 identifies vertices that can reach s, and are either in P itself or in a procedure that calls P (either directly or transitively). Because parameter-out edges are not followed, the traversal in Pass 1 does not "descend" into procedures called by P. However, the effects of such procedures are not ignored; the presence of summary edges from actual-in to actual-out vertices (subordinate-characteristic-graph edges) permits the discovery of vertices that can reach s only through a procedure call, although the graph traversal does not actually descend into the called procedure.

Pass 2

Pass 2 identifies vertices that can reach s from procedures (transitively) called by P or from

procedures called by procedures that (transitively) call P. Because call edges and parameter-in edges are not followed, the traversal in Pass 2 does not "ascend" into calling procedures; the summary edges from actual-in to actual-out vertices make such "ascents" unnecessary.

Figures 3.7 and 3.8 illustrate the two passes of the interprocedural slicing algorithm. Figure 3.7 shows the vertices of the example system dependence graph that are marked during Pass 1 of the interprocedural slicing algorithm when the system is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. Edges "traversed" during Pass 1 are also included in Figure 3.7. Figure 3.8 adds (in

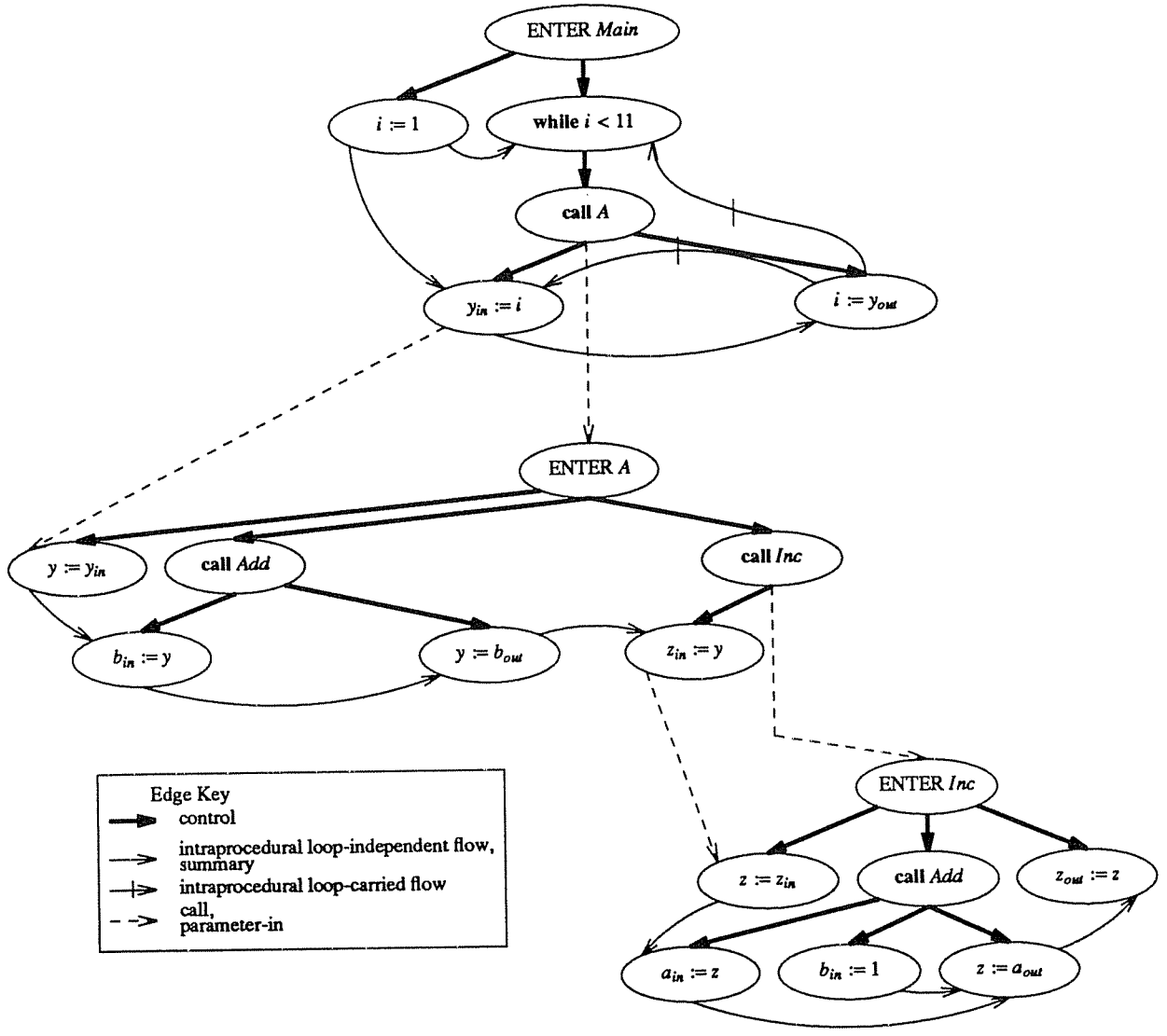


Figure 3.7. The example program's system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Pass 1 of the slicing algorithm as well as the edges traversed during this pass are shown above.

boldface) the vertices that are marked and the edges that are traversed during Pass 2 of the slice.

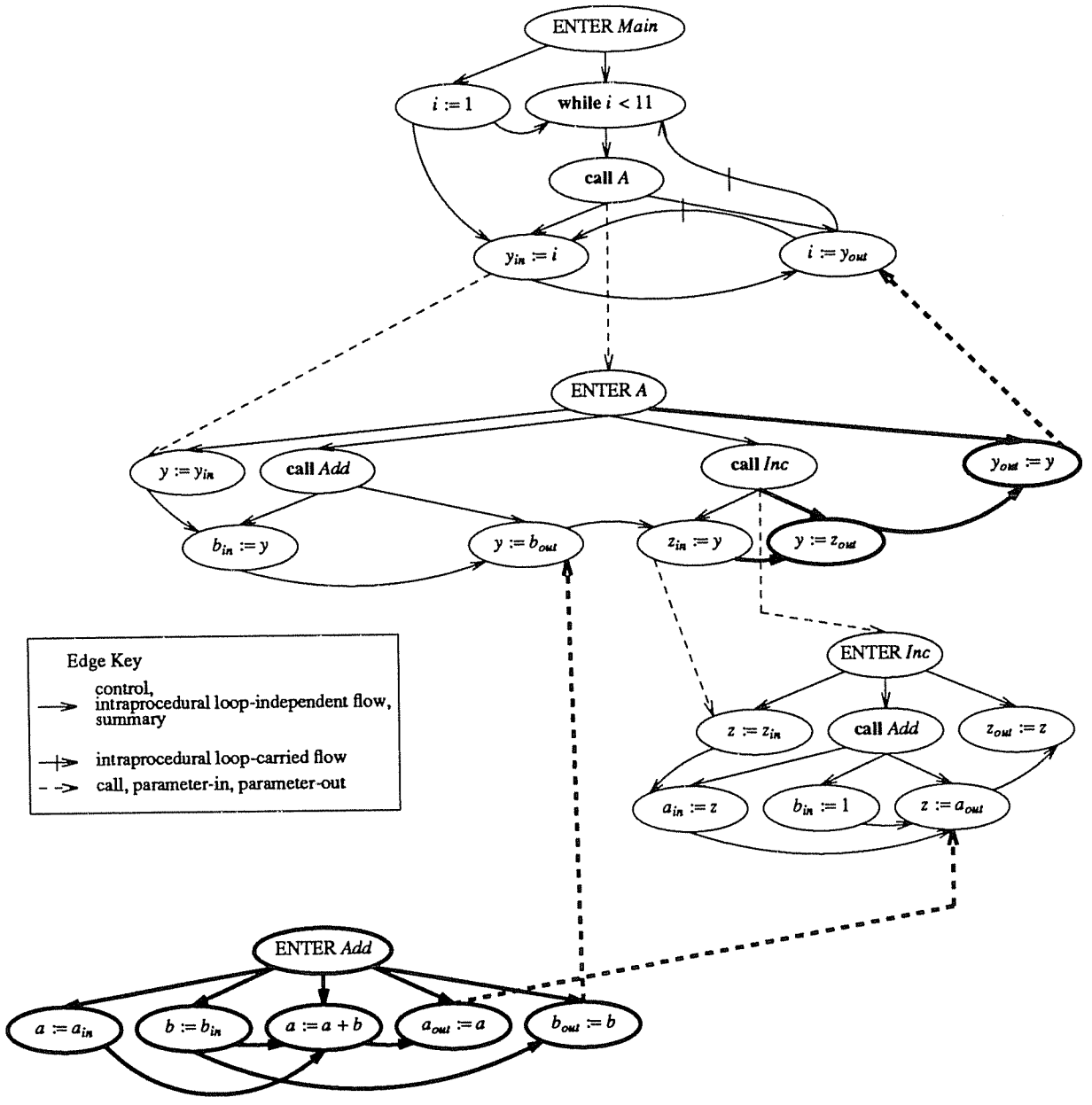


Figure 3.8. The example program's system dependence graph is sliced with respect to the formal-out vertex for parameter z in procedure *Increment*. The vertices marked by Pass 2 of the slicing algorithm as well as the edges traversed during this pass are shown above in boldface.

The result of an interprocedural slice consists of the sets of vertices identified by Pass 1 and Pass 2, and the set of edges induced by this vertex set. Figure 3.9 shows the completed example slice (excluding def-order edges).

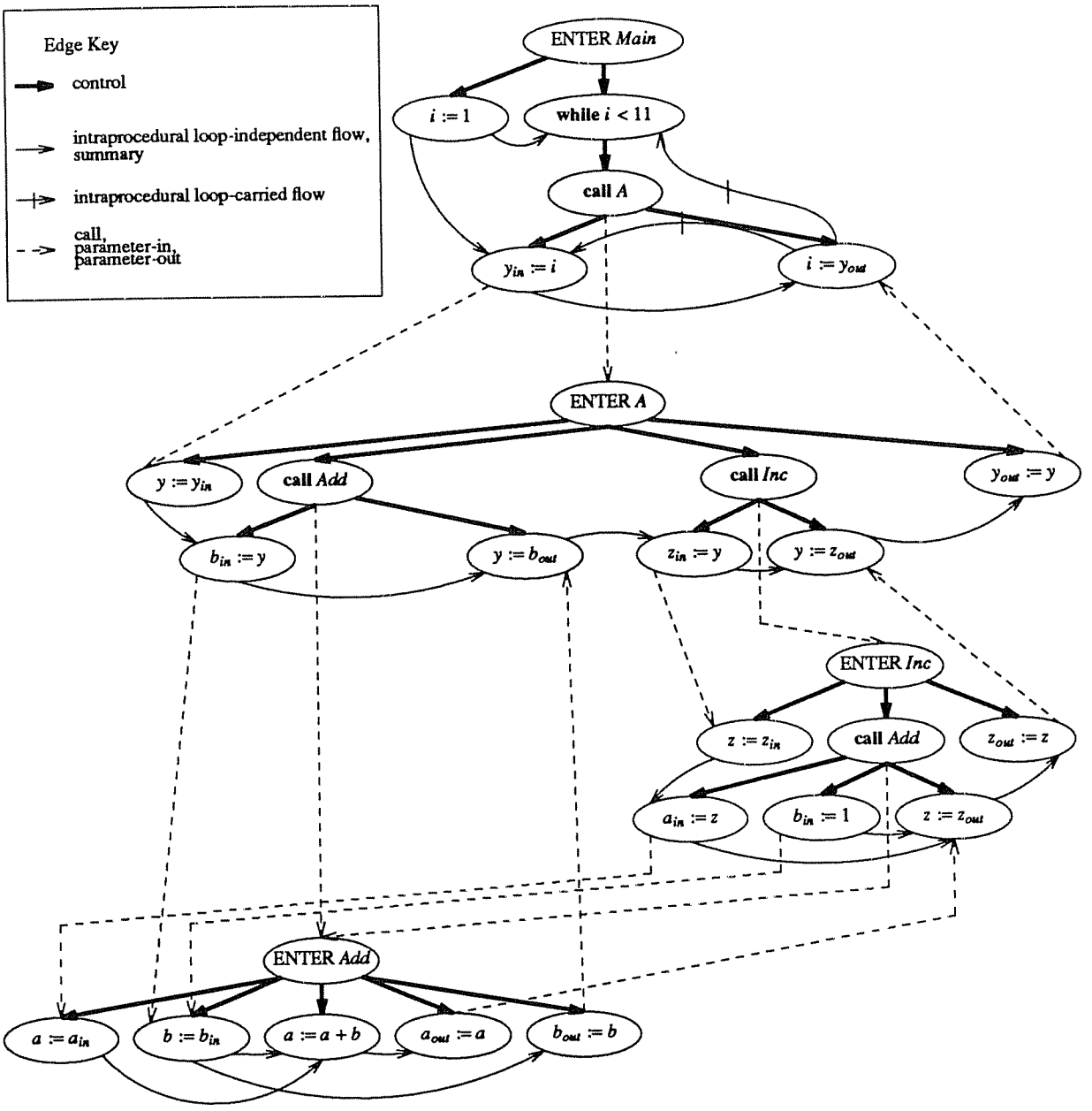


Figure 3.9. The complete slice (excluding def-order edges) of the example program's system dependence graph sliced with respect to the formal-out vertex for parameter z in procedure *Increment*.

In the following chapters of this dissertation, we use the operators *b1* and *b2* to designate the individual passes of the slicing algorithm (the "*b*" refers to the backward (target to source) traversal of edges performed by both passes of an interprocedural slice). Formally, operators *b1* and *b2* are applied to a system dependence graph (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 $SDG \times vertex\text{-}set \rightarrow SDG \times vertex\text{-}set$. In the terminology of Figure 3.6, they are defined as follows:

$$b1(G, S) \triangleq (G, \text{MarkReachingVertices}(G, S, \{\text{def-order, parameter-out}\}))$$
$$b2(G, S) \triangleq (G, \text{MarkReachingVertices}(G, S, \{\text{def-order, parameter-in, call}\})).$$

The operator *b* is used to denote the composition $b2 \circ b1$. Thus, the (full backwards) slice of graph G with respect to vertex set S is denoted by $Induce(b(G, S))$, where *Induce* is the extension to system dependence graphs of the function *Induce* defined in Section 2.2.1. Where there is no ambiguity, we omit *Induce*; for example, we use $b(G, S)$ in place of $Induce(b(G, S))$ to denote the graph resulting from the backward slice of G with respect to S.

Although operator *b* produces a (graph, vertex-set) pair, it is convenient to omit the operator that selects the vertex-set component when comparing slices or when testing whether a set of vertices is in a slice. For example, we use $b(G_1, v) \neq b(G_2, v)$ in place of $SelectVertexSet(b(G_1, v)) \neq SelectVertexSet(b(G_2, v)))$.

As mentioned in Section 3.1, an interprocedural slice, unlike an hpr slice, is not guaranteed to be feasible (*i.e.*, it may not be the system dependence graph of any system). The reason for this is that it is possible for the program elements found by operator *b* to include multiple calls on a procedure, each passing a different subset of the actual parameters. For example, the slice shown in Figure 3.10 contains two calls on procedure *P* that pass different subsets of the actual parameters.

3.3.2. Using Interprocedural Summary Information to Build Procedure Dependence Graphs

The slice shown in Figure 3.9 illustrates a shortcoming of the method for constructing procedure dependence graphs described in Section 3.2.1. The problem is that including both an actual-in and an actual-out

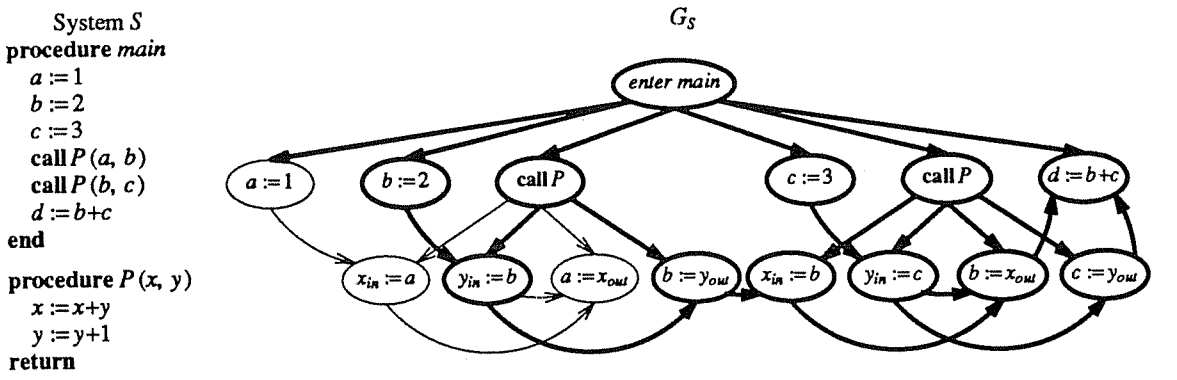


Figure 3.10. An infeasible slice. The slice of the system dependence graph for *S* with respect to the vertex labeled "*d* := *b*+*c*" is highlighted above in bold (only the procedure dependence graph for the main procedure is shown). Since this slice includes *both* actual parameters from the second call on *P* but only *one* (the second) actual parameter from the first call on *P*, the slice is infeasible (*i.e.*, no syntactically legal program has two calls on one procedure that include different numbers of parameters).

vertex for *every* argument in a procedure call can affect the precision of an interprocedural slice. The slice shown in Figure 3.9 includes the call vertex that represents the call to *Add* from *A*; however, this call does not in fact affect the value of *z* in *Increment*. The problem is that an actual-out vertex for argument *y* in the call to *Add* from *A* is included in *A*'s procedure dependence graph even though *Add* does not change the value of *y*.

To achieve a more precise interprocedural slice we use the results of interprocedural data flow analysis when constructing procedure dependence graphs in order to exclude vertices like the actual-out vertex for argument *y*. The appropriate interprocedural summary information consists of the following sets, which are computed for each procedure *P* [Banning79]:

GMOD(*P*):
the set of variables that might be *modified* by *P* itself or by a procedure (transitively) called from *P*.

GREF(*P*):
the set of variables that might be *referenced* by *P* itself or by a procedure (transitively) called from *P*.

The efficient computation of these sets is addressed in [Cooper84].

GMOD and GREF sets are used to determine which parameter vertices are included in procedure dependence graphs as follows: for each procedure *P*, *P*'s procedure dependence graph includes one formal-in vertex for each parameter of *P* and one formal-in vertex for each global variable in GMOD(*P*) ∪ GREF(*P*). It also includes one formal-out vertex for each global variable or parameter in GMOD(*P*). Similarly, for each call-site on *P*, there is one actual-in vertex for each actual parameter and one actual-in vertex for each global variable in GMOD(*P*) ∪ GREF(*P*). At each call-site, there is one actual-out vertex for each global variable or parameter in GMOD(*P*). (It is necessary to include an actual-in and a formal-in vertex for a variable *x* that is in GMOD(*P*) and is not in GREF(*P*) because there may be an execution path through *P* on which *x* is *not* modified. In this case, a slice of *P* with respect to the final value of *x* must include the initial value of *x*; thus, there must be a formal-in vertex for *x* in *P*, and a corresponding actual-in vertex at the call to *P*.)

Example. The GMOD and GREF sets for our example system are

procedure	GMOD	GREF
A	*x*, *y*	*x*, *y*
Add	*a*	*a*, *b*
Inc	*z*	*z*

Because parameter *b* is not in GMOD(*Add*), *Add*'s procedure dependence graph should not include a formal-out vertex for *b*, and the call to *Add* from *A* should not include the corresponding actual-out vertex.

Figure 3.11 shows *A*'s procedure dependence graph as it would be built using GMOD and GREF information. The actual-out vertex for argument *y* of the call to *Add* is omitted, and the flow edge from that vertex to the actual-in vertex "$z_{in} := y$" is replaced by an edge from the formal-in vertex "$y := y_{in}$" to the actual-in vertex "$z_{in} := y$". The new edge is traversed during Pass 1 of the interprocedural slice instead of the (now omitted) flow edge from "$y := a_{out}$" to "$z_{in} := y$", thus (correctly) bypassing the call to *Add* in procedure *A*.

Another advantage to using GMOD and GREF is that it increases the number of equivalent systems that have isomorphic system dependence graphs. (Recall that one of our motivating concerns from the beginning of Section 3.2 is to capture as many equivalent systems as possible in the equivalence classes determined by systems with isomorphic system dependence graphs.) For example, consider the following semantically equivalent systems *S1* and *S2* (*S2* is *S1* with the two calls to procedure *Add* reversed in order).

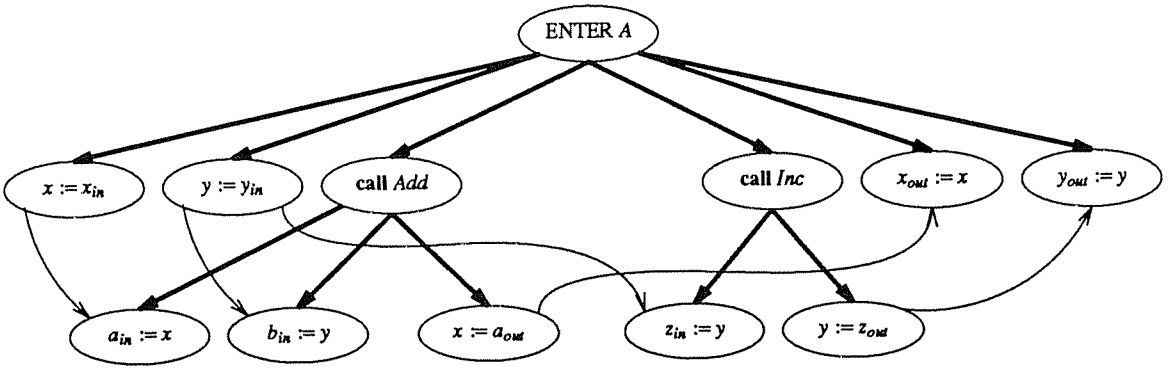


Figure 3.11. Procedure *A*'s procedure dependence graph built using interprocedural summary information. The actual-out vertex for argument y of the call to *Add* has been omitted, and the flow edge from that vertex to the vertex "$z_{in} := y$" has been replaced by an edge from the vertex "$y := y_{in}$" to the vertex "$z_{in} := y$".

System *S1*

```
procedure main
  a := 1
  b := 2
  c := 3
  call Add(a, c)
  call Add(b, c)
end

procedure Add(x, y)
  x := x+y
return
```

System *S2*

```
procedure main
  a := 1
  b := 2
  c := 3
  call Add(b, c)
  call Add(a, c)
end

procedure Add(x, y)
  x := x+y
return
```

Without the use of GMOD and GREF, *S1* and *S2* have *non*-isomorphic system dependence graphs: the edge from actual-out vertex "$c := y_{out}$" of "**call** *Add*(a, c)" to the actual-in vertex "$y_{in} := c$" of "**call** *Add*(b, c)" in *S1*'s system dependence graph is not found in *S2*'s graph, whereas *S2*'s system dependence graph contains an edge from actual-out vertex "$c := y_{out}$" of "**call** *Add*(b, c)" to the actual-in vertex "$y_{in} := c$" of "**call** *Add*(a, c)". In contrast, when using GMOD and GREF, systems *S1* and *S2* have isomorphic system dependence graphs; thus, using interprocedural data-flow information increases the number of (semantically) equivalent systems that have isomorphic system dependence graphs.

We can further increase the number of equivalent programs that have isomorphic system dependence graphs by employing more precise data-flow information. For procedure *P*, GDEF(*P*) is the set of variables that are modified on *every* control path through procedure *P* (*i.e.*, variables that must be modified whenever *P* is called); GUSE(*P*) is the set of variables referenced by procedure *P* before being defined in that procedure. Variables that are in GDEF and not in GUSE do not require the associated actual-in and formal-in vertex. However, because this information is costly to compute, we make a compromise and assume system dependence graphs are constructed using (only) GMOD and GREF.

3.3.3. Interprocedural Slicing in the Presence of Call-By-Reference Parameter Passing and Aliasing

Our definitions of system dependence graphs and interprocedural slicing have assumed that parameters are

passed by value-result.[3] The same definitions hold for call-by-reference parameter passing in the absence of aliasing; however, in the presence of aliasing, some modifications are required. This section presents two approaches for slicing with systems that use call-by-reference parameter passing and contain aliasing. The first provides a more precise slice than the second at the expense of the time and space needed to transform the original system into one that is alias free. (These costs may, in the worst case, be exponential in the maximum number of parameters passed to a procedure.) The second approach avoids this expense by making use of a generalized notion of data dependence that includes data dependences that exist under the possible aliasing patterns. After presenting these two approaches, we consider their use in multi-procedure program integration.

Procedure Specialization

Our first approach is to reduce the problem of interprocedural slicing in the presence of aliasing to the problem of interprocedural slicing in the *absence* of aliasing. The reduction is a transformation performed by simulating the calling behavior of the system (using the usual activation-tree model of procedure calls [Banning79]) to discover, for each instance of a procedure call, exactly how variables are aliased at that instance. (Although a recursive system's activation tree is infinite, the number of different alias configurations is finite; thus, only a finite portion of the activation tree is needed to compute aliasing information.)

Using alias information two transformations are performed. First, a new copy of the procedure (with a new procedure name) is created for each different alias configuration; the procedure names used at call sites are similarly adjusted. Second, within each procedure, variables are renamed so that each set of aliased variables is replaced by a single variable.

Example. Figure 3.12 shows a system with aliasing, and the portion of the system's activation tree that is used to compute alias information for each call instance. We use the notation of [Banning79], in which each node of the activation tree is labeled with the mapping from variable names to memory locations. The transformed, alias-free version of the system is shown below.

```
procedure Main                procedure P1(x, y)          procedure P2(xy)
  a := 1                        if y = 0 then               if xy = 0 then
  b := 0                          call P2(x)                  call P2(xy)
  call P1(a, b)                 fi                          fi
  c := b                        y := y + 1                  xy := xy + 1
end                           return                      return
```

The reduction may generate multiple copies of the vertex v with respect to which we are to perform a slice. If this happens, it is necessary to slice the transformed system with respect to *all* occurrences of v. This can be illustrated using the above example: if our original goal had been to slice with respect to the statement "$y := y + 1$" in procedure P, we must now slice with respect to the set of statements {"$y := y + 1$", "$xy := xy + 1$"}. The slice of the original system is obtained from the slice of the transformed system by projecting elements in the slice of the transformed system back into the original system: a vertex is in the slice of the original system if any of its copies are in the slice of the transformed system.

System Dependence Graph Generalization

Our second approach to the problem of interprocedural slicing in the presence of aliasing is to generalize the definition of the system dependence graph. This generalization represents a compromise between

[3] It is easy to modify the rules for including linkage vertices to support *value* (*in*) and *out* parameters. To pass variable x by value, neither actual-out nor formal-out vertices are included for x; to pass x as an *out* parameter, *only* actual-out and formal-out vertices are included for x.

```
procedure Main
  a := 1
  b := 0
  call P(a, b)
  c := b
end
```

```
procedure P(x, y)
  if y = 0 then
    call P(x, x)
  fi
  y := y + 1
return
```

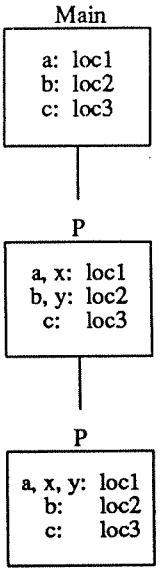


Figure 3.12. A program with aliasing and the portion of its activation tree needed to compute all alias configurations.

precision and efficiency. The transformation described above produces a precise slice, but potentially creates an exponential number of copies of each procedure; conversely, the approach described below suffers a loss of precision but maintains the polynomial construction cost of the system dependence graph.

We generalize the definition of the system dependence graph to account for aliasing by replacing the definitions for actual-out vertices, flow dependence edges, and def-order dependence edges. After giving the replacement definitions, we describe the data-flow information required to compute flow (and def-order) dependence edges under the new definitions and then describe how a slice is computed using the generalized definition of a system dependence graph. We conclude this subsection with an example that not only illustrates these techniques, but also illustrates the difference between this approach and the transformation-based approach for handling systems with alising.

DEFINITION. (Actual-out vertices in the presence of aliasing). The definition for the actual-out vertices at a call-site is unchanged except that if under the old definition two (or more) actual-out vertices represent assignments to the same variable, then these actual-out vertices are collapsed into a single actual-out vertex. (This vertex is made the target of multiple parameter-out edges: one from each of the formal-out vertices associated with the original actual-out vertices.)

DEFINITION. (Flow dependence in the presence of aliasing). A procedure dependence graph has a flow dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

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

(2) v_2 is a vertex that uses variable y.

(3) x and y are potential aliases, which includes the possibility that x and y are the same variable (in which case the edge is a "must depend" edge; otherwise, if x and y are potential aliases, but not the same variable, then the edge is a "may depend" edge).

(4) Control can reach v_2 after v_1 via a path in the control-flow graph along which there is no intervening definition of x or y.

Note that clause (4) does not exclude there being definitions of other variables that are potential aliases of x or y along the path from v_1 to v_2. An assignment to a variable z along the path from v_1 to v_2 only over-writes the contents of the memory location written by v_1 if x and z refer to the same memory location. If z is a potential alias of x, then there is only a *possibility* that x and z refer to the same memory location; thus, an assignment to z does not necessarily over-write the memory location written by v_1, and it may be possible for v_2 to read a value written by v_1.

DEFINITION. (Def-order dependence in the presence of aliasing). A procedure dependence graph has a def-order dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

(1) v_1 and v_2 define variables x_1 and x_2, respectively.

(2) x_1 and x_2 are potential aliases.

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

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

(5) v_1 occurs before v_2 in an in-order traversal of the procedure's abstract syntax tree.

The data-flow information required to compute flow (and def-order) dependence edges in the presence of aliasing is an extension of traditional reaching-definition information. Traditionally, for each definition, a pair consisting of the point at which the definition was made and the variable defined is recorded. To handle aliasing each pair is extended to a triple, which includes the set of potential aliases of the defined variable. For example, if y and z are potential aliases of x then a definition of x at point 1 would produce the triple $<1, x, \{x, y, z\}>$ (x is included in the third component because the alias relation is reflexive). As with the computation of traditional reaching-definitions, a subsequent definition of x would replace this triple. Furthermore, a definition of y or z would remove y or z, respectively, from the alias set (but not remove the triple).

For each procedure P, reaching-definition triples are computed from P^A: an augmented version of P created by adding to P assignment statements for the assignments represented by actual-in, actual-out, formal-in, and formal-out vertices in P's procedure dependence graph. P^A models these assignments with multiple assignment statements (*e.g.*, *variable*, *variable* $\cdots$:= *expression*, *expression* $\cdots$) to avoid some problems that would be introduced if an arbitrary order were used for the corresponding sequence of simple assignment statements.

The flow equation for multiple assignment statement "$x, y := \exp_1, \exp_2$" (which can be generalized to more than two variables) performs the following actions: (1) remove all triples containing x or y as the defined variable (second component), (2) create the triples $<p_1, x, aliases(x)>$ and $<p_2, y, aliases(y)>$, and (3) remove x and y from the alias sets (third component) of all triples, including those added in step (2). In action (2), points p_1 and p_2 are the points associated with the two vertices represented by the multiple assignment statement; if however these points correspond to a single actual-out vertex (because of the collapse defined above) then the same point is used in both triples (*e.g.*, point 7 in Figure 3.13).

Example. Figure 3.13 shows the computation of reaching definitions in the presence of aliasing for a slightly modified version of procedure P from Figure 3.12. This computation is straightforward except perhaps the reintroduction of x into the triple $<3, y, \{y\}>$ after point 8. In this example, and in general, the definitions reaching the end of a conditional (*e.g.*, point 8) are a combination of the definitions reaching the end of the **true** and **false** branches of the conditional. In Figure 3.13, the triple $<3, y, \{y\}>$ reaches the end of the **true** branch; however, the triple $<3, y, \{x, y\}>$ reaches the end of the **false** branch;

Procedure P	points (in P^A)	P^A	Active definitions immediately after each program point (for variables x and y only).
procedure $P(x, y)$		**procedure** P'	
	1, 2	$x, y := x_{in}, y_{in}$	<1, x, ∅>, <2, y, ∅>
$y := y*2$	3	$y := y*2$	<1, x, ∅>, <3, y, { x, y }>
	4	**if** $y = 0$ **then**	<1, x, ∅>, <3, y, { x, y }>
if $y = 0$ **then**	5	$x_{in} := x$	<1, x, ∅>, <3, y, { x, y }>
call $P(x, x)$	6	$y_{in} := x$	<1, x, ∅>, <3, y, { x, y }>
	7	$x, x := x_{out}, y_{out}$	<7, x, { y }>, <3, y, { y }>
	8	**fi**	<1, x, ∅>, <7, x, { y }>, <3, y, { x, y }>
fi	9	$y := y+1$	<1, x, ∅>, <7, x, ∅>, <9, y, { x, y }>
$y := y+1$	10	$x_{out} := x$	<1, x, ∅>, <7, x, ∅>, <9, y, { x, y }>
	11	$y_{out} := y$	<1, x, ∅>, <7, x, ∅>, <9, y, { x, y }>
		return	
return			

Figure 3.13. This figure adds the statement "$y := y*2$" to procedure P from Figure 3.12 and shows procedure P^A, the augmented version of P. The rightmost column gives the reaching-definition triples for the variables x and y after each point in P^A. Notice that the first multiple-assignment statement, "$x, y := x_{in}, y_{in}$", corresponds to P's two formal-in vertices and therefore is associated with two program points (points 1 and 2) while the second multiple-assignment statement, "$x, x := x_{out}, y_{out}$", corresponds to a single (collapsed) actual-out vertex and is therefore associated with a single program point (point 7).

combining these triples produces <3, y, { x, y }> at the join point of the conditional (*i.e.* point 8).

Given the generalized definition for the system dependence graph, the interprocedural slice of a system dependence graph is computed by the same two-pass algorithm used to compute the interprocedural slice of a system in the absence of aliasing. The data dependences in a procedure provide a safe approximation to the true dependences required for each alias configuration. However, because these edges cover all possible alias configurations, the resulting slice may contain unnecessary program elements.

Example. A comparison of the two techniques for handling aliasing can be made by considering the example shown in Figure 3.14. This example illustrates the conservative approximation made by the second approach which includes the statement "$x := 1$" in any slice that contains the statement "$z := z+1$" even though, given the aliasing configurations in which P can be called, the value produced by "$x := 1$" is never used by "$z := z+1$" (in the aliasing configuration defined by the first call there are no aliases and in the aliasing configuration defined by the second call y is also an alias of x and z and therefore the assignment to y overwrites the value produced by "$x := 1$" before it can be used at "$z := z+1$"). Statement "$x := 1$" is included in any slice that contains the statement "$z := z+1$" because there is a flow edge from "$x := 1$" to

procedure *main*	**procedure** $P(x, y, z)$
call $P(a, b, c)$	$x := 1$
call $P(d, d, d)$	$y := 2$
end	$z := z+1$
	return

Figure 3.14. An illustration of the imprecision of the generalized definition for the system dependence graph .

"$z := z+1$" (this edge exists because x and z are potential aliases and a path exists from "$x := 1$" to "$z := z+1$" that is free of assignments to x and z). Conversely, for this example, the transformation-based approach correctly omits the statement "$x := 1$" from any slice that includes "$z := z+1$".

Integration in the Presence of Aliasing

Because our ultimate goal is to define a multi-procedure program-integration algorithm, it is natural to ask how integration would be carried out using either of the two techniques described above for representing aliasing in system dependence graphs. One possibility is to extend the integration algorithm given in Chapter 5 to operate on the generalized definition of the system dependence graph that contains edges fort all possible aliasing situations (*i.e.*, the second of the two proposals described above). In other words, the same operations, as described in Chapter 5, would be carried out on the generalized system dependence graphs for *Base*, *A*, and *B*. As with slicing, this should provide a safe, conservative integration algorithm that may spuriously report interference in situations where, using either of the transformation-based approaches described below, no interference is reported).

There are two possible transformation-based approaches to integration in the presence of aliasing. The first uses transformations to determine the slices that make up the integrated system (as with the HPR algorithm, the integrated system is the union of slices of *Base*, *A*, and *B*). The second integrates the transformed versions of the original systems. This requires *normalizing* the transformations of *Base*, *A*, and *B* such that if two call-sites must call different copies of procedure *P* in one of *Base*, *A*, or *B* then they also call different copies of *P* in the other two systems. Because multiple copies of a procedure are allowed, integration is more likely to succeed using this approach; however, the resulting integrated program may contain multiple different copies of each procedure. Further investigation is needed to determine how to (and if it is desirable to) combine these copies into a single integrated version of the procedure.

Although any of the above approaches to integrating programs in the presence of aliasing could in theory be used to integrate systems with call-by-reference parameters and aliasing, the remainder of this dissertation deals only with value-result parameters where the aliasing problem does not arise.

3.3.4. Forward Slicing

Recall that *forward slicing* is the dual of (backward) slicing: whereas the *slice* of a program with respect to a program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p, the *forward slice* of a program with respect to a program point p and variable x consists of all statements and predicates of the program that might be affected by the value of x at point p. An algorithm for *forward* interprocedural slicing can be defined on system dependence graphs, using the same concepts employed for (backward) interprocedural slicing. As before, the key element is the use of the linkage grammar's characteristic graph edges in the system dependence graph to represent transitive dependences from actual-in vertices to actual-out vertices due to the effects of procedure calls.

Our algorithm for forward interprocedural slicing is given in Figure 3.15, in which the computation of the forward slice of system dependence graph G with respect to vertex set S is performed in two passes. The traversal in Pass 1 follows flow edges, control edges, and parameter-out edges, but does *not* follow call edges, def-order edges, or parameter-in edges. Because call edges and parameter-in edges are not followed, the traversal in Pass 1 does not descend into called procedures. The traversal in Pass 2 follows flow edges, control edges, call edges, and parameter-in edges, but does *not* follow def-order edges or parameter-out edges. Because parameter-out edges are not followed, the traversal in Pass 2 does not ascend into calling procedures.

```
procedure MarkVerticesOfForwardSlice(G, S)
declare
  G: a system dependence graph
  S, S': sets of vertices in G
begin
  /* Pass 1: Slice forward without descending into called procedures */
    S' := MarkVerticesReached(G, S, {def-order, parameter-in, call})
  /* Pass 2: Slice forward into called procedures without ascending to call sites */
    MarkVerticesReached(G, S', {def-order, parameter-out})
end

function MarkVerticesReached(G, V, Kinds) returns a set of vertices
declare
  G: a system dependence graph
  V: a set of vertices in G
  Kinds: a set of kinds of edges
  v, w: vertices in G
  WorkList: a set of vertices in G
begin
  WorkList := V
  while WorkList ≠ ∅ do
    Select and remove a vertex v from WorkList
    Mark v
    for each unmarked vertex w such that there is an edge v → w whose kind is not in Kinds do
      Insert w into WorkList
    od
  od
  return (the set of marked vertices in G)
end
```

Figure 3.15. The procedure MarkVerticesOfForwardSlice marks the vertices in the forward interprocedural slice of G with respect to S. The auxiliary procedure MarkVerticesReached marks all vertices in G to which there is a path from a vertex in V along edges of kinds other than those in the set *Kinds*.

Just as the operators *b1* and *b2* designate the individual passes of the backward slicing algorithm, the operators *f1* and *f2* designate the individual passes of the forward slicing algorithm. Formally, operators *f1* and *f2* are applied to a system dependence graph (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 $SDG \times vertex\text{-}set \rightarrow SDG \times vertex\text{-}set$. In the terminology of Figure 3.15, they are defined as follows:

$$f1(G, S) \triangleq (G, \text{MarkVerticesOfForwardSlice}(G, S, \{\text{def-order, parameter-in, call}\}))$$
$$f2(G, S) \triangleq (G, \text{MarkVerticesOfForwardSlice}(G, S, \{\text{def-order, parameter-out}\})).$$

Again, just as operator b is used to denote the composition $b2 \circ b1$, the operator f is used to denote the composition $f2 \circ f1$. Thus, the full forward slice of G with respect to vertex set S is denoted by $Induce(f(G, S))$. Finally, as with hpr slices, a forward slice of a system's system dependence graph may be infeasible (*i.e.*, it may not correspond to the system dependence graph of any system).

3.3.5. Slicing Partial System Dependence Graphs

The interprocedural slicing algorithm presented above is designed to be applied to a complete system dependence graph. In this section we discuss how to slice *incomplete* system dependence graphs.

The need to handle incomplete systems arises, for example, when slicing a program that calls a library procedure that is not itself available, or when slicing programs under development. In the first case, the missing components are procedures that are called by the incomplete system; in the second case, the missing components can either be not-yet-written procedures called by the incomplete system (when the program is developed top-down), or possible calling contexts (when the program is developed bottom-up).

In either case, information about the possible effects of missing calls and missing calling contexts is needed to permit slicing. This information takes the form of (safe approximations to) the subordinate characteristic graphs for missing called procedures and the superior characteristic graphs for missing calling contexts.

When no information about missing program components is available, subordinate characteristic graphs in which there is an edge from every inherited attribute to every synthesized attribute, and superior characteristic graphs in which there is an edge from every synthesized attribute to every other attribute (including the other synthesized attributes), must be used. This is because the slice of the incomplete system should include all vertices that could be included in the slice of some "completed" system, and it is always possible to provide a call or a calling context that corresponds to the graphs described above.

For library procedures, it would be possible to provide precise subordinate characteristic graphs even when the procedures themselves are not provided. For programs under development, it might be possible to compute characteristic graphs, or at least better approximations to them than the worst-case graphs, given specifications for the missing program components.

3.4. Related Work

In recasting the interprocedural slicing problem as a reachability problem in a graph, we are following the example of [Ottenstein84], which does the same for intraprocedural slicing. The reachability approach is conceptually simpler than the data-flow equation approach used in [Weiser84] and is also much more efficient when more than one slice is desired.

The recasting of the problem as a reachability problem does involve some loss of generality; rather than permitting a program to be sliced with respect to program point p and an *arbitrary* variable, a slice can only be taken with respect to a variable that is defined or used at p. For such slicing problems the interprocedural slicing algorithm presented in this chapter is an improvement over Weiser's algorithm because our algorithm is able to produce a more precise slice than the one produced by Weiser's algorithm. However, the extra generality is not the source of the imprecision of Weiser's method; as explained in the Introduction and in Section 3.2, the imprecision of Weiser's method is due to the lack of a mechanism to keep track of the calling context of a called procedure.

After the initial publication of our interprocedural-slicing algorithm [Horwitz88a], a different technique for computing interprocedural slices was presented by Hwang, Du, and Chou [Hwang88]. The slicing algorithm presented in [Hwang88] computes an answer that is as precise as our algorithm, but differs significantly in how it handles the calling-context problem. The algorithm from [Hwang88] constructs a *sequence* of slices of the system—where each slice of the sequence essentially permits there to be one additional level of recursion—until a fixed-point is reached (*i.e.*, until no further elements appear in a slice that uses one additional level of recursion). Thus, each slice of the sequence represents an approximation to the final answer. During each of these slice approximations, the algorithm uses a stack to keep track of the calling context of a called procedure. In contrast, our algorithm for interprocedural slicing is based on a two-pass process for propagating marks on the system dependence graph. In Pass 1 of our algorithm, the presence of the linkage grammar's subordinate-characteristic-graph edges (representing transitive dependences due to the effects of procedure calls) permits the entire effect of a call to be accounted for by a single backward step over the call site's subordinate-characteristic-graph edges.

Hwang, Du, and Chou do not include an analysis of their algorithm's complexity in [Hwang88], which makes a direct comparison with our algorithm difficult; however, there are several reasons why our algorithm may be more efficient. First, the algorithm from [Hwang88] computes a sequence of slices, each of which may involve re-slicing a procedure multiple times; in contrast, through its use of marks on system-dependence-graph vertices, our algorithm processes no vertex more than once during the computation of a slice. Second, if one wishes to compute multiple slices of the same system, our approach has a significant advantage. The system dependence graph (with its subordinate-characteristic-graph edges) need be computed only once; each slicing operation can use this graph, and the cost of each such slice is linear in the size of the system dependence graph. In contrast, the approach of [Hwang88] would involve finding a new fixed point (a problem that appears to have complexity comparable to the computation of the subordinate characteristic graphs) for each new slice.

In [Myers81], Myers presents algorithms for a specific set of interprocedural data flow problems, all of which require keeping track of calling context; however, Myers's approach to handling this problem differs from ours. Myers performs data flow analysis on a graph representation of the program, called a *super graph*, which is a collection of control-flow graphs (one for each procedure in the program), connected together by call and return edges. The information maintained at each vertex of the super graph includes a *memory component*, which keeps track of calling context (essentially by using the name of the call site). Our use of the system dependence graph permits keeping track of calling context while propagating simple marks rather than requiring the propagation of sets of names.

It is no doubt possible to formulate interprocedural slicing as a data flow analysis problem on a super graph, and to solve the problem using an algorithm akin to those described by Myers to account correctly for the calling context of a called procedure. As in the comparison with [Hwang88], our algorithm has a significant advantage when one wishes to compute multiple slices of the same system. Whereas the system dependence graph can be computed once and then used for each slicing operation, the approach postulated above would involve solving a new data flow analysis problem from scratch for each slice.

The vertex-reachability approach we have used here has some similarities to a technique used in [Kou77], [Callahan88], and [Cooper88] to transform data flow analysis problems to vertex-reachability problems. In each case a data flow analysis problem is solved by first building a graph representation of the program, and then performing a reachability analysis on the graph, propagating simple marks rather than, for example, sets of variable names. One difference between the interprocedural slicing problem and the problems addressed by the work cited above, is that interprocedural slicing is a "demand problem" [Babich78] whose goal is to determine information concerning a specific set of program points rather than an "exhaustive problem" in which the goal is to determine information for all program points.

Appendix to Chapter 3: Attribute Grammars and Attribute Dependences

An attribute grammar is a context-free grammar extended by attaching *attributes* to the terminal and nonterminal symbols of the grammar, and by supplying *attribute equations* to define attribute values [Knuth68]. In every production $p: X_0 \rightarrow X_1, ..., X_k$, each X_i denotes an *occurrence* of one of the grammar symbols; associated with each such symbol occurrence is a set of *attribute occurrences* corresponding to the symbol's attributes.

Each production has a set of attribute equations; each equation defines one of the production's attribute occurrences as the value of an *attribute-definition function* applied to other attribute occurrences in the production. The attributes of a symbol X are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes.

An attribute grammar is *well formed* when the terminal symbols of the grammar have no synthesized attributes, the root nonterminal of the grammar has no inherited attributes, and each production has exactly one attribute equation for each of the left-hand-side nonterminal's synthesized attribute occurrences and for each of the right-hand-side symbols' inherited attribute occurrences. (The grammars that arise in this dissertation are potentially *not* well formed in that a production may have equations for synthesized attribute occurrences of right-hand-side symbols. The reason that this does not cause problems is that the "linkage grammar" of the interprocedural slicing algorithm is used *only* to compute transitive dependences due to calls; we are not interested in the language defined by the grammar, nor in actual attribute values.)

A derivation tree node that is an instance of symbol X has an associated set of *attribute instances* corresponding to the attributes of X. An *attributed tree* is a derivation tree together with an assignment of either a value or the special token **null** to each attribute instance of the tree.

Ordinarily, although not in this dissertation, one is interested in analyzing a string according to its attribute-grammar specification. To do this, one first constructs the string's derivation tree with an assignment of **null** to each attribute instance, and then evaluates as many attribute instances as possible, using the appropriate attribute equation as an assignment statement. The latter process is termed *attribute evaluation.*

Functional dependences among attribute occurrences in a production p (or attribute instances in a tree T) can be represented by a directed graph, called a *dependence graph*, denoted by $D(p)$ (respectively, $D(T)$) and defined as follows:

(1) For each attribute occurrence (instance) b, the graph contains a vertex b'.

(2) If attribute occurrence (instance) b appears on the right-hand side of the attribute equation that defines attribute occurrence (instance) c, the graph contains the edge $b' \rightarrow c'$.

An attribute grammar that has a derivation tree whose dependence graph contains a cycle is called a *circular* attribute grammar. (The grammars that arise in this dissertation can be circular grammars.)

A node's *subordinate* and *superior characteristic graphs* provide a convenient representation of transitive dependences among the node's attributes. (A *transitive dependence* exists between attributes that are related in the transitive closure of the tree's attribute dependence relation, or, equivalently, that are connected by a directed path in the tree's dependence graph.) The vertices of the characteristic graphs at node r correspond to the attributes of r; the edges of the characteristic graphs at r correspond to transitive dependences among r's attributes.

The subordinate characteristic graph at r is the projection of the dependences of the subtree rooted at r onto the attributes of r. To form the superior characteristic graph at node r, we imagine that the subtree rooted at r has been pruned from the derivation tree, and project the dependence graph of the remaining tree onto the attributes of r. To define the characteristic graphs precisely, we make the following

definitions:

(1) Given a directed graph $G = (V, E)$, a *path* from vertex a to vertex b is a sequence of vertices, $[v_1, v_2, ..., v_k]$, such that $a = v_1$, $b = v_k$, and $\{ (v_i, v_{i+1}) \mid i = 1, ..., k-1 \} \subseteq E$.

(2) Given a directed graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of G onto V' is defined as

$G//V' = (V', E')$

where $E' = \{(v, w) \mid v, w \in V'$ and there exists a path $[v = v_1, v_2, ..., v_k = w]$ in G such that $v_2, ..., v_{k-1} \notin V'\}$. (That is, $G//V'$ has an edge from $v \in V'$ to $w \in V'$ when there exists a path from v to w in G that does not pass through any other elements of V'.)

The subordinate and superior characteristic graphs of a node r, denoted $r.C$ and $r.\overline{C}$, respectively, are defined formally as follows: Let r be a node in tree T, let the subtree rooted at r be denoted T_r, and let the attribute instances at r be denoted $A(r)$, then the subordinate and superior characteristic graphs at r satisfy

$r.C = D(T_r)//A(r)$
$r.\overline{C} = (D(T) - D(T_r))//A(r)$.

A characteristic graph represents the projection of attribute dependences onto the attributes of a single tree node; consequently, for a given grammar, each graph is bounded in size by some constant.

CHAPTER 4

A MODEL FOR MULTI-PROCEDURE INTEGRATION

In this chapter we consider two straightforward extensions of the HPR algorithm to handle multi-procedure integration. These two algorithms are strawmen that are used to motivate the need for more refined techniques. The first algorithm, discussed in Section 4.1, fails to satisfy Property (2) of the integration model of Section 1.1; 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. The second algorithm, discussed in Section 4.2, does satisfy Property (2) of the integration model, but is unacceptable because it reports interference when an intuitively acceptable integrated program exists. The latter example leads us to reformulate the integration model to capture better the goals of multi-procedure integration; the reformulated model is discussed in Section 4.3.

4.1. Integration of Separate Procedures

Our first candidate algorithm for multi-procedure integration applies the HPR algorithm separately to each of the procedures that make up a system (*i.e.*, for each procedure *P* in one or more of *Base*, *A*, and *B*, variant *A* 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 1.1. This is illustrated by the example shown in Figure 4.1, where the program labeled "Integrated System" is the result when the HPR algorithm is used to integrate the three versions of *Main* and to integrate the three versions of *P*. Although programs *Base*, *A*, and *B* in Figure 4.1 all terminate normally, the integrated program terminates

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** $P(x)$
		$\boxed{y := 1 / x}$	$y := 1 / x$
end(x)	**end**(x)	**end**(x)	**end**(x)
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 4.1. 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 1.2: although programs *Base*, *A*, and *B* all terminate normally, the integrated program terminates abnormally with a division-by-zero error. (The boxes indicate the modifications made to variants *A* and *B*.)

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.

4.2. 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 Chapter 2, except that each *intra*procedural slicing operation is reinterpreted as an *inter*procedural slicing operation.

Although this reinterpretation does yield a multi-procedure integration algorithm that satisfies the integration model of Section 1.1, 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 4.2, 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 system dependence graph G_M. However, because *both* slices are included in G_M, they are both "corrupted" in G_M; that is, although b(A, { $x := x + 1$ }) and b(B, { $x := x + 1$ }) are sub-*graphs* of G_M, neither b(A, { $x := x + 1$ }) nor b(B, { $x := x + 1$ }) is a *slice* of G_M. For example, the slice b(B, { $x := x + 1$ }) includes the actual-in vertex labeled "$x_{in} := b$" for the second call on *Incr*, which has an incoming dependence edge from statement "$b := 4$." The slice b(A, { $x := x + 1$ }) also includes the actual-in vertex labeled "$x_{in} := b$" for the second call on *Incr*, but with an incoming dependence edge from "$b := 2$." Therefore, in the slice b(G_M, { $x := x + 1$ }), the actual-in vertex labeled "$x_{in} := b$" has incoming dependence edges from *both* "$b := 4$" and "$b := 2$." Consequently, graph G_M fails the interference test and the integration algorithm reports interference.

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)
end(a, b)	**end**(a, b)	**end**(a, b)	**end**(a, b)
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 4.2. 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, a direct extension of the HPR algorithm that uses *inter*procedural slicing operations in place of *intra*procedural slicing operations would report interference on this example. (The boxes indicate the modifications made to variants A and B.)

Further examination of the example in Figure 4.2 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 4.2; 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 4.2 fails to meet Property (2) of the integration model of Section 1.1 (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 1.1, 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.

4.3. 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. In the presence of recursion, roll-out leads to an *infinite* program. (As defined in Chapter 7, 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.

Roll-out can be applied to an individual procedure or to all the procedures in a system. We use the notation *roll-out*(*A*, *Main*) to denote the single (possibly infinite) program produced by repeatedly expanding call-sites in procedure *Main* of system *A* and *roll-out*(*A*) to denotes the *set* of (possibly infinite) programs produced by rolling-out every procedure in *A*. In what follows, we present two revised versions of the program-integration model. Both versions make use of roll-out; however, the first version only requires that certain properties hold for *roll-out*(*Base*, *Main*), *roll-out*(*A*, *Main*), and *roll-out*(*B*, *Main*), while the second version requires that these properties hold for all of the procedures in the sets *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*).

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 the rolled-out program, different occurrences of a given component of some 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*, *Main*) than in *roll-out*(*Base*, *Main*), while another occurrence of the same component has a different behavior in *roll-out*(*B*, *Main*) than in *roll-out*(*Base*, *Main*), without there being interference.

Example. For each of the systems listed in Figure 4.2, the (finite) program obtained by applying roll-out to the system's main procedure contains 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 main procedure of the system labeled "Proposed Integrated System" in Figure 4.2, 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 "$x := x + 1$."

To justify the use of *roll-out* in our new integration model it is necessary to demonstrate that *roll-out* is a semantics preserving transformation. That is, when system *S* and program *roll-out*(*S*, *Main*) are applied to the same initial state, the sequences of values produced by the statements of *S* and the occurrences of these statements in *roll-out*(*S*, *Main*) are the same. Furthermore, the context in which a sequence is produced by a statement *s* in *S* uniquely identifies the occurrence of *s* that produces the identical sequence in *roll-out*(*S*, *Main*). For example, referring to system *Base* of Figure 4.2, in the context corresponding to "**call** *incr* (*a*)" statement "$x := x+1$" produces the sequence <2>; this context (uniquely) identifies the first occurrence of "$x := x+1$" in *roll-out*(*Base*, *Main*), which also produces the sequence <2>. We show in Chapter 7 that *roll-out* is a semantics preserving transformation.

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 (possibly infinite) 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 system dependence graphs). 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:

Revised Model of Program Integration, Version 1

(1) At the conceptual level, we retain the integration model of Section 1.1: let $M^{\infty} = I^{\infty}$(*roll-out*(*A*, *Main*), *roll-out*(*Base*, *Main*), *roll-out*(*B*, *Main*)). Then *roll-out*(*Base*, *Main*), *roll-out*(*A*, *Main*), *roll-out*(*B*, *Main*), and M^{∞} must meet the three properties of the integration model of Section 1.1 (with infinite programs now permitted).
(2) At the concrete level, we impose the following conditions:
    (i) Programs are finite and must be written in the simplified programming language described in Property (1) of the integration model of Section 1.1, extended with procedures, call statements, and value-result parameter passing.
    (ii) I^{S}(*A*, *Base*, *B*) succeeds and produces system *M* iff I^{∞}(*roll-out*(*A*, *Main*), *roll-out*(*Base*, *Main*), *roll-out*(*B*, *Main*)) succeeds and produces M^{∞}.
    (iii) *roll-out*(*M*, *Main*) must equal M^{∞}.

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

This two-level model of integration has the effect of weakening Property (2) of the integration model of Section 1.1, since the new model insists that Property (2) of the old model hold only for the *rolled-out* programs of the model's conceptual level. 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*, *Main*) than in *roll-out*(*Base*, *Main*), while another occurrence of the component has a different behavior in *roll-out*(*B*, *Main*) than in *roll-out*(*Base*, *Main*).

Unfortunately, there are two problems with our new model. The first problem is a complication introduced by the roll-out concept. A rolled-out program may contain multiple occurrences of each procedure that occurs in the original program. For all of the (infinite) programs in the range of *roll-out*, all of the different occurrences of a given procedure are identical; however, in the (infinite) program produced by operation I^{∞} this is not necessarily the case—the different occurrences of a procedure may contain different subsets of the procedure's parameters and statements. If this is the case, it is impossible to provide a single procedure that represents all of the different occurrences that occur in the integrated program. Under these conditions, we say that the result of I^{∞} is *inhomogeneous*.[1] Consequently, we modify Property (2)(ii) of the concrete level of the integration model as follows:

(2)(ii) [modified]

I^{S}(*A*, *Base*, *B*) succeeds and produces system *M* iff I^{∞} (*roll-out*(*A*, *Main*), *roll-out*(*Base*, *Main*), *roll-out*(*B*, *Main*)) succeeds and produces M^{∞} and M^{∞} is homogeneous.

The second problem is that the new model allows procedures to be omitted from an integrated system even though they are present in all three of the systems that are being integrated (*i.e.*, *Base*, *A*, and *B*). For example, consider the systems shown in Figure 4.3. The system labeled "Integrated System" satisfies the new model because *roll-out*(*Integrated System*, *Main*) captures the changed behaviors of both *roll-out*(*A*, *Main*) and *roll-out*(*B*, *Main*) as well as the behaviors that are common to *roll-out*(*A*, *Main*), *roll-out*(*Base*, *Main*), and *roll-out*(*B*, *Main*). Nevertheless, we judge this integrated system to be unsatisfactory because it excludes procedure *Q*, which has not been deleted in either variant.

Base	Variant *A*	Variant *B*	Integrated System

procedure *Main*
  $a := 0$
  $c := 2$
  call $P(a, c)$
 	
  $b := 1$
  $d := 3$
  call $P(b, d)$
 	
end(a, b, c, d)
	procedure *Main*
	  $a := 0$
	  $c := 2$
	  call $P(a, c)$
	  $a := a + 1$
	
	end(a, c)
		procedure *Main*
		
		  $b := 1$
		  $d := 3$
		  call $P(b, d)$
		  $b := b + 2$
		end(b, d)
			procedure *Main*
			  $a := 0$
			
			  call $P(a)$
			  $a := a + 1$
			  $b := 1$
			
			  call $P(b)$
			  $b := b + 2$
			end(a, b)

procedure $P(x, y)$
  $x := x + 1$
  call $Q(y)$
return
	procedure $P(x, y)$
	  $x := x + 1$
	  call $Q(y)$
	return
		procedure $P(x, y)$
		  $x := x + 1$
		  call $Q(y)$
		return
			procedure $P(x)$
			  $x := x + 1$
			
			return

procedure $Q(z)$
  $z := z + 1$
return
	procedure $Q(z)$
	  $z := z + 1$
	return
		procedure $Q(z)$
		  $z := z + 1$
		return	

Figure 4.3. The system labeled Integrated System is an acceptable integration of *Base*, *A*, and *B* according to the revised integration model. However, it is unsatisfactory because procedure *Q*, which is in *Base*, *A*, and *B*, is omitted. (The boxes indicate the modifications made to variants *A* and *B*.)

[1] A formal definition of the term *homogeneous* is given in Section 6.1.

The problem is that the new model only imposes constraints on the rolled-out main procedures of *Base*, *A*, and *B*. To rectify this problem we redefine the model so that integration of rolled-out programs involves *all* procedures, not just the main procedures. Thus, henceforth I^{∞} denotes an operation that combines three *sets* of (possibly infinite) programs, one set for each of three rolled-out systems.

Revised Model of Program Integration, Version 2

(1) The requirements at the conceptual level are modified to cover *sets* of rolled-out procedures: let $M^{\infty} = I^{\infty}(\textit{roll-out}(A), \textit{roll-out}(Base), \textit{roll-out}(B))$. 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 1.1 must hold.

(2) Similarly, at the concrete level, we require that the system produced by algorithm I^{S} must be consistent with the *set* of rolled-out procedures produced by I^{∞}:

  (i) Programs are finite and must be written in the simplified programming language described in Property (1) of the integration model of Section 1.1, extended with procedures, call statements, and value-result parameter passing.

  (ii) $I^{S}(A, Base, B)$ succeeds and produces system M iff $I^{\infty}(\textit{roll-out}(A), \textit{roll-out}(Base), \textit{roll-out}(B))$ succeeds and produces set of programs M^{∞} and M^{∞} is homogeneous.

  (iii) *roll-out*(M) must equal M^{∞}.

In the remaining chapters of this dissertation the terms "the model" and "the multi-procedure integration model" refer to the above model (*i.e.*, these terms are synonymous with "the Revised Model of Program Integration, Version 2").

To satisfy the integration model, we must devise operations I^{∞} and I^{S} and show that the required properties are satisfied. We first define HPR^{∞} to be the natural extension of the HPR algorithm that operates on the infinite program dependence graphs of infinite programs. We then define $Integrate^{\infty}$ to be the operator that applies HPR^{∞} to every (rolled-out) procedure in *roll-out*(*A*), *roll-out*(*Base*), and *roll-out*(*B*) given the additional requirement described below:

$$Integrate^{\infty}(\textit{roll-out}(A), \textit{roll-out}(Base), \textit{roll-out}(B)) \triangleq \bigcup_{P \in (A \cup Base \cup B)} HPR^{\infty}(\textit{roll-out}(A, P), \textit{roll-out}(Base, P), \textit{roll-out}(B, P)).$$

In addition, we require that $Integrate^{\infty}$ place the statements of all scopes derived from the same procedure in the same order. This avoids situations where any operation serving as I^{S} would fail to satisfy Property (2)(iii) because $Integrate^{\infty}$ picks a different statement order for the statements in two scopes with isomorphic dependence sub-graphs. It is important to note that this requirement does not reduce the number of integrations in which $Integrate^{\infty}$ is successful: if $Integrate^{\infty}$ would succeed *without* this requirement, then it will succeed *with* the requirement.

Operator $Integrate^{\infty}$ serves as I^{∞} in the remaining chapters of this dissertation. It is shown in Chapter 8 that $Integrate^{\infty}$ satisfies Property (1).

The remainder of this dissertation deals with the design of $Integrate^{S}$, the multi-procedure integration algorithm that serves as operation I^{S}, and the proof that this algorithm satisfies the model. To begin with, Chapter 5 describes $Integrate^{S}$. It is shown in Chapter 6 that this algorithm satisfies Property (2) of the model. Chapters 7 and 8 deal with the semantics of the merged system. Chapter 7 demonstrates that *roll-out* is a semantics-preserving transformation; thus, Chapter 7 justifies the use of *roll-out* to relate the meanings of *Base*, *A*, *B*, and M to the meanings of *roll-out*(*Base*), *roll-out*(*A*), *roll-out*(*B*), and M^{∞}.

Together with the results from Chapter 6, this implies M captures the correct execution behavior if M^{∞} captures the correct execution behavior. The final proof, in Chapter 8, demonstrates that $Integrate^{\infty}$ satisfies Property (1) of the model; thus, M^{∞} captures the correct execution behavior. Finally, Chapter 9 summerizes the work described in the dissertation and outlines possible future work on program integration.

CHAPTER 5

AN ALGORITHM FOR MULTI-PROCEDURE INTEGRATION

This chapter presents *Integrate*^{S}, an algorithm for multi-procedure program integration that meets the requirements of Version 2 of the Revised Integration Model presented in Section 4.3. Consider again the examples shown in Figures 4.2 and 4.3, which served to motivate our revisions to the integration model of Section 1.1. For the example shown in Figure 4.2, *Integrate*^{S} creates the system labeled "Proposed Integrated System" in Figure 4.2. (Recall that in Section 4.3 we argued that the "Proposed Integrated System" incorporates the changes that were made in variant A in addition to those made in variant B.) For the example shown in Figure 4.3, *Integrate*^{S} produces a system that is identical to the one labeled "Integrated System" in Figure 4.3, except that it also includes procedure Q, whose disappearance motivated Version 2 of the Revised Integration Model.

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. (The Direct-Extension Algorithm of Section 4.2 reports interference, and hence fails to produce the system labeled "Proposed Integrated System" for the example in Figure 4.2 because it does not take into account different calling contexts when determining the changed portions of A and B.) In Chapter 3, we solved a somewhat similar calling-context problem for interprocedural slicing using summary edges (edges that summarize transitive dependences between program elements) and a two-pass backwards slicing algorithm; for multi-procedure integration, we solve the calling-context problem using summary edges and a combination of the individual passes of the backward and forward slicing algorithms (see Section 5.2).

(2) One or more of the arguments to *Integrate*^{S} can contain dead code. During program development, dead code may be introduced temporarily, with the programmer intending to turn it into live code at a later time. Thus, 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. To handle properly systems that contain dead code, the first step of the integration algorithm augments the system dependence graphs 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 5.3). This information is used by the integration algorithm to determine what has changed in variants A and B even when one or more of the systems contains dead code.

The presentation of the multi-procedure integration algorithm is divided into seven subsections: Section 5.1 presents additional background material on the HPR algorithm and interprocedural slicing. Section 5.2 describes the construction of the merged system dependence graph. This construction is actually not quite correct because, although it handles the calling-context problem, it does not handle all examples correctly when systems contain dead code. Section 5.3 describes a preprocessing step (the addition of meeting-point vertices and edges) necessary to handle dead code correctly. Section 5.4 discusses the test for Type I

interference. Section 5.5 presents the homogeneity test, which tests whether M^{∞} (the result of applying $Integrate^{\infty}$ to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*)) is homogeneous. Section 5.6 outlines how the source text corresponding to the merged system dependence graph is obtained; and, finally, Section 5.7 contains a recap of the multi-procedure integration algorithm $Integrate^{S}$.

A common theme in these subsections is that the operations being defined are *generalizations* of those used in the HPR algorithm; that is, when restricted to single-procedure programs corresponding operations produce the same result. This relationship is formalized in [Binkley91], where it is shown that, when applied to three single-procedure programs, the HPR algorithm and $Integrate^{S}$ produce the same integrated program.

In most cases, our initial attempt to define a generalized operation for use in $Integrate^{S}$ is the *natural extension* of the corresponding operation used in the HPR algorithm. By the "natural extension" we mean the operator from the HPR algorithm, extended to consider the new kinds of vertices and edges found in a system dependence graph. For example, the natural extension of $AP^{hpr}(A, Base)$, which was defined as

$$AP^{hpr}(A, Base) \triangleq (A, \{ v \in V(A) \mid b^{hpr}(A, v) \neq b^{hpr}(Base, v) \})$$

is

$$AP^{S}(A, Base) \triangleq (A, \{ v \in V(A) \mid b(A, v) \neq b(Base, v) \}).$$

Although it is often used as our first attempt, in most cases, the natural extension of an operation from the HPR algorithm fails to correctly account for calling context (even though each individual interprocedural slicing operation used in the extended definition *does* correctly account for calling context).

5.1. Additional Background on the HPR Algorithm and Interprocedural Slicing

This section presents additional background material on the HPR algorithm and on interprocedural slicing. In particular, Section 5.1 begins by reformulating some of the concepts from the HPR algorithm, as introduced in Chapter 2, in a way that resembles more closely some related concepts used in $Integrate^{S}$. (Throughout the rest of the dissertation, we use superscripts to distinguish between related quantities in the HPR algorithm and $Integrate^{S}$: those used in the HPR algorithm have the superscript "*hpr*;" those used in $Integrate^{S}$ have the superscript "*S*," *e.g.*, Δ^{hpr} versus Δ^{S}, *etc.*). Section 5.1 ends with an equational definition for the summary edges introduced in Chapter 3.

A Reformulation of Affected Points

We now give an alternative definition of affected points in terms of a special subset of the affected points, called the *directly affected points*. First, define the sets of incoming edges for a vertex v in program dependence graph G as follows:

$$IncomingControl^{hpr}(G, v) \triangleq \{ w \longrightarrow_{c} v \mid w \longrightarrow_{c} v \in E(G) \}$$
$$IncomingFlow^{hpr}(G, v) \triangleq \{ w \longrightarrow_{f} v \mid w \longrightarrow_{f} v \in E(G) \}$$
$$IncomingDefOrder^{hpr}(G, v) \triangleq \{ x \longrightarrow_{do(v)} y \mid x \longrightarrow_{do(v)} y \in E(G) \}.$$

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

Directly affected points can be used to determine the affected points, which in turn are used to determine Δ^{hpr}. Vertex v is a *directly affected point* of A with respect to *Base* iff v is in A but not *Base* or v has different incoming edges in A and *Base*. Thus, for the HPR algorithm, the set of directly affected points of A with respect to *Base*, denoted by $DAP^{hpr}(A, Base)$, is defined as follows:

$$DAP^{hpr}(A, Base) \triangleq (A, \{\, v \in V(A) \mid \quad v \notin V(Base)$$
$$\vee IncomingControl^{hpr}(A, v) \neq IncomingControl^{hpr}(Base, v)$$
$$\vee IncomingFlow^{hpr}(A, v) \neq IncomingFlow^{hpr}(Base, v)$$
$$\vee IncomingDefOrder^{hpr}(A, v) \neq IncomingDefOrder^{hpr}(Base, v)\,\}).$$

The vertices of $DAP^{hpr}(A, Base)$ 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.*, $AP^{hpr}(A, Base)$, the affected points of A with respect to $Base$) by finding those vertices of A in the forward slice of A with respect to $DAP^{hpr}(A, Base)$. Thus, we have

$$AP^{hpr}(A, Base) = f^{hpr}(DAP^{hpr}(A, Base)) \text{ and}$$
$$\Delta^{hpr}(A, Base) = b^{hpr}(AP^{hpr}(A, Base)).$$

(Operator f^{hpr} denotes the hpr forward slice, introduced in Section 2.2.1.)

A Reformulation of Type I Interference

We now reconsider the test for Type I interference. According to the definition given in Section 2.3, Type I interference exists when the slice with respect to the affected points of a variant is corrupted in G_M by the other variant. Reps and Bricker have given an alternative characterization of the test for Type I interference used in the HPR algorithm [Reps89a]. The alternative test reports interference at the point of corruption; interference is reported when a vertex in $\Delta^{hpr}(A, Base)$ is a directly affected point of G_M with respect to A (or if the analogous condition holds for B). That is, Type I interference occurs iff

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

In [Reps89a], this definition of Type I interference is shown to be equivalent to the original definition.

An Equational Definition of Summary Edges

This section presents an equational definition for summary edges, in which the edges of a graph are viewed as a binary relation over the vertices of the graph.[1] This provides a convenient notation for expressing connections in a procedure dependence graph. For example, the transitive closure of the edge relation provides a convenient expression for paths in a graph. The purpose of this definition is to introduce some basic techniques and notation that are used later to introduce new concepts. Before defining summary edges using this notation, we make the following definitions:

DEFINITION. $Call_Q$ denotes a call-site vertex that represent 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 procedure dependence graph G_P (*i.e.*, G_P with no def-order edges).

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

For each procedure P, the *augmented control-flow dependence subgraph*, denoted by ACF_P, CF_P augmented with summary edges. The addition of a summary edge at a call-site in procedure Q may complete a path from a formal-in vertex to a formal-out vertex in G_Q, which in turn may enable the addition of further summary edges in procedures that call Q. This interdependence in the creation of summary edges is captured using a least fixed point in the definition. Thus, the augmented control-flow dependence subgraphs are the least fixed point of the following set of equations (one for each procedure P):

[1] Def-order edges would be represented by a ternary relation, however, they are not used in the following definitions.

DEFINITION. (Summary Edges).

$$ACF_P \triangleq CF_P \cup (V(G_P), \{ (Call_Q.a_{in}, Call_Q.b_{out}) \mid Call_Q \in V(G_P) \wedge (Enter_Q.a_{in}, Enter_Q.b_{out}) \in E(ACF_Q^+) \}).$$

Note that the last term in this definition refers to ACF_Q^+, the transitive closure of ACF_Q, because the edges of ACF_P depend on the existence of *paths* in ACF_Q.

The procedure dependence graph for P augmented with summary edges is obtained by taking the union of ACF_P and G_P.

5.2. Constructing the Merged System Dependence Graph

We now describe the construction of the merged system dependence graph. The form of the construction is similar to the construction of the merged program dependence graph in the HPR algorithm; it involves combining three graphs: $\Delta^S(A, Base)$, which captures the changes made to *Base* to create variant A, $\Delta^S(B, Base)$, which captures the changes made to *Base* to create variant B, and $Pre^S(A, Base, B)$, which captures what is common to *Base*, A, and B.

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

(Throughout the remainder of the dissertation, we use G_M to represent the merged graph created by $Integrate^S(A, Base, B)$.)

The computations of $\Delta^S(A, Base)$ and $\Delta^S(B, Base)$ require the computations of the sets $DAP^S(A, Base)$ and $DAP^S(B, Base)$. Below we discuss how to construct DAP^S, Δ^S, and Pre^S. A common thread in all three constructions is the need to keep track of calling context. (The construction described below is actually not quite correct because, although it handles the calling-context problem, the construction of Δ^S may fail to include sufficient vertices when systems contain dead code. This shortcoming will be rectified in Section 5.3.)

Constructing DAP^S

Given two system dependence graphs A and *Base*, the set of directly affected points of A with respect to *Base*, denoted by $DAP^S(A, Base)$, is defined similarly to $DAP^{hpr}(A, Base)$; $DAP^S(A, Base)$ consists of the vertices of A that are not in *Base* and those whose incoming edges are different in A and *Base*. However, for the purposes of identifying directly affected points in system dependence graphs, parameter-in, call, parameter-out, and summary edges are ignored. Thus, a vertex v that occurs in both A and *Base* is a directly affected iff v has different in incoming control, flow, or def-order edges.

$$DAP^S(A, Base) \triangleq (A, \{ v \in V(A) \mid v \notin V(Base) \vee IncomingControl^S(A, v) \neq IncomingControl^S(Base, v) \vee IncomingFlow^S(A, v) \neq IncomingFlow^S(Base, v) \vee IncomingDefOrder^S(A, v) \neq IncomingDefOrder^S(Base, v) \}).$$

It is crucial to the correctness of $Integrate^S$ to *exclude* differences in parameter-in and call edges from the definition of $DAP^S(A, Base)$. As discussed shortly, the construction of $\Delta^S(A, Base)$ from $DAP^S(A, Base)$ involves b (*i.e.*, "full backward") slices with respect to each directly affected point v of A;[2] 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 $\Delta^S(A, Base)$.

---

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

In particular, this problem occurs when A changes the incoming parameter-in or call edges of a procedure P (by adding or deleting a call on P or changing 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 an existing call-site on P. For example, Figure 5.1 illustrates the case where A deletes a call-site on procedure P and B changes the value of an actual parameter at an existing call-site on P. In A, the deletion of call statement "**call** $P(a)$" changes the incoming parameter-in and call edges of the formal-in vertex labeled "$x := x_{in}$" and entry vertex $Enter_P$, respectively. This should not cause the actual-in and call-site vertices of remaining call-site on P to be incorporated into $\Delta^S(A, Base)$. If they were then "$b := 0$" would also be included in $\Delta^S(A, Base)$ and thus there would be a conflict with the change made in variant B. (For this example, the goal is to produce the system labeled "Proposed M".)

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 $Integrate^S$ not does it change the merged program produced by $Integrate^S$. It does, however, simplify the argument that $Integrate^S$ satisfies Version 2 of the Revised Model of Program Integration. For example, considering differences in incoming parameter-out and summary edges complicates the relation that otherwise holds between $DAP^S(A, Base)$ and $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$: only when parameter-out and summary edges are ignored is the set of occurrences from $roll\text{-}out(A)$ of vertices in $DAP^S(A, Base)$ a subset of $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. This subset relation simplifies the argument that $Integrate^S$ satisfies Version 2 of the Revised Model of Program Integration (*e.g.*, see the DAP Equivalence Lemma in Chapter 6).

As discussed in Chapter 2, determining the directly affected points (as well as the other steps of $Integrate^{hpr}$) require program components to be tagged so that corresponding components can be identified in all three versions (*i.e.*, *Base*, *A*, and *B*). Chapter 2 does not discuss tags for call statements. Because a call statement is represented by multiple vertices in a system dependence graph it has multiple tags: one for each vertex. For example, *Call* $P(a, b)$, where a but not b is a member of $GMOD(P)$, has four tags: a tag for the call-site vertex, two tags for the two actual-in vertices, and a tag for a's actual-out vertex. Furthermore, in addition to the conditions imposed on tags in Chapter 2, 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 accomplished this by encoding the parameter-list position of actual parameters in the tags of their actual-in (and actual-out) vertices; thus, interchanging two parameters creates directly affected points only at the

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** $P(a)$		$x := x + 1$
$b := 0$	$b := 0$	[$b := 1$]	$b := 1$	**return**
call $P(b)$	**call** $P(b)$	**call** $P(b)$	**call** $P(b)$	
end()	**end**()	**end**()	**end**()	

Figure 5.1. Motivation for the definition of $DAP^S(A, Base)$. (The boxes indicate the modifications made to variants A and B.)

call-site where the interchange took place. (A similar encoding is performed for formal-in and formal-out vertices.)

Constructing Δ^S

As in the HPR algorithm, the notion of the *affected points* of variant A with respect to *Base* is used to identify the differences between A and *Base*. The affected points are vertices that potentially exhibit changed behavior in some calling context.

In the HPR algorithm, there are two equivalent definitions of $AP^{hpr}(A, Base)$:

(1) $AP^{hpr}(A, Base) \triangleq (A, \{ v \in V(A) \mid b^{hpr}(A, v) \neq b^{hpr}(Base, v) \})$, and
(2) $AP^{hpr}(A, Base) \triangleq f^{hpr}(DAP^{hpr}(A, Base))$.

However, the natural extensions of these definitions (*i.e.*, the definitions that use in*ter*procedural slices in place of in*tra*procedural slices) are not equivalent. This is illustrated by the example shown in Figure 5.1. Under the natural extension of Definition (1), all vertices of procedure P in variant A would be affected points. Under the natural extension of Definition (2), the set of affected points would be empty because the set of directly affected points $DAP^S(A, Base)$ is empty.

Because the natural extension of Definition (2) reuses the context information contained in $DAP^S(A, Base)$, it is the correct definition for multi-procedure integration; it is the one that is consistent with the notion that the affected points are the vertices that potentially exhibit changed behavior in some calling context. Consequently, we define the affected points as follows:

$$AP^S(A, Base) \triangleq f(DAP^S(A, Base)).$$

Note that, for the example in Figure 5.1, $AP^S(A, Base)$ being empty is consistent with the fact that the one calling context in which P is invoked in variant A exhibits the same behavior as the corresponding calling context for P in *Base*.

For multi-procedure integration it is also necessary to identify a special subset of the affected points, which we call the *strongly affected points*. Whereas an affected point potentially exhibits changed behavior in *some* calling context, a strongly affected point potentially exhibits changed behavior in *all* calling contexts. Thus, 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 *fl* slice with respect to a directly affected point:

$$SAP^S(A, Base) \triangleq fl(DAP^S(A, Base)).$$

Strongly affected points and affected points that are not strongly affected (hereafter referred to as weakly affected points) contribute differently to the definition of Δ^S. As expressed below, Δ^S is defined in two parts: one part captures changes associated with strongly affected points; the other captures changes associated with weakly affected points.

(1) Because the execution behavior at each strongly affected point v is potentially modified in *every* calling context in which v 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.
(2) Because the execution behavior at each weakly affected point v is potentially modified only in *some* calling contexts in which v is executed, it is necessary only to incorporate *some* of v's possible calling contexts in Δ^S. Since the vertices of the call-sites that make up the calling contexts in which v has potentially modified execution behavior are *themselves* affected points, it is only necessary to take a *b2* slice with respect to v.

This second point deserves some clarification. Suppose v is a weakly affected point. A *b2* slice with respect to v will only include vertices in P and procedures called by P; it does not include any vertices in procedures that call P. Initially this may seem incorrect because for a weakly affected point *some* calling context must have changed. However, a call-site associated with any changed calling context would itself

be an affected point; thus, any changed calling context for P in A with respect to *Base* will also be included in $\Delta^S(A, Base)$ (as desired).

Putting the two parts of Δ^S together produces the following definition of $\Delta^S(A, Base)$:

$$\Delta^S(A, Base) \triangleq b(SAP^S(A, Base)) \cup b2(AP^S(A, Base)).$$

(The second term of the union could be replaced by the $b2$ slice taken with respect to only the weakly affected points without affecting the resulting set $\Delta^S(A, Base)$.)

Expanding the right-hand side by the definitions of b, f, SAP^S, and AP^S, Δ^S may also be expressed as

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

Operationally, each of the two main terms in the definition of Δ^S represents three linear-time passes over the system dependence graph of A. During each pass, only certain kinds of edges in the graph are traversed.[3]

Example. In Figure 4.2, $\Delta^S(A, Base)$ 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 4.2) without causing interference.

Example. The example in Figure 5.2 shows the two parts of $\Delta^S(A, Base)$ computed from the systems A and *Base* also shown in the figure. In this example $DAP^S(A, Base)$ contains the new assignment 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 are also the strongly affected points. The weakly affected points are the formal-in vertex for z in procedure Q, and the assignment statement "$t2 := z$". For the strongly affected points, the first

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** $P()$
end(a, b)	**end**(a, b)	**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** $Q(z)$
$t2 := z$	$t2 := z$		$t2 := z$	$t2 := z$
return	**return**		**return**	**return**

Figure 5.2. 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 programs (really their system dependence graphs) yields a program (system dependence graph) 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.)

[3] Because for both terms the initial pass is $f1(DAP^S(A, Base))$, a total of five passes is required; a further reduction to four passes is possible if the union of $b1(f1((DAP^S(A, Base)))$ and $f2(f1(DAP^S(A, Base)))$ is performed before a final $b2$ pass is made.

part of $\Delta^S(A, Base)$, $b(SAP^S(A, Base))$, includes *all* calling contexts for procedure P because of the strongly affected points are in P. For the weakly affected points, the second part of $\Delta^S(A, Base)$, $b2(AP^S(A, Base))$, includes the necessary parts of procedure Q without including the first call-site on Q because Q contains only weakly affected points. Together these two parts capture all calling contexts for P and one calling context for Q; these are the only affected calling contexts in A.

Constructing Pre^S

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

(1) Procedure Q 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. This is consistent with Version 2 of the Revised Integration Model presented in Section 4.3, which was motivated by the example in Figure 4.3. Recall that for the example in Figure 4.3 we want procedure Q to be included in the integration of *Base*, A, and B even though there is no call-site on Q that is common to *Base*, A, and B. 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. Thus, we define Pre^S as follows:

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

By comparing $b2$ slices with respect to the vertices in a procedure Q, we achieve our goal of ignoring the contexts in which Q is called, while still correctly accounting for calling contexts in the calls made (transitively) from Q (*e.g.*, *all* of the vertices in Q from Figure 4.3 have the same $b2$ slice in *Base*, A, and B).

Constructing the Merged Graph

The merged graph is constructed as follows:

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

In [Binkley91] 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.

5.3. Adding Meeting-Point Vertices to System Dependence Graphs

The construction of Δ^S described in Section 5.2 is not quite correct because, although it handles the calling-context problem, it does not handle systems with dead code. This section describes how to overcome this limitation. To handle dead code correctly, it is necessary to add some additional vertices and edges to the system dependence graphs 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 (they also improve the efficiency of the computation of Δ^S in the presence of dead code). After the addition of these vertices and edges the computation of Δ^S proceeds exactly as described in Section 5.2 (except that the slicing operators $b1$, $b2$, $f1$, and $f2$ are extended to also traverse meeting-point edges). This section begins by describing the specific problem meeting-point vertices and edges solve and why this problem exists only in the presence of dead code. It then presents a formal definition of meeting-point vertices and edges before giving an algorithm for efficiently computing

them.

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.

An additional benefit of adding meeting-point vertices and edges is that the actual-in vertices that must be included in Δ^S can be determined without examining called procedures' procedure dependence graphs. In this regard meeting-point vertices and edges serve a purpose in the computation of Δ^S similar to that served by summary edges in the computation of an interprocedural slice.

Example. For the example shown in Figure 5.3, if Δ^S 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*; thus, these actual-in vertices are connected to a meeting-point vertex *mp*. Because *mp*, like the actual-in vertex for parameter *a*, is an 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.

The problem solved by meeting-point vertices and edges exists only in the presence of dead code because 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

Base	Variant *A*	Variant *B*	*Main*'s first call on *P* from *A*
procedure *Main*	**procedure** *Main*	**procedure** *Main*	
a :=2	[*a* :=1]	*a* :=2	
b :=1	*b* :=1	*b* :=1	
call *P*(*a*, 1)	**call** *P*(*a*, 1)	**call** *P*(*a*, 1)	
call *P*(*b*, 1)	**call** *P*(*b*, 1)	**call** *P*(*b*,[2])	
end(*a*)	**end**(*a*)	**end**(*a*)	
procedure *P* (*x*, *y*)	**procedure** *P* (*x*, *y*)	**procedure** *P* (*x*, *y*)	
t1 :=*x*+*y*	*t1* :=*x*+*y*	*t1* :=*x*+*y*	
call *Incr* (*x*)	**call** *Incr* (*x*)	[ ]	
	[*t3* :=*x*]		
x :=2	*x* :=2	*x* :=2	
return	**return**	**return**	
procedure *Incr* (*z*)	**procedure** *Incr* (*z*)	**procedure** *Incr* (*z*)	
t2 :=1/*z*	*t2* :=1/*z*	*t2* :=1/*z*	
z :=*z*+1	*z* :=*z*+1	*z* :=*z*+1	
return	**return**	**return**	

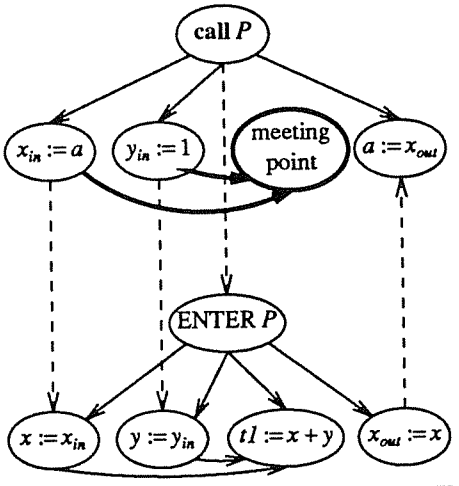


Figure 5.3. This example shows three systems that contain dead code (assignments to the temporary local variables *t1*, *t2* and *t3* and the call to *Incr* in *P*). The right-hand column shows a fragment of the system dependence graph for *A* augmented with meeting-point vertices (shown in bold). (The boxes indicate the modifications made to variants *A* and *B*.)

values are both affect the computation represented at vertex v (*e.g.*, in Figure 5.3, 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 using the kind of notation we used in Section 5.1. Meeting-point vertices can be obtained from the least fixed-point of the following set of equations, which define, for each procedure P, $MpACF_P$: the control-flow dependence subgraph of G_P augmented with summary edges and meeting-point vertices and edges. In the following 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 "$(Call_Q.a_{in}, v), (Call_Q.b_{in}, v) \mid \cdots$" 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

$$V = \{\, <Call_Q.a_{in}, Call_Q.b_{in}> \mid Call_Q \in V(G_P) \wedge Call_Q.a_{in} \neq Call_Q.b_{in} \,\}$$

$$\begin{aligned} E = \{\, & (Call_Q.a_{in}, v), (Call_Q.b_{in}, v) \mid Call_Q \in V(G_P) \\ & \wedge\ v \in V \wedge v = <Call_Q.a_{in}, Call_Q.b_{in}> \\ & \wedge\ \exists x \in V(MpACF_Q)\ s.t.\ \{\, (Enter_Q.a_{in}, x), (Enter_Q.b_{in}, x) \,\} \subseteq E(MpACF_Q^+) \\ & \wedge\ \exists u \in V(ACF_P)\ s.t.\ \{\, (Call_Q.a_{in}, u), (Call_Q.b_{in}, u) \,\} \subseteq E(ACF_P) \,\}. \end{aligned}$$

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 procedure dependence graph 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 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 5.3 illustrates the addition of meeting-point vertices and edges. In procedure dependence graph 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 system dependence graph for Variant A at both call-sites on P. As noted in the above example, 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 actual-in vertices associated with the first call on P (in particular the one labeled "$y_{in} := 1$").

An Algorithm for Efficiently Computing Meeting-Point Vertices and Edges

Meeting-point vertices can be computed efficiently using the algorithm shown in Figure 5.4. This algorithm makes use of the *TDP* and *TDS* graphs computed when constructing the subordinate characteristic graphs of the linkage grammar (the attribute grammar used in Section 3.2.2 to compute summary edges). Recall that this grammar has one nonterminal and one production for each procedure. The nonterminal for procedure P has one attribute for each formal-in and formal-out vertex in P's procedure dependence graph. The production for procedure P has nonterminal P as its left-hand-side and on its right-hand-side a nonterminal occurrence for each call-site in P. In essence, the graph $TDP(P)$ is a projection of the formal-in, entry, formal-out, call-site, actual-in, and actual-out vertices of G_P whose edges summarize paths in G_P;

the graph $TDS(P)$ is a projection of the formal-in, entry, and formal-out vertices of G_P whose edges also summarize paths in G_P. Both of these graphs are used in the algorithm for computing meeting-point vertices shown in Figure 5.4.

The algorithm shown in Figure 5.4 adds meeting-point vertices and edges to a system dependence graph in three steps. As described below, in the first two steps new attributes are added to $TDS(P)$ (provided there is no subsuming actual-out vertex at call-sites on P). New attributes from $TDS(P)$ are then used in the third step to add meeting-point vertices and edges at call-sites on P. As an optimization in the algorithm, only those meeting-point vertices that are connected to actual-in vertices are added (*i.e.*, not all pairings of the actual-in vertices are represented).

Step 1: Determining initial meeting-point attributes (lines [2] - [10] in Figure 5.4)

The first step determines meeting-point attributes that represent meeting points that are not themselves meeting-point vertices. This is done using *local slices*, extended HPR slices that traverses summary edges and meeting-point edges in addition to control and flow edges. As with other kinds of slices, there are both backward local slices and forward local slices, denoted by *bl* and *fl*, respectively.

Initial meeting-point attributes are determined using local slices to slice forward and then backward from each formal-in vertex and noting any other formal-in vertices encountered. For example, if $Enter_P.b_{in} \in bl\,fl(P, Enter_P.a_{in})$ then there is a vertex v in $fl(P, Enter_P.a_{in})$ such that $Enter_P.b_{in} \in bl(P, v)$, and hence vertex v is a meeting point.

Step 2: Adding additional meeting-point attributes (lines [11] - [22] in Figure 5.4)

The second step of the algorithm checks if the addition of a meeting-point attribute from either Step 1 or Step 2 enables the addition of further meeting-point attributes. For each call-site on P in procedure Q, the edges of $TDP(Q)$ are used to establish the existence of connections from two formal-in vertices of Q to two actual-in vertices at a call-site on P. If these two actual-in vertices will be connected to a meeting-point vertex (because of an added meeting-point attribute and edges in $TDS(P)$) then a new meeting-point attribute is added to $TDS(Q)$.

Step 3: Computing meeting-point vertices from meeting-point attributes (lines [23] - [26] in Figure 5.4)

For each meeting-point attribute in $TDS(P)$, the third step of the algorithm adds a meeting-point vertex and associated edges to each call-site on P.

After the addition of meeting-point vertices and edges the construction of $\Delta^S(A, Base)$ proceeds exactly as described in Section 5.2 (except that the slicing operators *bl*, *b2*, *fl*, and *f2* are each extended to traverse meeting-point edges).

Example. Figure 5.5 shows the merged system created by $Integrate^S$ when applied to the three systems shown in Figure 5.3 when meeting-point vertices are used to compute $\Delta^S(A, Base)$ and $\Delta^S(B, Base)$.

5.4. Testing for Type I Interference

As with the computation of AP^S, there are two equivalent way of expressing the test for Type I interference in the HPR algorithm, yet only one of them provides the correct generalization. For the HPR algorithm there is no Type I interference if

```
     function AddMeetingPointVerticesAndEdges(G) returns an updated system dependence graph
     declare
       G : a system dependence graph
       worklist, F, F1, F2 : sets of attributes
       a, x1, x2, y1, y2 : attributes
     begin
[ 1]   worklist := ∅

     /* determine initial meeting-point attributes (those not dependent on the existence of other meeting-point attributes
        i.e., those for which the "common computation" is in the same procedure and the formal-in vertices) */
[ 2]   for each procedure P do
[ 3]     Let the formal-in vertices of GP be numbered 1, ···, n
[ 4]     Let P.i denote the attribute corresponding to formal-in vertex EnterP.iin
[ 5]     for i := 1 to n do
[ 6]       F := { P.j | j>i ∧ EnterP.jin ∈ bl fl(P, EnterP.iin) }
[ 7]       for each f ∈ F do
[ 8]         if there is no attribute a such that P.i ⟶ a and f ⟶ a in TDS(P) then
[ 9]           Add (meeting-point) attribute a, together with edges P.i ⟶ a and f ⟶ a, to TDS(P)
[10]           Insert a into worklist
             fi
           od
         od
       od

     /* add dependent meeting-point attributes */
[11]   while worklist ≠ ∅ do
[12]     Select and remove a meeting-point attribute a from worklist
[13]     Let TDS(P) be the TDS graph that contains a
[14]     for each callsite CallP do
[15]       Let Q be the procedure that contains CallP
[16]       (x1, x2) := the two attributes such that x1 ⟶ a and x2 ⟶ a in TDS(P)
[17]       F1 := { f | f ⟶ x̂1 in TDP(Q) ∧ x̂1 is the occurrence of x1 at CallP ∧ f corresponds to a formal-in vertex }
[18]       F2 := { f | f ⟶ x̂2 in TDP(Q) ∧ x̂2 is the occurrence of x2 at CallP ∧ f corresponds to a formal-in vertex }
[19]       for each pair (y1, y2) from F1 × F2 such that y1 ≠ y2 do
[20]         if there is no attribute a such that y1 ⟶ a and y2 ⟶ a in TDS(Q) then
[21]           Add (meeting-point) attribute a, together with edges y1 ⟶ a and y2 ⟶ a, to TDS(Q)
[22]           Insert a into worklist
             fi
           od
         od
       od

     /* insert meeting-point vertices and edges at call-sites in G from meeting-point attributes in the TDS graphs */
[23]   for each added meeting-point attribute a do
[24]     Let TDS(P) be the TDS graph that contains a and suppose that formal-in vertices EnterP.xin and EnterP.yin
           correspond to the two attributes that are connected to a
[25]     for each call-site on P do
[26]       Add (meeting-point) vertex m, together with edges CallP.xin ⟶mp m and CallP.yin ⟶mp m, to G
         od
       od
[27]   return(G)
     end
```

Figure 5.4. This algorithm contains the three steps by which meeting-point vertices and edges are added to a system dependence graph G.

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

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 5.5. The result of applying $Integrate^S$ to the three systems shown in Figure 5.3.

(1) $b^{hpr}(AP^{hpr}(A, Base)) = b^{hpr}(M, (AP^{hpr}(A, Base)))$ and
$b^{hpr}(AP^{hpr}(B, Base)) = b^{hpr}(M, (AP^{hpr}(A, Base)))$,
or equivalently if
(2) $\Delta^{hpr}(A, Base) \cap DAP^{hpr}(M, A) = \emptyset$ and
$\Delta^{hpr}(B, Base) \cap DAP^{hpr}(M, B) = \emptyset$.

Test (1) (which was discussed in Chapter 2) ensures that the (backward) slices of A and B with respect to their affected points are preserved in the merged graph. Intuitively, this test ensures that the computations represented in $\Delta^{hpr}(A, Base)$ (*i.e.*, $b^{hpr}(AP^{hpr}(A, Base))$) is not corrupted in M by the changes in B (and symmetrically for B). Test (2), which was discussed in Section 5.1, identifies corrupted slices at the point of corruption.

For multi-procedure integration, the natural extension of Test (1) loses necessary calling-context information because it uses full backward slices. Such slices consider parameter-in and call edges, which, similar to the construction of DAP^S, should be ignored in the test for Type I interference. On the other hand, the natural extension of Test (2), by virtue of using DAP^S, ignores these edges. Thus, for multi-procedure integration, the test that the correct slices are uncorrupted in G_M (*i.e.*, that there is no Type I Interference) is the natural extension of the second test:

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

As shown in Chapter 6, under the assumptions that M^∞ (the result of applying $Integrate^\infty$ to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*)) is homogeneous and that G_M is feasible, $Integrate^S$ determines there is Type I interference in G_M iff $Integrate^\infty$ determines there is Type I interference in G_{M^∞}.

5.5. Testing for Homogeneity

Recall that Version 2 of the Revised Model of Multi-Procedure Program Integration requires that *roll-out*(M) equal M^∞ whenever M^∞ is homogeneous. The final step of $Integrate^S$ is to test whether M^∞ is homogeneous (it is shown in Chapter 6 that whenever M^∞ is homogeneous, it is also equal to *roll-out*(M)). However, because M^∞ is potentially infinite, it cannot be tested directly; therefore, the homogeneity test (function *IsHomogeneous* in Figure 5.10) is applied to the finite system dependence graphs for *Base*, *A*, *B*, and *M*. If they pass the homogeneity test then, as shown in Chapter 6, M^∞ is guaranteed to be homogeneous and equal to *roll-out*(M); if they fail to pass the homogeneity test then M^∞ is inhomogeneous, in which case it is impossible for M^∞ to equal *roll-out*(M).

Example. The need for the homogeneity test is illustrated in Figure 5.6. In this example, there is no Type I or Type II interference in the integration of *Base*, *A*, and *B*. For the integration of single-procedure programs this absence of interference is sufficient to guarantee the merged program has the correct semantic properties. However, as illustrated by the integrated system M in Figure 5.6, this is not the case for multi-procedure integration. (System M in Figure 5.6 fails the termination property of a successful integration—it aborts with a division-by-zero error during the second call on P for initial states on which

Base	Variant *A*	Variant *B*	*M*

procedure *Main*
  $a := 1$
  $b := 2$
  call $P(a)$
  call $P(b)$
end()
	procedure *Main*
	  $\boxed{a := 10}$
	  $b := 2$
	  call $P(a)$
	  call $P(b)$
	end()
		procedure *Main*
		  $a := 1$
		  $\boxed{b := 0}$
		  call $P(a)$
		  call $P(b)$
		end()
			procedure *Main*
			  $a := 10$
			  $b := 0$
			  call $P(a)$
			  call $P(b)$
			end()

procedure $P(x)$
  $t := 1/x$
  $x := x+1$
return
	procedure $P(x)$
	  $t := 1/x$
	  $x := x+1$
	return
		procedure $P(x)$
		  $\square$
		  $x := x+1$
		return
			procedure $P(x)$
			  $t := 1/x$
			  $x := x+1$
			return

roll-out(*Base*)	*roll-out*(*A*)	*roll-out*(*B*)	M^{∞}	*roll-out*(*M*)

procedure *Main*
  $a := 1$
  $b := 2$
  scope $P(x := a;$
  $a := x)$
  $t := 1/x$
  $x := x+1$
  epocs
  scope $P(x := b;$
  $b := x)$
  $t := 1/x$
  $x := x+1$
  epocs
end()
	procedure *Main*
	  $\boxed{a := 10}$
	  $b := 2$
	  scope $P(x := a;$
	  $a := x)$
	  $t := 1/x$
	  $x := x+1$
	  epocs
	  scope $P(x := b;$
	  $b := x)$
	  $t := 1/x$
	  $x := x+1$
	  epocs
	end()
		procedure *Main*
		  $a := 1$
		  $\boxed{b := 0}$
		  scope $P(x := a;$
		  $a := x)$
		  $\square$
		  $x := x+1$
		  epocs
		  scope $P(x := b;$
		  $b := x)$
		  $\square$
		  $x := x+1$
		  epocs
		end()
			procedure *Main*
			  $a := 10$
			  $b := 0$
			  scope $P(x := a;$
			  $a := x)$
			  $t := 1/x$
			  $x := x+1$
			  epocs
			  scope $P(x := b;$
			  $b := x)$
			  $x := x+1$
			  epocs
			end()
				procedure *Main*
				  $a := 10$
				  $b := 0$
				  scope $P(x := a;$
				  $a := x)$
				  $t := 1/x$
				  $x := x+1$
				  epocs
				  scope $P(x := b;$
				  $b := x)$
				  $\boxed{t := 1/x}$
				  $x := x+1$
				  epocs
				end()

procedure $P(x)$
  $t := 1/x$
  $x := x+1$
return
	procedure $P(x)$
	  $t := 1/x$
	  $x := x+1$
	return
		procedure $P(x)$
		  $\square$
		  $x := x+1$
		return
			procedure $P(x)$
			  $x := x+1$
			return
				procedure $P(x)$
				  $\boxed{t := 1/x}$
				  $x := x+1$
				return

Figure 5.6. This example illustrates the need for the homogeneity test (which the example fails to pass, since it fails to pass the absent-vertex test— see Section 5.5.2). The boxes in *A* and *B* indicate the modifications made to Variants *A* and *B*; the boxes in *roll-out*(*A*) and *roll-out*(*B*) indicate the modifications that would have been made to create *roll-out*(*A*) and *roll-out*(*B*) from *roll-out*(*Base*); and the boxes in *roll-out*(*M*) highlight the extra occurrences of the statement "$t := 1/x$", *i.e.*, those not in M^{∞}. (Recall that because a scope statement defines a new name space, it is parameterized by two lists of assignment statements that transfer values to and from this name space. For example, in the scope statement

scope $P(x := b;$
$b := x)$
...
epocs

transfer-in statement "$x := b$" transfers the value of b from the name space of the enclosing scope to x in the name space of **scope** P; similarly, transfer-out statement "$b := x$" transfers the return value of x from the name space of **scope** P to b in the name space of enclosing scope.)

Base, *A*, and *B* all terminate normally.) In this example, it is the homogeneity test that determines *M* is unsatisfactory. This test determines failure because M^{∞}, the result of integrating *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*), is inhomogeneous: it contains two *P*-scopes (*i.e.*, scopes created from procedure *P* by the expansion of a call on *P*), one with an occurrence of the statement "$t := 1/x$" (the first) and one without (the second).

The following description of the homogeneity test has four parts: it describes *extra* occurrences, a condition that leads to inhomogeneity; the *absent-vertex test* and the *absent-call-site test* of the homogeneity test,

which test for extra occurrences; and, finally, the algorithm given in Figure 5.10 that implements the homogeneity test.

5.5.1. Extra Occurrences

An *extra* occurrence is a vertex that occurs in *roll-out*(M) but not M^∞ (*e.g.*, the occurrence of "$t := 1/x$" in the second P-scope of *Main* from *roll-out*(M) shown in Figure 5.6). As shown in Chapter 6, if no extra occurrences exist in *roll-out*(M) then M^∞ is homogeneous and equal to *roll-out*(M). (In other words, the occurrences in M^∞ always exist in *roll-out*(M), the only question is whether M^∞ contains all the occurrences of *roll-out*(M) or only some of these occurrences.) The purpose of the homogeneity test is to determine if any extra occurrences exist in *roll-out*(M).

For an occurrence in *roll-out*(M) to be extra, it must *not* exist in both *roll-out*(A) and *roll-out*(B).[4] Without loss of generality, the remainder of the discussion of the homogeneity test is phrased in terms of an occurrence v^∞ from *roll-out*(M) that does not occur in *roll-out*(B). Since v^∞ exists in *roll-out*(M), M must contain a vertex v and a sequence of call-sites *Seq* such that the expansion of *Seq* produces v^∞. However, since v^∞ does not exist in *roll-out*(B), either (1) v itself is absent from B or (2) one of the call-sites from *Seq* is absent from B. These two possibilities form the basis for the two parts of the homogeneity test: the *absent-vertex test* and the *absent-call-site test.*

5.5.2. The Absent-Vertex Test

The absent-vertex test is applied to all vertices v in M that are absent from B (*e.g.*, the vertex representing the statement "$t := 1/x$" in Figure 5.6). Vertex v's absence from B implies that no occurrences of v exist in *roll-out*(B), which has the following implications on the occurrences of v in M^∞ and *roll-out*(M): first, the occurrences of v in M^∞ are the same as those in Δ^∞(*roll-out*(A), *roll-out*(*Base*)), and second, as shown in Chapter 6, the occurrences of v in *roll-out*(M) are a subset of those in *roll-out*(A). Adding the relation stated above that the occurrences in M^∞ are a subset of those in *roll-out*(M), yields the following inequality (*occ* abbreviates *occurrences*):

$$occ(v, \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))) = occ(v, M^\infty) \subseteq occ(v, roll\text{-}out(M)) \subseteq occ(v, roll\text{-}out(A)).$$

If we can show that $occ(v, \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))) = occ(v, roll\text{-}out(A))$ then this inequality becomes

$$occ(v, \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))) = occ(v, M^\infty) = occ(v, roll\text{-}out(M)) = occ(v, roll\text{-}out(A)).$$

In particular, it implies that $occ(v, M^\infty) = occ(v, roll\text{-}out(M))$.

The following lemma provides a condition sufficient to prove that $occ(v, \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base)))$ equals $occ(v, roll\text{-}out(A))$. It is shown in Chapter 6, that this condition is also necessary.

LEMMA. (EVERY OCCURRENCE LEMMA). *If* $v \in blfl\,DAP^S(A, Base)$ *then* *roll-out*(A) *and* Δ^∞(*roll-out*(A), *roll-out*(*Base*)) *contain the same occurrences of* v.

PROOF. The occurrences of v in Δ^∞(*roll-out*(A), *roll-out*(*Base*)) are, by definition, in *roll-out*(A); thus, we must show that if $v \in blfl\,DAP^S(A, Base)$ then every occurrence of v from *roll-out*(A) is in Δ^∞(*roll-out*(A), *roll-out*(*Base*)). To this end, let u be a directly affected point such that $v \in blfl\,(A, u)$. We show below that, for every occurrence v^∞ of v, there exists an occurrence u^∞ of u such that

[4] The reason for this is a general property of the HPR algorithm: if a vertex exists in both A and B, it is in M (if the vertex is not in $\Delta^{hpr}(A, Base)$ or $\Delta^{hpr}(B, Base)$ then it must be in $Pre^{hpr}(A, Base, B)$). For *Integrate*$^\infty$ this property implies that vertex occurrences in both *roll-out*(A) and *roll-out*(B) are in M^∞. Thus, for an occurrence to be in *roll-out*(M) but not in M^∞ (*i.e.*, to be an extra occurrence) it must not exist in both *roll-out*(A) and *roll-out*(B).

$v^\infty \in b^\infty f^\infty (roll\text{-}out(A), u^\infty)$. It is shown in Chapter 6 that every occurrence of a directly affected point from A is a directly affected point in $roll\text{-}out(A)$; therefore, $u^\infty \in DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. Combining these two results, v^∞ is in $b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that v^∞ is in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, since $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ is defined as $b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

The existence of an occurrence u^∞ such that $v^\infty \in b^\infty f^\infty (roll\text{-}out(A), u^\infty)$ follows because both *bl* and *fl* slices "ascend" to calling procedures; thus, the procedure containing v transitively calls the procedure containing u through a sequence of call-sites *Seq*. In the roll-out of A, the expansion of the call-sites in *Seq* places an occurrence of u into a scope transitively enclosed in the scope containing each occurrence of v. Because v^∞ and u^∞ are connected by occurrences of the edges that connect v and u this implies $v^\infty \in b^\infty f^\infty (roll\text{-}out(A), u^\infty)$.

□

Thus, the absent-vertex test tests whether all vertices of M that are absent from B are in $bl\,fl\,DAP^S(A, Base)$ (*i.e.*, it tests if $V(G_M) - V(G_B) \subseteq bl\,fl\,DAP^S(A, Base)$). If this inclusion does not hold then, as proven in Chapter 6, $roll\text{-}out(M)$ contains an extra occurrence and M^∞ is inhomogeneous.

Example. In Figure 5.6, the vertex representing the statement "$t := 1/x$" is an absent vertex (it is in M but not B). Therefore, all occurrences of this vertex in M^∞ must come from $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. In the example, $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ contains the first occurrence from *roll-out*(*A*, *Main*), but it does not contain the second occurrence from *roll-out*(*A*, *Main*) or the occurrence from *roll-out*(*A*, *P*); consequently, M^∞ is inhomogeneous. The absent-vertex test detects the existence of extra occurrences of "$t := 1/x$"; thus, this example fails to pass the absent-vertex test.

5.5.3. The Absent-Call-Site Test

For each call-site $Call_P$ in M that is absent from B, the absent-call-site test is applied to the statements of the procedures transitively callable in M from P. In $roll\text{-}out(M)$, each expansion of $Call_P$ (and the call-sites in P) produces a P-scope that is not in $roll\text{-}out(B)$ (because $Call_P$ is not in B). Thus, to be in M^∞, the vertex occurrences in such scopes (*i.e.*, occurrences of the statements from procedure P and the procedures transitively callable from P) must be in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

Example. The call statement "**call** $Q(y)$" in M of Figure 5.7 is absent from B and the statement "$x := x+1$" of procedure R is in a procedure transitively callable in M from Q. Because procedure R is callable along two different sequences of call-sites, *roll-out*(*M*, *P*) contains two occurrences of "$x := x+1$," neither of which is in *roll-out*(*B*, *P*) (see Figure 5.7); thus, to be in M^∞ these occurrences must be in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

There are two ways a vertex occurrence can be included in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. First, the Every Occurrence Lemma implies that all of v's occurrences are in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ if v is in $bl\,fl\,DAP^S(A, Base)$. Second, even if v is not in $fl\,bl\,DAP^S(A, Base)$, it is possible for *some* of v's occurrences to be in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ because of certain directly affected points outside of the procedure containing v (and its transitively called procedures). This second case is complicated because, in the call-graph for M,[5] each path from a procedure to the procedure containing v produces an occurrence of v. Thus, each path from the procedure that contains a call-site in M that is not in B produces an occurrence that must be in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. As introduced below, *slice-need sets* are used by the *slice-need-set* part of the absent-call-site test to track occurrences of v *backwards* along the

[5] The call-graph for system M includes one node for each procedure in M and one directed edge from the node representing P to the node representing Q for each call-site $Call_Q$ in P.

Base	*A*	*B*	*M*	*roll-out(M, P)* *roll-out(A, P)*	*roll-out(B, P)*

procedure *Main*
  call *P*
end
	procedure *Main*
	  call *P*
	end
		procedure *Main*
		  call *P*
		end
			procedure *Main*
			  call *P*
			end
				procedure *P*
				  scope *Q*(*a* := *y*;
				  y := *a*)
				  a := 10
				  b := 20
				  scope *R* (*x* := *a*;
				  a := *x*)
				  x := *x*+1
				  epocs
				  scope *R* (*x* := *b*;
				  b := *x*)
				  x := *x*+1
				  epocs
				  c := *b*
				  epocs
				return
					procedure *P*
					
					return

procedure *P*
  call *Q* (*y*)
return
	procedure *P*
	  call *Q* (*y*)
	return
		procedure *P*
		  □
		return
			procedure *P*
			  call *Q* (*y*)
			return		

procedure *Q* (*a*)
  a := 1
  b := 2
  call *R* (*a*)
  call *R* (*b*)
  c := *b*
return
	procedure *Q* (*a*)
	  a := 10
	  b := 20
	  call *R* (*a*)
	  call *R* (*b*)
	  c := *b*
	return
		procedure *Q* (*a*)
		  a := 1
		  b := 2
		  call *R* (*a*)
		  call *R* (*b*)
		  c := *b*
		return
			procedure *Q* (*a*)
			  a := 10
			  b := 20
			  call *R* (*a*)
			  call *R* (*b*)
			  c := *b*
			return		

procedure *R* (*x*)
  x := *x*+1
return
	procedure *R* (*x*)
	  x := *x*+1
	return
		procedure *R* (*x*)
		  x := *x*+1
		return
			procedure *R* (*x*)
			  x := *x*+1
			return		

Figure 5.7. An example that passes the absent-call-site test. (The box indicates the call in *M* that is absent from *B*.)

paths in the call-graph to the procedure that contains the call-site in *M* that is absent from *B*.[6] For M^{∞} to be homogeneous, it is necessary to encounter a directly affected point along each path that causes the particular occurrence to be included in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Only if directly affected points are encountered along *all* paths back to $Call_P$ then *all* occurrences of v in the scope produced by expanding $Call_P$ (and the call-sites in *P*) are in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

Thus, for every call-site $Call_P$ in *M* that is absent from *B* and every vertex v in *P* or a procedure transitively callable from *P* in *M*, the absent-call-site test tests that v is either in $blfl\,DAP^S(A, Base)$ or in *A* where it does not cause the slice-need-set part of the absent-call-site test (described below) to determine failure. If neither of these conditions holds then, as shown in Chapter 6, an occurrence of v in $roll\text{-}out(M)$ is extra (*i.e.*, not in M^{∞}) and M^{∞} is inhomogeneous.

The complete description of the slice-need sets, given in the following subsection, requires some additional terminology and notation, which we develop in tandem with the description. After the description, we present an algorithm for efficiently computing slice-need sets and follow this (in the next section) with the algorithm that implements the homogeneity test. But first, we intuitively (and incompletely) describe slice-need sets and then list three factors that complicate the (complete) formal definition.

Slice-Need Sets

The *slice-need set* for procedure *R* is a set of subsets of *R*'s linkage vertices (linkage vertices are the formal-in, entry, and formal-out vertices of a procedure dependence graph). The elements of this set are referred to as *summary elements*. If v's summary element contains formal-in vertex $Enter_R x_{in}$ then the

[6] The slice-need-set approach can be viewed as a bottom-up approach: it propagates information about potential extra occurrences in *P* "up" to call-sites on *P*, and then to call-sites on the procedures containing these call-sites, and so on. Perhaps a more natural approach is a top-down approach, in which information about the slices taken with respect to directly affected points is propagated "down" to called procedures, then to their called procedures, and so on; however, the top-down approach has potentially exponential cost, and hence the absent-vertex test is based on the bottom-up approach.

occurrences of v produced by the expansion of a call-site on R are in the $b^{\infty} \circ f^{\infty}$ slice of *roll-out*(A) taken with respect to the corresponding occurrence of $Enter_R.x_{in}$ (*i.e.*, the occurrence of $Enter_R.x_{in}$ produced by the expansion of the same sequence of call-sites as the occurrence of v). Similarly, if the summary element for vertex v contains the entry vertex then the same relationship holds. Finally, if v's summary element contains formal-out vertex $Enter_R.x_{out}$ then the occurrences of v produced by the expansion of a call-site on R are in the b^{∞} slice of *roll-out*(A) with respect to the corresponding occurrence of $Enter_R.x_{out}$.

Example. In Figure 5.7, the summary element for the statement "$x := x+1$" in the slice-need set for procedure R is $\{Enter_R, Enter_R.x_{in}, Enter_R.x_{out}\}$. $Enter_R$ and $Enter_R.x_{in}$ are included in this set because the $b^{\infty} \circ f^{\infty}$ slice with respect to an occurrence of $Enter_R$ or $Enter_R.x_{in}$ will include the corresponding occurrence of v. Similarly, $Enter_R.x_{out}$ is included in this set because the b^{∞} slice with respect to an occurrence of $Enter_R.x_{out}$ will include the corresponding occurrence of v. These relationships can be easily verified from the procedure dependence graph shown in Figure 5.8. (Because R contains no call statements, the procedure dependence graph for each scope created by expanding a call-site on R is identical to the procedure dependence graph shown in Figure 5.8.)

The vertices in the summary element for v are useful in the absent-call-site test because they are related to the inclusion of occurrences of v in Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$). (In the following discussion recall that Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$) is defined as $b^{\infty}f^{\infty}DAP^{\infty}$(*roll-out*$(A)$, *roll-out*$(Base)$).) First, suppose that v's summary element contains $Enter_R.x_{in}$ (the case for $Enter_R$ is similar). If, for a call-site $Call_R$, *every* occurrence of $Call_R.x_{in}$ is in $f^{\infty}DAP^{\infty}$(*roll-out*(A), *roll-out*$(Base)$) then *every* occurrence of v produced by the expansion of $Call_R$ in *roll-out*(A) is in $b^{\infty}f^{\infty}DAP^{\infty}$(*roll-out*$(A)$, *roll-out*$(Base)$)).[7] (Recall that, as discussed above, all occurrences of vertices whose summary elements contain $Enter_R.x_{in}$ are in the

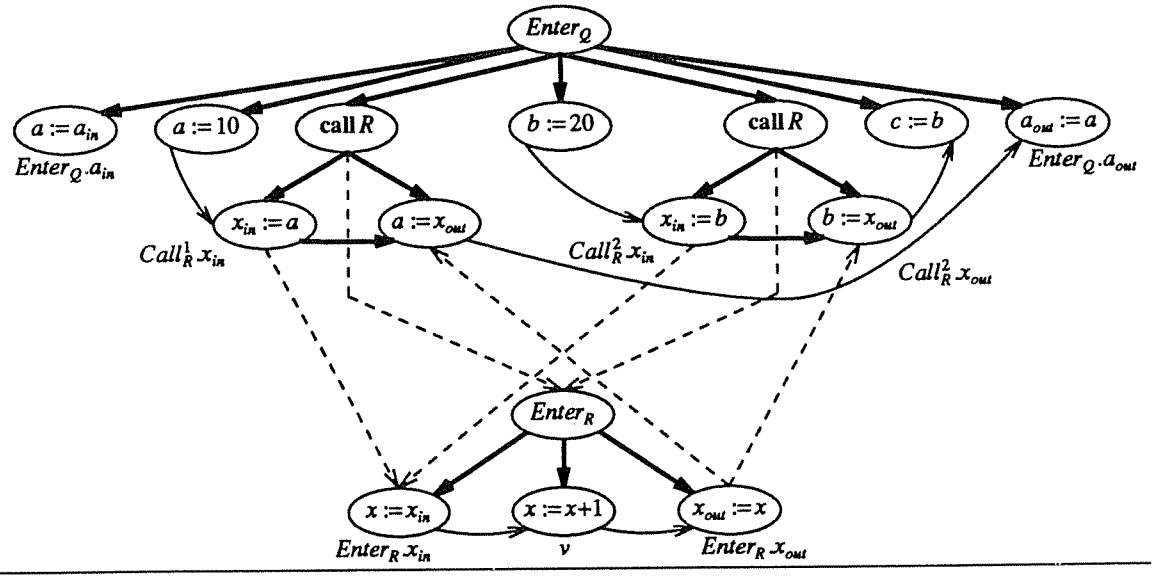


Figure 5.8. The procedure dependence graphs for procedures Q and R of system M from Figure 5.7. (The vertex names used in the text (*e.g.*, $Enter_R.x_{in}$) are shown outside the corresponding vertex. The vertex names for the call-sites on R are differentiated by the superscripts 1 and 2—*e.g.*, $Call_R^1$ is the call-site vertex for "call $R(a)$").

$b^{\infty} \circ f^{\infty}$ slice of $roll\text{-}out(A)$ taken with respect to the corresponding occurrence of $Enter_R.x_{in}$.) Similarly, if v's summary element contains $Enter_R.x_{out}$ and *every* occurrence of $Call_R.x_{out}$ is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ then *every* occurrence of v produced by the expansion of $Call_R$ in $roll\text{-}out(A)$ is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. (Recall that, as discussed above, all occurrences of vertices whose summary element contains $Enter_R.x_{out}$ are in the b^{∞} slice of $roll\text{-}out(A)$ taken with respect to the corresponding occurrence of $Enter_R.x_{out}$.)

Example. In Figure 5.7, occurrences of the directly affected points "$a := 10$" and "$b := 20$" cause *every* occurrence of $Call_R.x_{in}$ to be in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Therefore, since $Enter_R.x_{in}$ is in the summary element for v, $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ contains the occurrences of v produced by the expansion of the call-sites on R. For the example shown in Figure 5.7, this implies that the potentially extra occurrences of v are not extra, since having them in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ implies they are in M^{∞}.

Example. As a second example, consider what would happen if statement "$c := b$", rather than statement "$b := 20$", is a directly affected point. In this case, the occurrences produced by the expansion of $Call_R$ are still included in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Like the previous example, the occurrences of v produced by the expansion of the first call-site on R are included, since occurrences of "$a := 10$" cause occurrences of $Call^1_R.x_{in}$[8] to be in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Unlike the previous example, the occurrences of v produced by the expansion of the second call-site are included since occurrences of "$c := b$" cause the occurrences of $Call^2_R.x_{out}$ to be in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies the occurrences of v are in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ because v's summary element contains $Enter_R.x_{out}$.

In the preceding examples it is the presence of directly affected points "$a := 10$", "$b := 20$", and "$c := b$" that cause the potentially extra occurrences of v to be included in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and therefore not be extra. In general, testing for such directly affected points is complicated by the following three factors:

(1) Representatives for directly affected points.

In Figure 5.7 directly affected points "$a := 10$" and "$b := 20$" are in procedure Q, the procedure that calls procedure R. This is not strictly necessary; Q need only contain a *representative* of a directly affected point. For example, in Figure 5.7 if "$a := 10$" is replaced by "**call** $S(a)$" where procedure S contains a directly affected point that affects the value of a after the call then the actual-out vertex for a is a representative of the directly affected point in S. Similar to the original example—where all occurrences of $Call^1_R.x_{in}$ are in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ because all occurrences of "$a := 10$" are in $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$—in the modified example, all occurrences of $Call^1_R.x_{in}$ are in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ because all occurrences of the actual-out vertex for the call on S are in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Representatives of directly affected points are defined as *DAP-connected vertices* in the next sub-section.

(2) Propagation through a sequence of call-sites.

The summary element for vertex v in procedure R's slice-need set may be propagated to the slice-need sets of procedures that call procedure R. For the example shown in Figure 5.7 this possibility allows occurrences of directly affected points from P to cause $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ to include the two occurrences of "$x := x+1$" that result from the expansion of $Call_Q$ and the subsequent

---

[7] The occurrence is actually in $b^{\infty}f^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$; however, since f^{∞} is idempotent (an operator op is idempotent if $op(op(x))=op(x)$), this simplifies to $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which is, by definition, $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Operator f^{∞} is idempotent because it is a transitive closure. Also note that b^{∞} is idempotent. This is important when v's summary element contains formal-out vertex $Enter_R.x_{out}$.

[8] Superscripts for the two calls on R in Q denote the first and second calls—*e.g.* $Call^1_R$ is the call-site vertex for "**call** $R(a)$".

expansions of the two call-sites on R in Q. In general, summary elements can be propagated though a sequence of call-sites to (a representative of) a directly affected point. However, this sequence cannot contain the absent call-site. An additional complication, introduced by this propagation, is that it is possible for v to have multiple summary elements in the slice-need set of a procedure that transitively calls R because in A's call graph multiple paths may connect a calling procedure to R.

(3) Combining backward and forward slices.

In the computation of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ a backward slice always follows a forward slice (recall that $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ is defined as $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$); thus, a slice-need involving a backward slice can be transformed into a slice-need involving a forward slice. This possibility complicates the test because it must be modeled in the computation of slice-need sets.

We now give the formal definition for slice-need sets. As stated above, the slice-need set for procedure P is a set of subsets of P's linkage vertices. These sets are formalized in a manner similar to the definition of other interprocedural summary information (*e.g.*, *GMOD*). Such definitions are structured as a set of equations expressed in terms of initial values associated with each procedure and functions that determine how this information is transferred from procedure to procedure. For example, in the definition of *GMOD*, the initial information is *IMOD*, the set of variables known to be modified without examining called procedures, and the transfer functions simply map formal parameter names to actual parameter names at call-sites. The final solution to such problems is the least fixed point of a set of equations. For example, the definition of *GMOD* is the least fixed point of the following set of equations (one for each procedure P):

$$GMOD(P) = IMOD(P) \cup \bigcup_{Call_Q \in P} map\_formals\_to\_actuals(Call_Q, GMOD(Q))$$

The slice-need set for each procedure, denoted by SN_P for procedure P, is determined in an analogous manner. First, an initial-slice-need set and a transfer function are computed for each procedure P (these are denoted by ISN_P and TF_P, respectively). Then final slice-need sets are defined as the least fixed point of the following set of equations (one for each procedure P):

$$SN_P = ISN_P \cup \bigcup_{Call_Q \in P} TF_P(Call_Q, SN_Q).$$

We now define ISN_P and TF_P. First, ISN_P summarizes the "slice-needs" of vertices in P of system M but not those of vertices in procedures transitively called by P. The elements of ISN_P are subsets of P's linkage vertices. The subset that summarizes a vertex v includes the following:

(1) Every formal-in vertex $Enter_P.x_{in}$ such that $v \in bl\,fl\,(P, Enter_P.x_{in})$[9], since a $b^{\infty} \circ f^{\infty}$ slice in $roll\text{-}out(A)$ taken with respect to an occurrence of $Enter_P.x_{in}$ will include the occurrence of v from the same scope.

(2) $Enter_P$ (the entry vertex) because $v \in fl(P, Enter_P)$ due to the control edges of P.

(3) Every formal-out vertex $Enter_P.x_{out}$ such that $v \in bl(P, Enter_P.x_{out})$, since a b^{∞} slice in $roll\text{-}out(A)$ taken with respect to an occurrence of $Enter_P.x_{out}$ will include the occurrence of v from the same scope. (Note that $v \in bl(P, Enter_P.x_{out})$ is equivalent to $Enter_P.x_{out} \in fl(P, v)$.)

To formalize the initial-slice-need sets, we begin with the definition of *potential-problem vertices* (*i.e.*, vertices that may have extra occurrences in $roll\text{-}out(M)$ because of an absent call-site).

---

[9] Recall that *bl* and *fl* denote backward local and forward local slices, which consider only control, flow, summary, and meeting-point edges.

DEFINITION. (Potential-Problem Vertices).

$$PotentialProblemVertices \triangleq \{\, v \in V(G_M) \mid \exists \text{ a call-site } Call_P \text{ in } M \text{ that is absent from } B \wedge v \text{ is in } P \text{ or a procedure transitively callable in } M \text{ from } P \wedge v \notin bl\,fl\,DAP^S(A, Base) \,\}.$$

Recall that the Every Occurrence Lemma implies that if $v \in bl\,fl\,DAP^S(A, Base)$ then every occurrence of v is in M^∞, which is why such v's are not potential-problem vertices.

The second definition uses local slices in each procedure P to determine which linkage vertices of P summarize a vertex v from P. As an optimization in the definition, $Enter_P$ is always included because, for every vertex v in P, $Enter_P \in fl(P, v)$.

DEFINITION. (Connected Linkage Vertices).

$$Linkage_P(v) \triangleq \{\, Enter_P \,\} \cup \{\, Enter_P.a_{in} \mid Enter_P.a_{in} \in bl\,fl\,(P, v) \,\} \cup \{\, Enter_P.a_{out} \mid Enter_P.a_{out} \in fl\,(P, v) \,\}.$$

Finally, the initial-slice-need sets for procedure P contains the subsets of the linkage vertices from P that are connected to each potential-problem vertex in P.

DEFINITION. (Initial-Slice-Need Sets).

$$ISN_P \triangleq \{\, Linkage_P(v) \mid v \in V(G_P) \wedge v \in PotentialProblemVertices \,\}.$$

Example. For the example shown in Figure 5.7, the potential problem vertices are the vertices of procedure R, which are all summarized by the single summary element $\{ Enter_R.x_{in}, Enter_R, Enter_R.x_{out} \}$ (*i.e.*, $ISN_R = \{ \{ Enter_R.x_{in}, Enter_R, Enter_R.x_{out} \} \}$). Thus, (as shown in Figure 5.8) the forward local slice (followed by a backward local slice) with respect to $Enter_P.x_{in}$ or $Enter_P$ and the backward local slice with respect to $Enter_P.x_{out}$ each include all the vertices of R.

For the purposes of illustrating slice-need sets and other related concepts, Table 5.1 gives the summary elements for the vertices of M of Figure 5.7, and indicates which vertices are summarized by each summary element (the procedure dependence graphs from which the last four summary elements are computed are shown in Figure 5.8).

Summary Element	Vertices of M summarized by the element
S_1: $\{ Enter_{Main} \}$	all vertices in *Main*'s procedure dependence graph
S_2: $\{ Enter_P \}$	all vertices in P's procedure dependence graph
S_3: $\{ Enter_Q.a_{in}, Enter_Q \}$	$Enter_P.a_{in}$
S_4: $\{ Enter_Q, Enter_Q.a_{out} \}$	$Enter_Q.a_{out}$, "$a := 10$", and the three vertices representing "**call** $R\,(a)$"
S_5: $\{ Enter_Q \}$	"$b := 20$", "$c := b$", and the three vertices representing "**call** $R\,(b)$"
S_6: $\{ Enter_R.x_{in}, Enter_R, Enter_R.x_{out} \}$	all vertices in R's procedure dependence graph

Table 5.1

If all the vertices of M were potential problem vertices then the following would be the initial slice-need sets (in Figure 5.7, only the vertices from procedure R *are* potential problem vertices):

$$ISN_{Main} = \{ S_1 \} \qquad ISN_Q = \{ S_3, S_4, S_5 \}$$
$$ISN_P = \{ S_2 \} \qquad ISN_R = \{ S_4 \}$$

The next series of definitions formalizes the transfer functions. For each procedure P, and each call-site $Call_Q$ in P, TF_P updates SN_P from SN_Q. We begin by giving a simplified definition for TF_P that may cause the slice-need-set test to determine failure erroneously (*i.e.*, the test may determine failure even though M^∞ is homogeneous) because the resulting transfer function for P transfers every summary element of SN_Q through $Call_Q$ in P to SN_P. We then give the correct definition that, by taking into account the possibility that $S \in SN_Q$ may not need to be transferred to SN_P, transfers only necessary elements from SN_Q to SN_P. It

is shown in Chapter 6 that using the second definition the slice-need-set test determines failure iff *roll-out*(M) contains an extra occurrence due to an absent call-site.

The definition of TF_P is divided into two parts. The first maps a call-site $Call_Q$ and a set S of Q's linkage vertices to the corresponding set of actual-in, call-site, and actual-out vertices at $Call_Q$:

DEFINITION. (Map formals to actuals).

$$Map(Call_Q, S) \triangleq \textbf{if } Enter_Q \in S \textbf{ then } \{ Call_Q \} \textbf{ else } \emptyset \textbf{ fi } \cup \{ Call_Q.a_{in} \mid Enter_Q.a_{in} \in S \} \cup \{ Call_Q.a_{out} \mid Enter_Q.a_{out} \in S \}.$$

The second part of the definition uses local slices to propagate actual-in, call-site, and actual-out vertices associated with a call-site $Call_Q$ in procedure P to the linkage vertices of P. (As an optimization in the definition, $Enter_P$ is always included because, for every vertex v in P, $Enter_P \in fl(P, v)$).

DEFINITION. (Slice Transfer). For s, a single actual-in, call-site, or actual-out vertex associated with a call-site $Call_Q$ in procedure P, $Transfer_P(s)$ is defined as follows:

$$Transfer_P(s) \triangleq \{ Enter_P \} \cup \begin{cases} \{ Enter_P.a_{in} \mid Enter_P.a_{in} \in bl(P, s) \} & \text{if } s = Call_Q \text{ or } s = Call_Q.x_{in} \\ \{ Enter_P.a_{in} \mid Enter_P.a_{in} \in bl\, fl(P, s) \} & \text{if } s = Call_Q.a_{out} \\ \{ Enter_P.a_{out} \mid Enter_P.a_{out} \in fl(P, s) \} & \text{if } s = Call_Q.a_{out}. \end{cases}$$

This definition is extended to S, a *set* of actual-in, call-site, and actual-out vertices associated with a call-site $Call_Q$ in procedure P, as follows:

$$Transfer_P(S) \triangleq \bigcup_{s \in S} Transfer_P(s).$$

Example. Consider an actual-in vertex $Call_Q.x_{in}$ such that $Enter_P.a_{in} \in bl(P, Call_Q.x_{in})$ (note that this is equivalent to $Call_Q.x_{in} \in fl(P, Enter_P.a_{in})$). Let v^∞ be an occurrence of a vertex from procedure Q or a procedure transitively called by Q. If v^∞ is in the $b^\infty \circ f^\infty$ slice taken with respect to an occurrence of $Call_Q.x_{in}$ then, because $Call_Q.x_{in} \in fl(P, Enter_P.a_{in})$, v^∞ is also in the $b^\infty \circ f^\infty$ slice taken with respect to the occurrence of $Enter_P.a_{in}$ in the same scope as the occurrence of $Call_Q.x_{in}$. If it were the case that $Enter_P.a_{in}$ were (a representative of) a directly affected point, then this would imply v^∞ was in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) and therefore not extra.

Finally, putting the two parts of the transfer function together yields the following:

DEFINITION. (Transfer Functions).

$$TF_P(Call_Q, SN_Q) \triangleq \{ Transfer_P(Map(Call_Q, S)) \mid S \in SN_Q \}.$$

Example. The transfer functions for procedures *Main*, P, Q, and R of system M from Figure 5.7 are given in the following table. For example, TF_Q is computed from the procedure dependence graph for procedure Q (see Figure 5.8). As can be seen in Figure 5.8, for the first call-site on R, $Transfer_Q(\{ Call^1_R.x_{in}, Call^1_R, Call^1_R.x_{out} \})$ includes $Enter_Q$ (because of the control edges in Q) and $Enter_Q.a_{out}$ (because $Enter_Q.a_{out}$ is in $fl(Q, Call^1_R.x_{out})$). Notice that $Transfer_Q(\{ Call^1_R.x_{in}, Call^1_R \})$ includes only $Enter_Q$—even though $Enter_Q.a_{out}$ is in $fl(Q, Call^1_R.x_{in})$ and $fl(Q, Call^1_R)$—because we are only interested in forward slices with respect to actual-in and call-site vertices and the slices resulting from the connection between $Call^1_R.x_{in}$ (or $Call^1_R$) and $Enter_Q.a_{out}$ are backward slices.

Transfer Functions:

$TF_{Main}(Call_P, x) = \{ Enter_{Main} \}$ for all $x \in Main$	$TF_P(Call_Q, x) = \{ Enter_{Main} \}$ for all $x \in P$
$TF_Q(Call_R^1, \{ Enter_R.x_{in} \}) = \{ Enter_Q \}$	$TF_Q(Call_R^2, \{ Enter_R.x_{in} \}) = \{ Enter_Q \}$
$TF_Q(Call_R^1, \{ Enter_R \}) = \{ Enter_Q \}$	$TF_Q(Call_R^2, \{ Enter_R \}) = \{ Enter_Q \}$
$TF_Q(Call_R^1, \{ Enter_R.x_{out} \}) = \{ Enter_Q, Enter_Q.x_{out} \}$	$TF_Q(Call_R^2, \{ Enter_R.x_{out} \}) = \{ Enter_Q \}$
$TF_R(Call, x) = \emptyset$ (there are no calls in procedure R)	

The above functions are for singleton sets, the functions for larger sets can be obtained using the following rule:

$$TF(Call, \{ a, b \}) = TF(Call, \{ a \}) \cup TF(Call, \{ b \}).$$

As mentioned above, this definition is incorrect because, although it transfers the necessary elements from SN_Q through $Call_Q$ to SN_P, it may transfer too many elements. In other words this definition fails to take into account the possibility that the "slice-needs" summarized by summary element $S \in SN_Q$ may be satisfied at call-site $Call_Q$ in P; in this case, S should not be transferred to SN_P.

Recall that $S \in SN_Q$ represents vertices having occurrences in $roll\text{-}out(M)$ that may be extra (not in M^∞) because of the expansion of a call-site on Q. If $Map(Call_Q, S)$ contains an actual-in, call-site, or actual-out vertex that is *DAP-connected* (defined below) then all occurrences of the vertices summarized by S that result from the expansion of $Call_Q$ are guaranteed to exist in M^∞. Furthermore, if $Call_Q$ is in P then this implies that all summarized occurrences that result from the expansion of a call-site on P also exist in M^∞; thus, S should not be transferred to SN_P.

DAP-connected vertices are actual-in and call-site vertices in $fl\,DAP^S(A, Base)$ and actual-out vertices in $bl\,fl\,DAP^S(A, Base)$. The following relationships show why DAP-connected vertices can be used as representatives of directly affected points in the slice-need-set test (these relationships are proven in Chapter 6).

(1) First, for a DAP-connected vertex d in A, every occurrence of d is in either $f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ or $b^\infty f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$: if d is an actual-in or call-site vertex then every occurrence of d is in $f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$; if d is an actual-out vertex then every occurrence of d is in $b^\infty f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

(2) Second, suppose vertex v is summarized by $S \in SN_Q$ where v is propagated to S through a sequence of call-sites Seq. The occurrence v^∞ of v produced by expanding a call-site $Call_Q$ and then the call-sites in Seq is in the $b^\infty \circ f^\infty$ or b^∞ slice of $roll\text{-}out(A)$ taken with respect to an occurrence of *each* vertex in $Map(Call_Q, S)$: if $Map(Call_Q, S)$ contains $Call_Q$ or $Call_Q.x_{in}$ then v^∞ is in the $b^\infty \circ f^\infty$ slice of $roll\text{-}out(A)$ with respect to an occurrence of $Call_Q$ or $Call_Q.x_{in}$, respectively; if $Map(Call_Q, S)$ contains $Call_Q.x_{out}$ then v^∞ is in the b^∞ slice of $roll\text{-}out(A)$ with respect to an occurrence of $Call_Q.x_{out}$.

To combine these relationships, suppose that $Map(Call_Q, S)$ from Relationship (2) includes a DAP-connected vertex d. If d is an actual-in or call-site vertex then Relationship (1) implies every occurrence of d is in $f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and Relationship (2) implies that for some occurrence d^∞ of d, $v^\infty \in b^\infty f^\infty(roll\text{-}out(A), d^\infty)$; thus, composing these slices, v^∞ is in $b^\infty f^\infty f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. On the other hand, if we suppose that d is an actual-out vertex then Relationship (1) implies every occurrence of d is in $b^\infty f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and Relationship (2) implies that for some occurrence d^∞ of d, $v^\infty \in b^\infty(roll\text{-}out(A), d^\infty)$; thus, composing these slices, v^∞ is in $b^\infty b^\infty f^\infty\,DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

To show that v^∞ is in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ requires the idempotence of f^∞ and b^∞ (an operator op is idempotent if $op(op(x)) = op(x)$). Operators f^∞ and b^∞ are idempotent because they are both transitive closures; thus, we can replace $f^\infty f^\infty$ and $b^\infty b^\infty$ with f^∞ and b^∞, respectively. This simplifies both

$b^{\infty} f^{\infty} f^{\infty} DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $b^{\infty} b^{\infty} f^{\infty} DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ to $b^{\infty} f^{\infty} DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Since this is the definition of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, it means (in both cases) that v^{∞} is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

The definition for TF_P that avoids unnecessary transfers incorporates the following definition of DAP-connected vertices.

DEFINITION. (DAP-connected Vertices).

$$DAPConnected(S) \triangleq \bigvee_{s \in S} \begin{cases} s \in fl\, DAP^{S}(A, Base) & \text{if } s \text{ is of the form } Call_Q.a_{in} \\ s \in fl\, DAP^{S}(A, Base) & \text{if } s \text{ is of the form } Call_Q \\ s \in bl\, fl\, DAP^{S}(A, Base) & \text{if } s \text{ is of the form } Call_Q.a_{out} \end{cases}$$

Example. The DAP-connected vertices from system A of Figure 5.7 are $Call_R^1.x_{in}$, $Call_R^1.x_{out}$, $Call_R^2.x_{in}$, and $Call_R^2.x_{out}$. For this example, these are determined solely from the procedure dependence graph for procedure Q shown in Figure 5.8 because the directly affected points "$a := 10$" and "$b := 20$" are also in procedure Q.

The complete definition for TF_P incorporates a check for DAP-connected vertices.

DEFINITION. (Transfer Functions that account for DAP-connected vertices).

$$TF_P(Call_Q, SN_Q) \triangleq \{\, Transfer_P(Map(Call_Q, S)) \mid S \in SN_Q \wedge \neg DAPConnected(Map(Call_Q, S)) \,\}.$$

The final definition is for slice-need sets. In addition to the elements of the initial-slice-need set, the final slice-need set for each procedure contains elements that summarize vertices from *transitively* called procedures. Thus, the slice-need sets are the least fixed point of the following set of equations (one for each procedure P in M):

DEFINITION. (Slice-Need Sets).

$$SN_P = ISN_P \cup \bigcup_{Call_q \in P} TF_P(Call_Q, SN_Q).$$

Computing Slice Need Sets

Initial slice-need sets and the transfer functions are computed from the procedure dependence graphs of M using local slices. The (final) slice-need sets are computed by the function *UpdateSNs*, shown in Figure 5.9. This function is called with SN_P initialized to ISN_P for each procedure P of M (see line [10] in Figure 5.10).

The algorithm implemented by *UpdateSNs* makes use of a worklist to avoid redundant computations; each element of a slice-need set appears in the worklist at most once. Summary element S from SN_Q is processed by this algorithm in lines [5] through [9] by first checking at each call-site on procedure Q (line [5]) whether the set of actual-in, call-site, and actual-out vertices identified by the linkage vertices in S contains a DAP-connected vertex (line [7]). If a DAP-connected vertex is in this set then all summarized occurrences are in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and hence S should not be propagated to the calling procedure's slice-need set. Otherwise, if this set contains no DAP-connected vertices then the calling procedure's transfer function is used to transfer S to a set of linkage vertices from the calling procedure (procedure P in lines [8] and [9] of the algorithm). This new element is added to SN_P and the worklist, provided it does not already exist in SN_P. (A comprehensive example that illustrates the operation of this algorithm is given at the end of the next section.)

```
        function UpdateSNs(SNS, TFS)
        declare
          SNS : set of slice-need-sets
          TFS : set of transfer functions
          SN_P : a slice-need-set
          Call_Q : a call-site vertex
          S : set of linkage vertices
          worklist : set of slice-need-set elements
        begin
[ 1]      worklist := { S | S ∈ SN_P ∧ SN_P ∈ SNS }
[ 2]      while worklist ≠ ∅ do
[ 3]        remove S from worklist
[ 4]        Let S be in SN_Q
[ 5]        for each call-site Call_Q do
[ 6]          Let Call_Q be in procedure P
[ 7]          if ¬DAPConnected(Map(Call_Q, S)) then
[ 8]            if TF_P(Call_Q, S) ∉ SN_P then
[ 9]              Add TF_P(Call_Q, S) to SN_P and worklist
                fi
              fi
            od
          od
[10]      return (the updated SNS)
        end
```

Figure 5.9. The function *UpdateSNs* propagates elements of the initial slice-need-sets until a DAP-connected vertex at a call-site is encountered or no further updating is possible.

5.5.4. The Homogeneity Test Algorithm

The homogeneity test is implemented by the algorithm shown in Figure 5.10. The bulk of the algorithm is the function *ExtraOccurrences*, which is called twice: once for vertices of M that are not in B and once for vertices of M that are not in A. If no extra occurrences are discovered on either call then $roll\text{-}out(M)$ contains no extra occurrences and, as proven in Chapter 6, is equal to M^∞, which implies that M^∞ is homogeneous.

The function *ExtraOccurrences* implements the two parts of the homogeneity test. The absent-vertex test (line [4] of Figure 5.10) tests whether there are extra occurrences in $roll\text{-}out(M)$ because of absent vertices; the absent-call-site test (lines [5] - [13]) tests whether there are extra occurrences in $roll\text{-}out(M)$ because of absent call-sites. The absent-vertex test simply checks that vertices absent from B are in $blfl\,DAP^S(A, Base)$. The absent-call-site text has two parts: a trivial *subset test* and the *slice-need-set test*. For both tests, the set PP computed in line [5] contains all vertices reachable in M from an absent call-site. Because all occurrences of vertices in $blfl\,DAP^S(A, Base)$ are in M^∞, these vertices are removed from PP in line [6]. For occurrences of the remaining vertices to be in M^∞ they must come from $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. This requires that these vertices exist in A (the subset test in line [7]), and that they not cause the slice-need-set test (lines [8] - [13]) to fail in line [13]. If the function *ExtraOccurrences* returns **true** then the test has discovered an extra occurrence in $roll\text{-}out(M)$ and M^∞ is therefore inhomogeneous. Otherwise, if no extra occurrences exist in $roll\text{-}out(M)$, then, as shown in Chapter 6, $roll\text{-}out(M) = M^\infty$ and hence M^∞ is homogeneous.

```
        function IsHomogeneous(G_M, A, B, Base) returns Boolean
        declare
          G_M : the merged system dependence graph
          A, B, Base : systems
        begin
[ 1]      if ExtraOccurrences(G_M, (V(G_M) − V(B)), A, Base) then return (false) fi
[ 2]      if ExtraOccurrences(G_M, (V(G_M) − V(A)), B, Base) then return (false) fi
[ 3]      return (true)
        end

        function ExtraOccurrences(G_M, Absent, A, Base) returns Boolean
        declare
          G_M : the merged system dependence graph
          Absent : the set of system dependence graph vertices in G_M but not in one of the variants
          A, Base : systems
          PP : set of potential problem vertices
          SNs : set of slice-need-sets
          TFs : set of transfer functions
          S : set of linkage nodes
        begin
[ 4]      if Absent ⋤ blfl DAP^S(A, Base) then return (true) fi
[ 5]      PP := { v ∈ V(G_M) | ∃ call-site Call_Q ∈ Absent and v is in Q or a procedure (transitively) called from Q }
[ 6]      PP := PP − blfl DAP^S(A, Base)
[ 7]      if PP ⋤ V(A) then return (true) fi
[ 8]      SNs := ConstructInitialSNs(G_M, PP)
[ 9]      TFs := ConstructTFs(G_M)
[10]      SNs := UpdateSNs(SNs, TFs)
[11]      for each call-site Call_Q ∈ Absent do
[12]        for each S ∈ SN_Q do
[13]          if ¬DAPConnected(Map(Call_Q, S)) then return (true) fi
            od
          od
[14]      return (false)
        end
```

Figure 5.10. The function *IsHomogeneous* performs the two checks of the homogeneity test. The auxiliary function *ExtraOccurrences* applies the absent-vertex test (the first line of the function) and then the two parts of the absent-call-site test (the remainder of the function).

Example. For the integration example shown in Figure 5.6, the absent-vertex test fails: the vertex labeled "$t := 1/x$" is in $V(G_M) - V(G_B)$ but is not in $blfl\ DAP^S(A, Base)$. Intuitively, the homogeneity test fails because A and B make conflicting use of procedure P. The modification made to A requires both statements from the body of P to be included in M; however, the modification made to B clashes with the need for statement "$t := 1/x$" in A's modification.

A Comprehensive Example

As a second example, consider the application of the homogeneity test to the integration example shown in Figure 5.11 (this figure repeats systems *Base*, *A*, and *B* from Figure 5.3 and system *M* from Figure 5.5). Like the example shown in Figure 5.6, this example fails the homogeneity test; however, unlike that example, it is the absent-call-site test that fails rather than the absent-vertex test. (This is the desired outcome

Base

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

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

  x := 2
return

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

Variant *A*

```
procedure Main
  [a := 1]
  b := 1
  call P(a, 1)
  call P(b, 1)
end(a)

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
```

Variant *B*

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

procedure P(x, y)
  t1 := x + y
  [          ]

  x := 2
return

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

M

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

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
```

roll-out(Base)

```
program Main
  a := 2
  b := 1
  scope P(x := a, y := 1;
          a := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs

  x := 2
  epocs
  scope P(x := b, y := 1;
          b := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs

  x := 2
  epocs
end(a)
```

roll-out(A)

```
program Main
  a := 1
  b := 1
  scope P(x := a, y := 1;
          a := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs
  t3 := x
  x := 2
  epocs
  scope P(x := b, y := 1;
          b := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs
  t3 := x
  x := 2
  epocs
end(a)
```

roll-out(B)

```
program Main
  a := 2
  b := 1
  scope P(x := a, y := 1;
          a := x)
    t1 := x + y

  x := 2
  epocs
  scope P(x := b, y := 2;
          b := x)
    t1 := x + y

  x := 2
  epocs
end(a)
```

M^{∞}

```
program Main
  a := 1
  b := 1
  scope P(x := a, y := 1;
          a := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs

  x := 2
  epocs
  scope P(x := b, y := 2;
          b := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)

      z := z + 1
    epocs

  x := 2
  epocs
end(a)
```

roll-out(M)

```
program Main
  a := 1
  b := 1
  scope P(x := a, y := 1;
          a := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs
  t3 := x
  x := 2
  epocs
  scope P(x := b, y := 2;
          b := x)
    t1 := x + y
    scope Incr(z := x;
               x := z)
      t2 := 1/z
      z := z + 1
    epocs
  t3 := x
  x := 2
  epocs
end(a)
```

Figure 5.11. An example that illustrates the absent-call-site test of the homogeneity test. This figure repeats the three systems shown in Figure 5.3 and the merged system shown in Figure 5.5. It also includes the roll-outs of these systems. (In the roll-outs, only the roll-out of the main procedures are shown.)

since, as can be seen in Figure 5.11, M^{∞} is inhomogeneous.)

We present two traces in Figures 5.12 and 5.13 that show how the homogeneity test determines that M^{∞} is inhomogeneous. Figure 5.12 shows the trace of the function *ExtraOccurrences*. The computation of the slice-need sets for this example is shown in Figure 5.13. For example, consider the second to last iteration of the loop in lines [2] through [9] of Figure 5.9 where S_3 is removed from the worklist (see Figure 5.13).

$Absent = \{\ a := 1,\ Call_{Incr}.x_{in},\ Call_{Incr},\ Call_{Incr}.x_{out},\ t3 := x\ \}$

This set is a subset of $bIfI\ DAP^S(A, Base)$; therefore, the absent-vertex test does not fail.

$PP = \{\ Enter_{Incr}.z_{in},\ Enter_{Incr},\ Enter_{Incr}.z_{out},\ t2 := 1/z,\ z := z + 1\ \}$

PP is a subset of $V(A)$; therefore, the trivial part of the absent-call-site test does not fail.

The initial and final slice-need sets and transfer functions are shown in Figure 5.13.

$Call_{Incr}$ is the only call-site in *Absent*; $SN_{Incr} = \{\ S_5, S_6\ \}$.

For S_6, $Map(Call_{Incr}, S_6) = \{\ Call_{Incr}.z_{in},\ Call_{Incr},\ Call_{Incr}.z_{out}\ \}$.

$Call_{Incr}.z_{out}$ is DAP-connected to a directly affected point; therefore, the necessary occurrences of the vertices summarized by S_6 are all included in G_M-.

For S_5, $Map(Call_{Incr}, S_5) = \{\ Call_{Incr}.z_{in},\ Call_{Incr}\ \}$.

Neither $Call_{Incr}.a_{in}$ nor $Call_{Incr}$ is DAP-connected to a directly affected point; therefore, the absent-call-site test and thus the homogeneity test fail because occurrences of the vertices summarized by S_5 are missing from G_M-.

Figure 5.12. A trace of the homogeneity test applied to $V(G_M) - V(B)$ for the integration of the systems shown in Figure 5.11. In this case, the test correctly determines that (as shown in Figure 5.11) the HPR integration of *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) is inhomogeneous.

Slice-Need Set Elements:

Summary Element	Vertices of M summerized by the element
S_1: { $Enter_{Main}$ }	all vertices in *Main*'s procedure dependence graph
S_2: { $Enter_P.x_{in}$, $Enter_P.y_{in}$, $Enter_P$ }	$t1 := x+y$, $Enter_P.x_{in}$, and $Enter_P.y_{in}$,
S_3: { $Enter_P.x_{in}$, $Enter_P$ }	$Call_{Incr}.x_{in}$, $Call_{Incr}$, $Call_{Incr}.x_{out}$, and $t3 := x$
S_4: { $Enter_P$, $Enter_P.x_{out}$ }	$Enter_P.x_{out}$ and$x := 2$
S_5: { $Enter_{Incr}.z_{in}$, $Enter_{Incr}$ }	$t2 := 1/z$
S_6: { $Enter_{Incr}.z_{in}$, $Enter_{Incr}$, $Enter_{Incr}.z_{out}$ }	$z := z+1$, $Enter_{Incr}.z_{in}$ and $Enter_{Incr}.z_{out}$

$S_1, \cdots, S_6$ represent all possible elements for the slice-need sets computed from M shown in Figure 5.11.

Initial Slice-Need Sets:

$ISN_{Main} = \emptyset \quad ISN_P = \emptyset \quad ISN_{Incr} = \{ S_5, S_6 \}$

Transfer Functions:

$TF_{Main}(Call, x) = \{ Enter_{Main} \}$ for all $x \in Main$

$TF_P(Call_{Incr}, \{ Enter_{Incr}.z_{in} \}) = \{ Enter_P, Enter_P.x_{in} \}$
$TF_P(Call_{Incr}, \{ Enter_{Incr} \}) = \{ Enter_P \}$
$TF_P(Call_{Incr}, \{ Enter_{Incr}.z_{out} \}) = \{ Enter_P, Enter_P.x_{in} \}$

$TF_{Incr}(Call, x) = \emptyset$ (there are no calls in procedure *Incr*)

These functions transfer singleton sets; for larger sets the transfer function can be obtained using the following rule:

$$TF(Call, \{ a, b \}) = TF(Call, \{ a \}) \cup TF(Call, \{ b \}).$$

Slice-Need Set Update:

Procedure	*ISN*	After processing S_5	After processing S_6	After processing S_3 and S_1
Main	$\emptyset$	$\emptyset$	$\emptyset$	$\{ S_1 \}$
P	$\emptyset$	$\{ S_3 \}$	$\{ S_3 \}$	$\{ S_3 \}$
Incr	$\{ S_5, S_6 \}$	$\{ S_5, S_6 \}$	$\{ S_5, S_6 \}$	$\{ S_5, S_6 \}$
Worklist	S_5, S_6	S_6, S_3	S_3	$\emptyset$

Final Slice-Need Sets:

$ISN_{Main} = \{ S_1 \} \quad ISN_P = \{ S_3 \} \quad ISN_{Incr} = \{ S_5, S_6 \}$

Figure 5.13. The computation of the slice-need sets for the application of the homogeneity test to the systems shown in Figure 5.11.

Since S_3 summarizes vertices in P and there are two calls on P in M, the loop at line [5] goes through two iterations. During the first (when call "**call** $P(a)$" is processed), $Map(Call_P^1, S_3)$ contains the actual-in vertex for a, which is a DAP-connected point; thus, no new element is added to SN_{Main} or the worklist. However, $Map(Call_P^2, S_3)$ contains no DAP-connected vertices and therefore a new element (S_1) is added to SN_{Main} and the worklist.

5.6. Reconstituting a System from the Merged System Dependence Graph

The final 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 system dependence graph is G_M. There are two possible reasons why it may be impossible to reconstitute a system from the merged system dependence graph. First, G_M may be pdg-infeasible (*i.e.*, one of the procedure dependence graphs may be infeasible, this is Type II interference as found by the HPR algorithm). Second, even if all of the system dependence graph's

procedure dependence graphs are feasible, G_M may be sdg-infeasible (*i.e.*, the system dependence graph may still be infeasible due to procedure linkage constraints). Sdg-infeasibility occurs when there is a parameter-vertex mismatch (*i.e.*, two call-sites on the same procedure have different sets of actual parameter vertices).

As shown in Chapter 6, if G_M has passed the homogeneity test it can only be infeasible if one of its procedure dependence graphs is pdg-infeasible. Pdg-infeasibility is tested for by applying the program-reconstitution algorithm from [Horwitz89][10] to each procedure dependence graph 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 procedure dependence graph is infeasible (and thus that G_M is infeasible); in this case, there is Type II interference and *Integrate*^{S} reports failure.

If there is no Type I interference, if the homogeneity test succeeds, and if G_M is feasible, then a system M whose system dependence graph is G_M is returned as the result of the integration.

5.7. Recap of *Integrate*^{S}

Putting all the pieces together, a complete algorithm for multi-procedure integration appears in Figure 5.14.

---

function *Integrate*^{S}(*A*, *Base*, *B*) **returns** a system or Failure
declare
  A, *Base*, *B*, *M* : systems
  G : a system dependence graph
begin
  $G := \Delta^S(A, Base) \cup \Delta^S(B, Base) \cup Pre^S(A, Base, B)$
  if *TypeI_Interference*(*A*, *Base*, *B*, *G*) **then return (Failure) fi**
  if not *IsHomogeneous*(*G*, *A*, *B*, *Base*) **then return (Failure) fi**
  if *TypeII_Interference*(*G*) **then return (Failure) fi**
  M := *ReconstituteSystem*(*G*)
return (*M*)

Figure 5.14. 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 and the integration of the roll-outs of *Base*, *A*, and *B* is homogeneous, *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 $G_M = G$).

[10] A corrected version of the algorithm is given in [Ball90].

CHAPTER 6

Integrate^S SATISFIES THE SYNTACTIC REQUIREMENTS ON I^S

This chapter demonstrates that *Integrate*^{S}, the multi-procedure integration algorithm developed in Chapter 5, satisfies the syntactic requirements on I^S from Version 2 of the Revised Model of Program Integration given in Section 4.3 (henceforth, simply "the model"). This is done under the assumption that I^∞ is *Integrate*$^\infty$ (the natural extension of the HPR algorithm). Recall that the model places the following requirements on *Integrate*$^\infty$ and *Integrate*^{S}:

(1: a semantic requirement)
the "conceptual" integration operation *Integrate*$^\infty$ must satisfy the original HPR model, extended to sets of infinite programs, and

(2: a syntactic requirement)
the concrete integration operation *Integrate*^{S} must
(i) deal with finite systems,
(ii) succeed in producing M iff *Integrate*$^\infty$ succeeds in producing M^∞, and
(iii) be consistent with *Integrate*$^\infty$ (*i.e.* *roll-out*(M) must equal M^∞).

In addition, to relate the semantics of M^∞ and M, it is required that *roll-out* be a semantics-preserving transformation.

These requirements can be viewed in terms of the diagram shown in Figure 6.1 as follows. The syntactic proof in this chapter essentially states that this diagram commutes. (Informally stated, it shows that

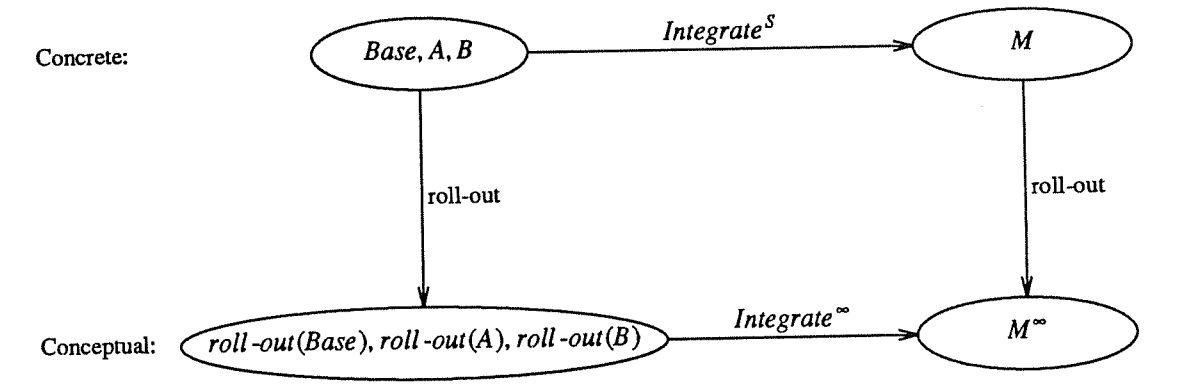


Figure 6.1. The commutative square that captures Version 2 of the Revised Model for Multi-Procedure Integration.

Integrate$^{\infty} \circ$ *roll-out* equals *roll-out* $\circ$ *Integrate*^{S}.) The semantic proofs in Chapters 7 and 8 show that M captures the changed and preserved behavior of A and B with respect to *Base*. First, Chapter 7 proves that *roll-out* is a semantics-preserving transformation; this relates the meanings of a system at the concrete level with the meaning of its roll-out at the conceptual level. Second, it is shown in Chapter 8 that integration at the conceptual level satisfies the HPR integration model (extended to allow infinite programs). In other words, Chapter 8 shows that M^{∞} (the result of integrating *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*)) captures the changed and preserved behavior of *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*). Chapter 7, concerns the *semantic* relationship between *Base*, A, and B and *roll-out*(*Base*), *roll-out*(A), and *roll-out*(B), respectively. Because Chapter 6 shows that *roll-out*(M)$=M^{\infty}$ (which is a *syntactic* relationship between M and M^{∞}), Chapter 6 and 7 together tell us about the *semantic* relationship between M and M^{∞}.

Hence, in accordance with the preceding discussion, this chapter proves that *Integrate*^{S} satisfies Requirement (2) of the model. To begin with, by design *Integrate*^{S} satisfies Requirement (2)(i); thus, we demonstrate, in the Syntactic Correctness Theorem of Section 6.5, that *Integrate*^{S} satisfies Requirements (2)(ii) and (2)(iii).

The proof of the Syntactic Correctness Theorem is based on the Sufficiency, Necessity, and Type I Interference Theorems; proofs of these theorems appear in Sections 6.2, 6.3, and 6.4 respectively. (An overview of the structure of the entire proof is provided in Figure 6.2; the way that this chapter is organized is that the theorems and lemma listed in Figure 6.2 are proven in left-to-right depth-first order). Before presenting the Sufficiency Theorem, Section 6.1 gathers together some of the terminology and notation used in the proof.

6.1. Terminology and Notation

The Result of Successful and Unsuccessful Integrations

This section defines the terms *successful integration* and *unsuccessful integration* and defines the merged programs that result from each. To begin with, *Integrate*^{S} is *successful* if there is no Type I or Type II interference; for a successful integration the result is a system M constructed from *Base*, A, and B as described in Chapter 5 (note that this means *Integrate*^{S} produces a merged system even if the homogeneity test detects inhomogeneity). Otherwise, if there is Type I or Type II interference then integration is *unsuccessful*, in which case M is defined to be the empty program. Similarly, *Integrate*$^{\infty}$ is *successful* when applied to *roll-out*(*Base*), *roll-out*(A), and *roll-out*(B) if there is no Type I or Type II interference; for a successful integration the result is a set of integrated procedures M^{∞}. Otherwise, if there in Type I or Type II interference then integration is *unsuccessful*, in which case M^{∞} is defined to contain only the empty program. Even for unsuccessful integrations, both operations produce an intermediate merged dependence graph: throughout this section, we continue to use G_M to denote the intermediate system dependence graph produced by *Integrate*^{S} and $G_{M^{\infty}}$ to denote the set of intermediate graphs produced by *Integrate*$^{\infty}$.

Representation of Parameter Transfer

To understand the proof presented in this chapter, it is essential to understand the relationship between the representation of call statements in G_A and the representation of scope statements in $G_{roll\text{-}out(A)}$. To understand this relationship, it is first necessary to understand how the transfer of values associated with the execution of call statements in A and scope statements in *roll-out*(A) are represented in G_A and $G_{roll\text{-}out(A)}$, respectively (an example of these representations is shown of Figure 6.3).

A:

In system A, parameters to a procedure call are passed by value-result: initial actual-parameter values

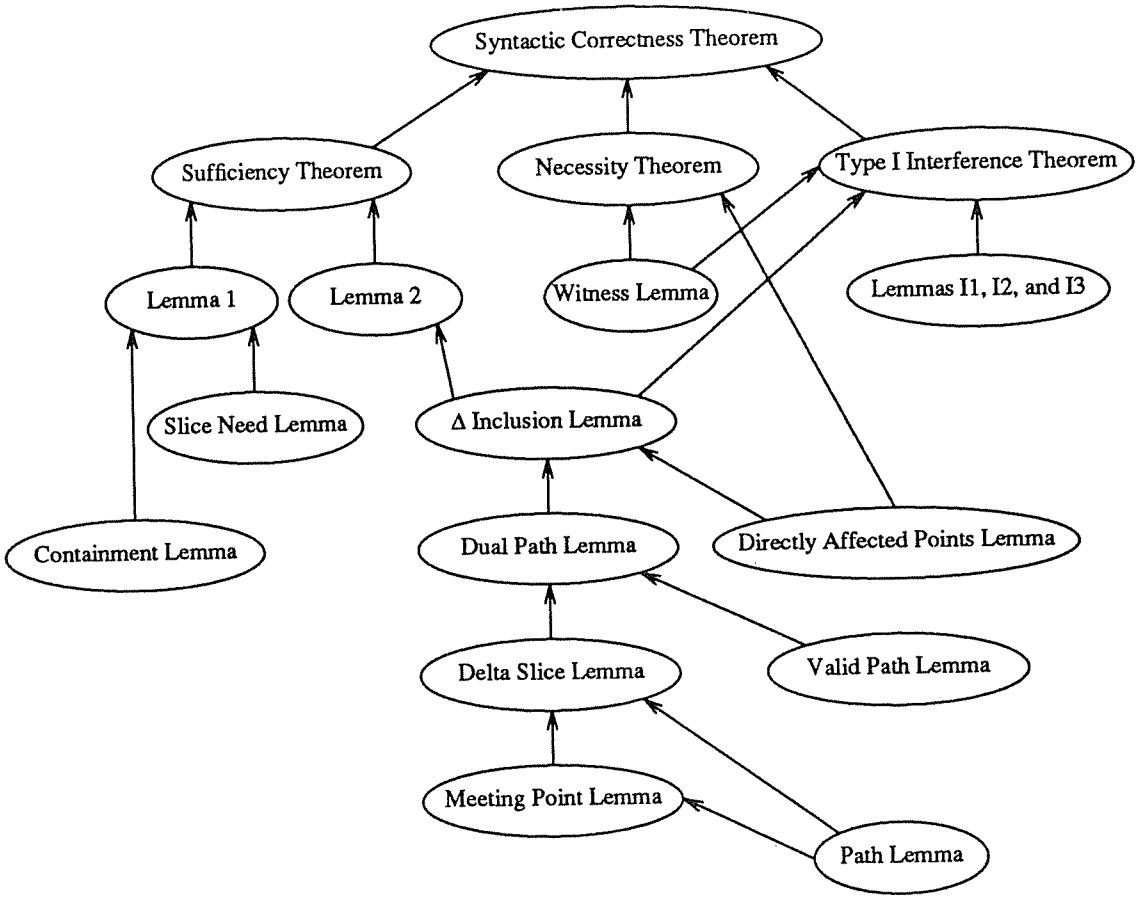


Figure 6.2. A pictorial breakdown of the proof that *Integrate*^{S} satisfies requirements (2)(ii) and (2)(iii) of the Integration Model of Section 4.3.

are copied to formal parameters before calling a procedure; result values are then copied back to actual parameters after the procedure completes execution.

G_A:

By breaking down the transfer of values into actions taken by the calling procedure and actions taken by the called procedure, G_A represents the passing of parameter values at a finer level of granularity than described for the passing of parameters in A. As described in Chapter 3, the transfer of initial actual-parameter values to a called procedure is represented by two kinds of vertices: actual-in vertices, which represent the transfer by the calling procedure of initial values from actual parameters to intermediate temporary variables (*e.g.* x_{in} in Figure 6.3), and formal-in vertices, which represent the transfer by the called procedure of initial values from the intermediate temporaries to the formal parameters. After the call, the transfer of result values from formal parameters (through different intermediate temporaries, *e.g.*, the variable x_{out} in Figure 6.3) back to the actual parameters is also represented by vertices of two kinds: formal-out and actual-out vertices.

roll-out(A):

In *roll-out*(A), each scope statement sets up a new name space. To transfer values into and out of this name space, a scope statement is parameterized by two kinds of assignment statements: *transfer-in* statements, which transfer initial (actual-parameter) values to variables in the new name space (previously the formal parameters of the called procedure) and *transfer-out* statements, which transfer result (formal-parameter) values back to variables of the enclosing name space (previously the actual parameters of the call statement). Thus, transfer-in and transfer-out statements make explicit the transfer of values dictated by value-result parameter passing.

$G_{roll\text{-}out(A)}$:

Unlike G_A, in $G_{roll\text{-}out(A)}$, a single vertex represents the value transfer associated with entering or exiting a scope. A *transfer-in* vertex represents the transfer accomplished by a transfer-in statement and a *transfer-out* vertex represents the transfer accomplished by a transfer-out statement.

The relationship between the representation of a call statement in G_A and the representation of a scope statement in $G_{roll\text{-}out(A)}$ is illustrated in Figure 6.3, which shows a call on procedure P, its expansion, and the graphical representation for each. In general, the representation of a scope in $G_{roll\text{-}out(A)}$ "condenses" the representation in G_A of a call statement and the vertices representing the interface to the called procedure as follows: corresponding actual-in and formal-in vertices are condensed into a single transfer-in

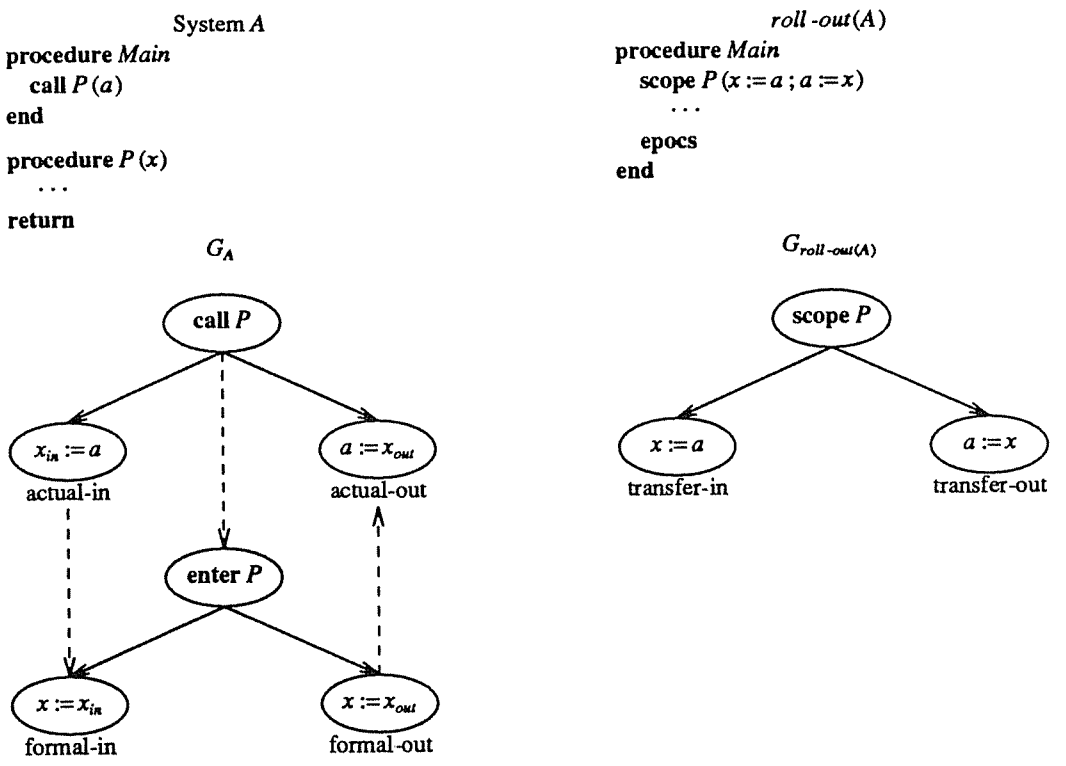


Figure 6.3. This figure shows a system A, *roll-out*(A), and fragments of their respective dependence graphs. Notice that in G_A the two vertices that represent the two-step process by which values are transferred to and from P (using intermediate temporaries x_{in} and x_{out}, respectively), are condensed into a single assignment vertices in $G_{roll\text{-}out(A)}$.

vertex, corresponding actual-out and formal-out vertices are condensed into a single transfer-out vertex, and the call-site and entry vertices are condensed into a single scope vertex. This is illustrated in Figure 6.3 where the six vertices representing a call on P and the interface to P are condensed into three vertices representing the P-scope replacing the call on P. In addition, the edge occurrences in $G_{roll\text{-}out(A)}$ associated with the condensation of two vertices from G_A can be determined from the edges associated with the two vertices. For example, referring to Figure 6.3, the incoming edges for the transfer-in vertex labeled "$x := a$" are occurrences of the incoming edges for the actual-in vertex labeled "$x_{in} := a$" and the edges whose source is the transfer-in vertex labeled "$x := a$" are occurrences of edges that have the formal-in vertex labeled "$x := x_{in}$" as their source.

Occurrences of Vertices

This section formalizes the notion of the *occurrences* of a vertex and the inverse of this mapping. Excluding actual-in, call-site, and actual-out vertices, the vertices of G_A partition the vertices of $G_{roll\text{-}out(A)}$. Furthermore, because of the condensation of actual-in and formal-in vertices, call-site and entry vertices, and actual-out and formal-out vertices, the actual-in, call-site, and actual-out vertices further partition the partition associated with each formal-in, entry, and formal-out vertex, respectively. Thus, for vertex occurrence v^∞ in $G_{roll\text{-}out(A)}$, v^∞ corresponds to a single vertex v from G_A if v^∞ is not a transfer-in, scope, or transfer-out vertex; otherwise, v^∞ corresponds to a pair of vertices in G_A. For example, a transfer-in vertex corresponds to an actual-in vertex and a formal-in vertex.

Before formalizing the correspondence (or mapping) between the vertices of G_A and the vertex occurrences in $G_{roll\text{-}out(A)}$, we identify a distinguished occurrence of each vertex v from G_A. This occurrence is the one associated with the expansion of the empty sequence of call-sites (if v is in procedure P then the distinguished occurrence of v is in *roll-out*(A, P)). Because the statement corresponding to this occurrence has scope-nesting-depth zero (*i.e.* is nested within zero scope statements), it is referred to as the *depth-zero* occurrence.

DEFINITION (Occurrences). The following table defines the mapping *occurrences* for a vertex v in G_A and the inverse mapping for a vertex occurrence v^∞ in $G_{roll\text{-}out(A)}$. These mappings are determined by the type of v and v^∞, respectively.

Type of v	*occurrences*(*roll-out*(A), v)
actual-in	Let "*Call* $P(\cdots, p, \cdots)$" be the call statement at which v represents parameter p. The occurrences of v are the transfer-in vertices that represent the transfer-in statements of *roll-out*(A) obtained from p and the expansion of any sequence of call-sites ending in "*Call* $P(\cdots, p, \cdots)$".
call-site	Similar to actual-in.
actual-out	Similar to actual-in.
formal-in	The occurrences of v are the union of the occurrences from $G_{\textit{roll-out}(A)}$ of all actual-in vertices from G_A that are connected to the formal-in vertex by an interprocedural edge in G_A, plus the depth-zero occurrence of v.
entry	Similar to formal-in.
formal-out	Similar to formal-in.
Type of v^{∞}	Inverse of *occurrences* (*i.e.* v such that $v^{\infty} \in$ *occurrences*(*roll-out*(A), v)).
transfer-in	v^{∞} is an occurrence of the actual-in and formal-in vertices in G_A condensed to form v^{∞}.
scope	v^{∞} is an occurrence of the call-site and entry vertices in G_A condensed to form v^{∞}.
transfer-out	v^{∞} is an occurrence of the actual-out and formal-out vertices in G_A condensed to form v^{∞}.
others	v^{∞} is an occurrence of the unique vertex v from G_A of which v^{∞} is a copy.[1]

In the remainder of Chapter 6, if v^{∞} is a transfer-in, scope, or transfer-out vertex then the phrase "corresponding vertex in G_A" refers to the actual-in, call-site, or actual-out vertex of which v^{∞} is an occurrence, unless it is explicitly stated that v^{∞} is an occurrence of a formal-in, entry, or formal-out vertex, respectively.

Homogeneity

In Section 4.3, we informally defined a set of programs that contained scope statements as *inhomogeneous* if "different occurrences of a procedure (*i.e.* scopes) contain different subsets of a procedure's parameters and statements." While programs satisfying this statement are inhomogeneous, there are programs that do not satisfy the statement that are nonetheless inhomogeneous. The formal definition of *homogeneous*, rules out such programs:

DEFINITION. A set of programs P is *homogeneous* iff there exists a system S (without scope statements) such that G_P is isomorphic[2] to $G_{\textit{roll-out}(S)}$.

Informally, for each procedure Q, the definition requires that all Q-scopes in P and the procedure corresponding to *roll-out*(S, Q) (the roll-out that begins with procedure Q) be the same (*i.e.*, their dependence graph must be isomorphic). Note that in the proofs, because *roll-out*(S, Q) must be the same as the

[1] This can be further formalized using the tags of v and v^{∞}.

[2] Two sets of graphs S_1 and S_2 are isomorphic iff there is a 1-to-1 mapping f from S_1 onto S_2 such that, for every graphs $G_1 \in S_1$, G_1 and $f(G_1)$ are isomorphic graphs; two graphs G_1 and G_2 are isomorphic iff the following conditions are satisfied:
(1) There is a 1-to-1 mapping g from the vertex set of G_1 onto the vertex set of G_2 and for every v in G_1, v and $g(v)$ have the same text.
(2) There is a 1-to-1 mapping h from the edge set of G_1 onto the edge set of G_2 and for every edge e in G_1, e and $h(e)$ are of the same type (*e.g.* both control edges, or both flow edges, etc.) and have the same label.
(3) For every edge $v \longrightarrow u$ in G_1, $h(v \longrightarrow u) = g(v) \longrightarrow g(u)$.
When S_1 and S_2 are isomorphic or when we are trying to prove S_1 and S_2 are isomorphic, for brevity, we will say v and $g(v)$ are the *same* vertex and e and $h(e)$ are the *same* edge.

Q-scopes, *roll-out*(S, Q) is treated as a Q-scope

Additional Notation

NOTATION. The symbol "$\simeq$" denotes an isomorphism between two graphs. Hence, one goal of Chapter 6 is to show that $G_{roll\text{-}out(M)} \simeq G_{M^{\infty}}$.

NOTATION. P_v denotes the procedure containing the vertex v.

NOTATION. x^{∞} denotes an occurrence of x (if unspecified in the text, the rolled-out system is implied by context).

NOTATION. Table 1 summarizes the kinds of edges that exist in a system dependence graph (v and u denote the system dependence graph vertices).

Type of Edge		Denoted by
Any of the following edge kinds		$v \longrightarrow u$
Intraprocedural Edges	control	$v \longrightarrow_{c} u$
	flow	$v \longrightarrow_{f} u$
	def-order	$v \longrightarrow_{do(w)} u$ (w is the *witness* vertex)
	summary	$v \longrightarrow_{s} u$
	meeting-point	$v \longrightarrow_{mp} u$
Interprocedural Edges	call	$v \longrightarrow_{call} u$
	parameter-in	$v \longrightarrow_{in} u$
	parameter-out	$v \longrightarrow_{out} u$

Table 1

All edges are *incoming* edges for vertex u except a def-order edge, which because it is viewed as a hyperedge from v to u to w, is an incoming edge for the witness vertex w.

A path in G_A from v to u (which is obtained from the reflexive transitive closure of the G_A's edge relation) is denoted by $v \longrightarrow^{*} u$. Paths in G_A made up of only certain kinds of edges are denoted by the reflexive transitive closure of particular sub-relations of G_A's edge relation. For example, $v \longrightarrow^{*}_{c,\,call} u$ denotes a path from v to u composed of only control dependence and interprocedural call edges.

NOTATION. We use the symbol $\subseteq$, when applied to dependence graphs, to mean "is a sub-graph of". Similarly, when applied to sets of dependence graphs x and y, $x \subseteq y$ means that for each dependence graph G_x in x there is a corresponding dependence graph G_y in y such that $G_x \subseteq G_y$. For example, $G_{roll\text{-}out(M)} \subseteq G_{M^{\infty}}$ means that for each procedure P in M there is a procedure dependence graph in $G_{M^{\infty}}$ that contains all the vertices and edges from $G_{roll\text{-}out(M,\,P)}$.

Finally, as in previous sections, we continue to omit selection operations when they are clear from context. For example, $v \in bl(P, u)$ continues to abbreviate $v \in SelectVertexSet(bl(P, u))$.

6.2. The Sufficiency Theorem

The Sufficiency Theorem, which is proven by mutual containment, demonstrates that the homogeneity test is sufficient—that is, whenever the test is passed, $G_{M^{\infty}}$ is guaranteed to be homogeneous and isomorphic to $G_{roll\text{-}out(M)}$. The proof has two parts: Lemma 1, which shows that whenever the test is passed and no Type I interference exists, $G_{roll\text{-}out(M)} \subseteq G_{M^{\infty}}$, and Lemma 2, which shows that whenever no Type I or Type II interference exists, $G_{M^{\infty}} \subseteq G_{roll\text{-}out(M)}$. Together Lemmas 1 and 2 imply that whenever the homogeneity test is passed and no Type I or Type II interference exists, $G_{roll\text{-}out(M)} \simeq G_{M^{\infty}}$ (which, because *roll-out*(M) is homogeneous, implies that M^{∞} is homogeneous).

6.2.1. Lemma 1

We first show, in Lemma 1, that if *Base*, *A*, *B*, and G_M pass the homogeneity test and no Type I interference exists then $G_{roll\text{-}out(M)} \subseteq G_{M^-}$ (*i.e.* $G_{roll\text{-}out(M)}$ is contained in G_{M^-}). The proof of Lemma 1 makes use of two supporting results: the Containment Lemma, which bounds the vertices and edges of $G_{roll\text{-}out(M)}$, and the Slice-Need Lemma, which relates the inclusion of certain occurrences of v in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, and thus in G_{M^-}, with the DAP-connection of a vertex in $Map(Call, S)$, where S summarizes vertex v.

The Containment Lemma

LEMMA (CONTAINMENT LEMMA). *If Base, A, B, and G_M pass the homogeneity test and no Type I interference exists in the integration of Base, A, and B then $roll\text{-}out(M) \subseteq roll\text{-}out(A) \cup roll\text{-}out(B)$.*

PROOF. The proof is by contradiction. We first consider the vertices and then the edges of $roll\text{-}out(M)$.

Vertices

Assume $v^\infty \in V(roll\text{-}out(M))$ but not $V(roll\text{-}out(A)) \cup V(roll\text{-}out(B))$. Vertex v^∞ can be absent from $roll\text{-}out(A)$ because either v or one of the call-sites whose expansion leads to v^∞ in $roll\text{-}out(M)$ is absent from A; vertex v^∞ is absent from $roll\text{-}out(B)$ for the same reasons. There are two cases to consider:

(1) If $v \notin V(A)$ and $v \notin V(B)$ then $v \notin V(G_M)$, and hence $v^\infty \notin V(roll\text{-}out(M))$, a contradiction.

(2) We assume, without loss of generality, that $v \in V(A)$ and that v occurs in the procedure dependence graph for procedure P. Because $v^\infty \notin V(roll\text{-}out(A))$ but $v \in V(A)$, $roll\text{-}out(A)$ must not contain the P-scope that contains v^∞ in $roll\text{-}out(M)$. Thus, one of the call-sites in M whose expansion leads to v^∞ in $roll\text{-}out(M)$ does not exist in A; let $Call_Q$ be this call-site. Because $v^\infty \notin V(roll\text{-}out(B))$, either v is not in B or a call-site $Call_R$ whose expansion leads to v^∞ in $roll\text{-}out(M)$ is not in B. We consider these two possibilities separately. First, if v is not in B then the homogeneity test fails (which contradicts our assumption that *Base*, *A*, *B*, and G_M pass the homogeneity test). The test fails because $Call_Q$ is an absent call-site from A, v is in a procedure (transitively) callable from $Call_Q$ in M, but v is not in B (see line [7] of Figure 5.10).

Otherwise, if v is in B then $Call_R$ is not in B. Call-sites $Call_Q$ and $Call_R$ cannot be the same call-site because if they were, then by part (1) $Call_Q$ would not be in M and hence v^∞ would not be in $roll\text{-}out(M)$ (which contradicts our assumption that $v^\infty \in V(roll\text{-}out(M))$). Thus in G_M, since $Call_Q$ and $Call_R$ are both part of the sequence of call-sites whose expansion produces v^∞, either $Call_Q$ is in a procedure transitively callable from $Call_R$ or $Call_R$ is in a procedure transitively callable from $Call_Q$. The proof for both of these cases is the same as the case when v is absent from B. For example, if $Call_R$ is in a procedure transitively callable from $Call_Q$ then, because $Call_Q$ is an absent call-site from A, $Call_R$ is in a procedure (transitively) callable from $Call_Q$ in M, but $Call_R$ is not in B, the homogeneity test fails (which contradicts our assumption that *Base*, *A*, *B*, and G_M pass the homogeneity test).

Edges

Assume $e^\infty \in E(roll\text{-}out(M))$ but not $E(roll\text{-}out(A)) \cup E(roll\text{-}out(B))$. As above we consider two cases.

(1) If $e \notin E(A)$ and $e \notin E(B)$ then $e \notin E(G_M)$ and hence $e^\infty \notin E(roll\text{-}out(M))$, a contradiction.

(2) We assume, without loss of generality, that $e \in E(A)$ and occurs in the procedure dependence graph for procedure P. As in Case (2) above, there is a call-site $Call_Q$ that is absent from A and either e is absent from B, which is considered below, or $Call_R$ is absent from B, in which case the proof is subsumed by Case (2) above.

When e is absent from B, we consider the target of e^{∞}; assume this is vertex y^{∞}. Vertex y^{∞} is, by assumption, in $roll\text{-}out(M)$; however, because $Call_Q$ is not in A, y^{∞} is not in $roll\text{-}out(A)$. Since the vertex argument implies $V(roll\text{-}out(M)) \subseteq V(roll\text{-}out(A)) \cup V(roll\text{-}out(B))$, y^{∞} must be in $roll\text{-}out(B)$; therefore, y exists in B. Because call-site $Call_Q$ is absent from A, y is in a procedure (transitively) called by $Call_Q$ in M, and $Base$, A, B, and G_M pass the homogeneity test, y is either in $bl\,fl\,DAP^S(B, Base)$ or is summarized by an element of a slice-need set that is transferred to a DAP-connected set of vertices. Both of these imply that y is in $\Delta^S(B, Base)$ (the later case is detailed below). But $e \in E(G_M)$ and $e \notin E(B)$ imply that y is in $DAP^S(G_M, B)$; thus, $y \in \Delta^S(B, Base) \cap DAP^S(G_M, B)$, which leads to a contradiction because it violates the assumption that no Type I interference exists.

What remains to be shown is that if y is summarized by an element of a slice-need set that is transferred to a DAP-connected set of vertices then y is in either $b2\,bl\,fl\,DAP^S(B, Base)$ or $b2\,f2\,fl\,DAP^S(B, Base)$ and thus in $\Delta^S(B, Base)$. To begin with, a DAP-connected vertex is either an actual-out vertex in $bl\,fl\,DAP^S(B, Base)$ or an actual-in (or call-site) vertex in $fl\,DAP^S(B, Base)$. For an actual-out vertex the element initially summarizing y contains a formal-out vertex whose bl slice includes y. The formal-out vertex is transferred to the DAP-connected actual-out vertex because of a series bl slices that, taken with respect to formal-out vertices, include actual-out vertices. The kinds of edges traversed by a bl slice and the parameter-out edges that connect formal-out vertices to actual-out vertices are the kinds of edges traversed by a $b2$ slice; thus, y is the $b2$ slice with respect to the DAP-connected actual-out vertex and hence in $b2\,bl\,fl\,DAP^S(B, Base)$. The case for an actual-in or call-site vertex is identical except that y's initial summary element and its transfer may involve fl slices followed by bl slices. Thus y is in the $b2 \circ f2$ slice with respect to the DAP-connected actual-in (or call-site) vertex and hence in $b2\,f2\,fl\,DAP^S(B, Base)$.

□

The Slice-Need Lemma

The second result used by Lemma 1 is the Slice-Need Lemma. This lemma describes a sufficient condition for the inclusion of v^{∞} from $roll\text{-}out(A)$ in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and thus in G_M-. Each occurrence of v in $roll\text{-}out(M)$, including v^{∞}, can be associated with (the expansion of) a unique sequence of call-sites Seq from M. Occurrence v^{∞} is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ if this sequence contains a call-site $Call_P$ such that $DAPConnected(Map(Call_P, TF(\{ Linkage(v) \})))$ where function TF (as explained below) is the composition of the functions that result when the transfer functions of each procedure called between P and P_v are applied to the corresponding call-sites from Seq.

A transfer function has the signature $call\text{-}site \longrightarrow Slice\text{-}Need\text{-}Set \longrightarrow Slice\text{-}Need\text{-}Set$. If TF_i denotes the transfer function for the procedure that contains call-site c_i then $TF_i(c_i)$ denotes a function, having signature $Slice\text{-}Need\text{-}Set \longrightarrow Slice\text{-}Need\text{-}Set$, that transfers slice-need sets through call-site c_i. Function TF, also having signature $Slice\text{-}Need\text{-}Set \longrightarrow Slice\text{-}Need\text{-}Set$, is the result of composing $TF_i(c_i)$ for the sequence of call-sites Seq_P—the suffix of Seq occurring after $Call_P$. This function transfers elements of P_v's slice-need set to P's slice-need set; thus in $DAPConnected(Map(Call_P, TF(\{ Linkage(v) \})))$, TF transfers $Linkage(v)$ (backwards) along the call-sites of Seq_P until, at $Call_P$, it is mapped to a DAP-connected set of actual-in, call-site, and actual-out vertices.

LEMMA (THE SLICE-NEED LEMMA). *Let v^{∞} be the occurrence of vertex v associated with the expansion of Seq, a sequence of call-sites from A. Let $Call_P$ be a call-site from Seq and let Seq_P denote the suffix of Seq that occurs after $Call_P$. If TF is the composition of $TF_i(c_i)$ for each call-site c_i in Seq_P and $DAPConnected(Map(Call_P, TF(\{ Linkage(v) \})))$ then $v^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.*

PROOF. Suppose l is one of the linkage vertices from $Linkage(v)$ that is transferred to a DAP-connected vertex dc by $Map(Call_P, TF(\{ Linkage(v) \}))$. The proof considers separate cases based on the types of l

and *dc*. In general there are nine cases since a DAP-connected vertex is either an actual-in, call-site, or actual-out vertex and there are three kinds of linkage vertices; however, the cases involving call-site and entry vertices are simplifications of the cases involving actual-in and formal-in vertices, respectively (this is because every vertex in G_P is in $fl(P, Enter_P)$ due to the control edges of G_P). Of the four remaining cases a formal-in vertex is never transferred by *TF* to an actual-out vertex (see Section 5.5.3); thus, only three cases remain.

In each case, we show that v^∞ is in $b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ (*i.e.*, $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$) by establishing the existence of a directly affected point u^∞ and two paths of edges $u^\infty \rightarrow^*_{c,f} x^\infty$ and $v^\infty \rightarrow^*_{c,f} x^\infty$ in $roll\text{-}out(A)$. (The edges of $u^\infty \rightarrow^*_{c,f} x^\infty$ are traversed by the f^∞ slice of Δ^∞; the edges of $v^\infty \rightarrow^*_{c,f} x^\infty$ are traversed by the b^∞ slice.) In all cases, u^∞ is an occurrence of the directly affected point u from A that causes *dc* to be a DAP-connected vertex.

Before considering the individual cases, we show that every occurrence of u, including u^∞, is in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. Directly affected points are determined solely from the incoming edges for a vertex (or by the existence of a vertex in A but not *Base*). This local information is not affected by the expansion of call-sites; thus, if u is in $DAP^S(A, Base)$ and therefore has different incoming edges in A and *Base*, every occurrences of u has different edge occurrences in $roll\text{-}out(A)$ and $roll\text{-}out(Base)$ and is therefore in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. (If u is a directly affected point because it is in A but not *Base* then, because no occurrences of u are in $roll\text{-}out(Base)$, all occurrences from $roll\text{-}out(A)$ are in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.)

We now consider the three cases based on the types of l and *dc*. In each case we identify three paths in $roll\text{-}out(A)$ that correspond to the paths in A between v and l, l and *dc*, and *dc* and u.

(1) l is a formal-in vertex and *dc* is an actual-in vertex.

The first path in $roll\text{-}out(A)$ corresponds to the path between v and l in A. To begin with, formal-in vertex l is in $Linkage(v)$ iff $v \in bl\,fl(P_v, l)$; thus, in P_v paths connect v and l to some vertex x. With two caveats, occurrences of the edges that make up these paths connect v^∞ and l^∞ (the occurrence of l in the same scope as v^∞) to some vertex x^∞. The first caveat is that if either path in P_v contains summary edges then the "occurrences" of these edges are the occurrences of the edges that induced the summary edges. The second caveat is that if either path in P_v contains a meeting-point edge then the "occurrences" of this edge are the occurrences of the edges that induce the meeting-point edge. Notice that x^∞ is an occurrence of x unless both paths in P_v end with meeting-point edges. In this case x^∞ is an occurrence of the vertex that induced the meeting-point vertex. Thus, the occurrences of the edges that make up $l \rightarrow^*_{c,f,s,mp} x$ and $v \rightarrow^*_{c,f,s,mp} x$ imply that the paths $l^\infty \rightarrow^*_{c,f} x^\infty$ and $v^\infty \rightarrow^*_{c,f} x^\infty$ exist in $roll\text{-}out(A)$.

The second path in $roll\text{-}out(A)$ corresponds to the path from l to *dc* in A. When l is a formal-in vertex and *dc* is an actual-in vertex then the transfer of l to *dc* by *TF* involves only the mapping of actual-in or call-site vertices to formal-in or entry vertices. Such mappings exists solely because of forward local slices. For example, transfer function TF_Q transfers (maps) formal-in vertex $Enter_R.a_{in}$ to formal-in vertex $Enter_Q.b_{in}$ iff $Call_R.a_{in} \in fl(Q, Enter_Q.b_{in})$. The composition of the *fl* slices used to compute the functions that transfer l to *dc* identifies a path of edges from *dc* to l. Because this transfer is along the call-sites in A that are expanded to create v^∞, occurrences of these edges in $roll\text{-}out(A)$ form the path $dc^\infty \rightarrow^*_{c,f} l^\infty$.

The third path in $roll\text{-}out(A)$ corresponds to the path from *dc* to u in A. A DAP-connected actual-in vertex, such as *dc*, is in $fl(A, u)$, which implies that a path from u to *dc* exists in A. Occurrences of the edges on this path connect an occurrence of u to *every* occurrence of *dc*, including dc^∞ because an *fl* slice "ascends" to calling procedures; thus, the procedure containing *dc* transitively calls the procedure containing u through a sequence of call-sites. In the roll-out of A, the expansion these call-

sites places an occurrence of u into a scope transitively enclosed in the scope containing each occurrence of dc Because occurrences of the edges that connect dc to u connect u^∞ to dc^∞, $roll\text{-}out(A)$ contains the path $u^\infty \rightarrow^*_{c,f} dc^\infty$.

Finally, concatenating the three paths $u^\infty \rightarrow^*_{c,f} dc^\infty$, $dc^\infty \rightarrow^*_{c,f} l^\infty$, and $l^\infty \rightarrow^*_{c,f} x^\infty$ produces the path $u^\infty \rightarrow^*_{c,f} x^\infty$. Since $u^\infty \in DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and the path $v^\infty \rightarrow^*_{c,f} x^\infty$ exists in $roll\text{-}out(A)$, this implies $v^\infty \in b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ as was to be shown.

(2) l is a formal-out vertex and dc is an actual-out vertex.

First, formal-out vertex l is in $Linkage(v)$ iff $v \in bl(P_v, l)$; similar to Case (1), the path from v to l implies that the path $v^\infty \rightarrow^*_{c,f} l^\infty$ exists in $roll\text{-}out(A)$. Second, the transfer of l to dc by TF is identical to Case (1) except that *bl* slices replace *fl* slices; thus, a path from l to dc exists in A and the path $l^\infty \rightarrow^*_{c,f} dc^\infty$ exist in $roll\text{-}out(A)$. Third, when dc is an actual-out vertex it is in $bl\,fl(A, u)$. In this case an argument similar to the one used for a DAP-connected actual-in vertex implies two paths exist in $roll\text{-}out(A)$: $u^\infty \rightarrow^*_{c,f} x^\infty$ and $dc^\infty \rightarrow^*_{c,f} x^\infty$. (Note that a *bl* slice, like an *fl* slice, "ascends" to calling procedures.)

Finally, concatenating the three paths $v^\infty \rightarrow^*_{c,f} l^\infty$, $l^\infty \rightarrow^*_{c,f} dc^\infty$, and $dc^\infty \rightarrow^*_{c,f} x^\infty$, produces the path $v^\infty \rightarrow^*_{c,f} x^\infty$ in $roll\text{-}out(A)$. Since $u^\infty \in DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and the path $u^\infty \rightarrow^*_{c,f} x^\infty$ exists in $roll\text{-}out(A)$, this implies that $v^\infty \in b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ as was to be shown.

(3) dc is an actual-in vertex and l is a formal-out vertex.

In this case one of the procedures called between P and P_v plays a special role in the argument that $v^\infty \in \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. Let Q be the procedure such that TF_Q transfers a formal-*out* vertex $Enter_R.b_{out}$ to a formal-*in* vertex $Enter_Q.a_{in}$. This transfer takes place because, in procedure Q, actual-out vertex $Call_R.b_{out}$ is in $bl\,fl(Q, Enter_Q.a_{in})$. Now suppose that $Call_R.b_{out}{}^\infty$ and $Enter_Q.a_{in}{}^\infty$ are occurrences of $Call_R.b_{out}$ and $Enter_Q.a_{in}$, respectively, from the same scope. Because $Call_R.b_{out} \in bl\,fl(Q, Enter_Q.a_{in})$, a vertex occurrence x^∞ exists such that $Call_R.b_{out}{}^\infty \rightarrow^*_{c,f} x^\infty$ and $Enter_Q.a_{in}{}^\infty \rightarrow^*_{c,f} x^\infty$.

The rest of the argument borrows parts from Cases (1) and (2). First, similar to Case (1) where there is a path $dc^\infty \rightarrow^*_{c,f} l^\infty$, we can show that there is a path $dc^\infty \rightarrow^*_{c,f} Enter_Q.a_{in}{}^\infty$. Second, similar to Case (2) where there is a path $l^\infty \rightarrow^*_{c,f} dc^\infty$, we can show that there is a path $l^\infty \rightarrow^*_{c,f} Call_R.b_{out}{}^\infty$. Combining these paths with $Enter_Q.a_{in}{}^\infty \rightarrow^*_{c,f} x^\infty$ and $Call_R.b_{out}{}^\infty \rightarrow^*_{c,f} x^\infty$, respectively, produces the paths $dc^\infty \rightarrow^*_{c,f} x^\infty$ and $l^\infty \rightarrow^*_{c,f} x^\infty$, respectively. Again borrowing from Cases (1) and (2), because (as in Case (1)) dc^∞ is an occurrence of a DAP-connected actual-in vertex, $roll\text{-}out(A)$ contains the path $u^\infty \rightarrow^*_{c,f} dc^\infty$ and because (as in Case (2)) $v \in bl(P_v, l)$, $roll\text{-}out(A)$ also contains the path $v^\infty \rightarrow^*_{c,f} l^\infty$. Combining these paths with $dc^\infty \rightarrow^*_{c,f} x^\infty$ and $l^\infty \rightarrow^*_{c,f} x^\infty$ produces the paths $u^\infty \rightarrow^*_{c,f} x^\infty$ and $v^\infty \rightarrow^*_{c,f} x^\infty$, respectively, which, since $u^\infty \in DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, imply that $v^\infty \in b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ as was to be shown.

□

Lemma 1

LEMMA 1. *If Base, A, B, and G_M pass the homogeneity test and no Type I interference exists in the integration of Base, A, and B then $G_{roll\text{-}out(M)} \subseteq G_{M^*}$.*

PROOF. The proof considers first the vertices and then the edges of $roll\text{-}out(M)$.

Vertices

Let v^∞ be an occurrence of vertex v in $roll\text{-}out(M)$. The Containment Lemma implies that $v^\infty \in V(roll\text{-}out(A))$ or $v^\infty \in V(roll\text{-}out(B))$; without loss of generality assume $v^\infty \in V(roll\text{-}out(A))$. If v^∞

is also a member of $roll\text{-}out(B)$ then v^∞ is in at least one of $Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$, $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, or $\Delta^\infty(roll\text{-}out(B), roll\text{-}out(Base))$, and hence in $G_M{}^-$. Otherwise, v^∞ is absent from $roll\text{-}out(B)$ because either v is absent from B, or a call-site $Call_P$ is absent from B where $Call_P$ is a call-site from Seq—the sequence of call-sites in M whose expansion leads to v^∞. We consider separately these two possibilities for v^∞ being absent from $roll\text{-}out(B)$:

(1) Vertex v is absent from B.

Because $v \notin V(B)$ but $v \in V(G_M)$ and $Base$, A, B, and G_M pass the homogeneity test, v must be in $bl\,fl\,DAP^S(A, Base)$. Thus, as shown in the Every Occurrence Lemma of Section 5.5.2, all occurrences of v in $roll\text{-}out(A)$, including v^∞, are in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, which implies v^∞ is in $G_M{}^-$.

(2) Call-site $Call_P$ is absent from B.

Because $Call_P$ is in $V(G_M)$ but not $V(B)$, v is in a procedure transitively called from $Call_P$, and $Base$, A, B, and G_M pass the homogeneity test, it must be the case that either

(a) $v \in bl\,fl\,DAP^S(A, Base)$, or

(b) v is summarized by an initial summary set $Linkage(v)$ that is either not transferred to SN_P, or if it is transferred to $S \in SN_P$ then $DAPConnected(Map(Call_P, \{S\}))$ holds.

In Case (a), all occurrences of v in $roll\text{-}out(A)$, including v^∞, are in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. In Case (b), there is some call-site of Seq at which $Linkage(v)$ has been transferred (and then mapped) to a DAP-connected set of actual-in, call-site and actual-out vertices; thus, the Slice-Need Lemma implies that v^∞ is in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. Finally, since $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ is contained in $G_M{}^-$, together these cases imply v^∞ is in $G_M{}^-$.

Edges

Let $e^\infty = x^\infty \longrightarrow y^\infty$ be any non-def-order edge from $roll\text{-}out(M)$; the proof for a def-order edge $x^\infty \longrightarrow_{do(y^-)} z^\infty$ is the same except z^∞ is included with x^∞ in the following argument. The Containment Lemma implies that e^∞ is in $roll\text{-}out(A)$ or $roll\text{-}out(B)$. If e^∞ is in both $roll\text{-}out(A)$ and $roll\text{-}out(B)$ then it is in M^∞. Otherwise, without loss of generality assume e^∞ is in $roll\text{-}out(A)$ but not $roll\text{-}out(B)$; there are two possible ways e^∞ can be absent from $roll\text{-}out(B)$: either (1) e is absent from B, or (2) a call-site $Call_P$ from Seq is absent from B. We consider separately these two possibilities:

(1) Edge e is absent from B.

Recall that $e = x \longrightarrow y$. There are two sub-cases to consider:

(i) $y \notin V(B)$.

The vertex argument implies that $y^\infty \in M^\infty$, because y^∞ is in $roll\text{-}out(M)$. Furthermore, y^∞ must be in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ because it is in M^∞ but not $roll\text{-}out(B)$ (by assumption $y \notin V(B)$). Computing Δ^∞ involves a b^∞ slice; therefore, since $e^\infty \in E(G_{roll\text{-}out(A)})$, x^∞ is also included in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. The inclusion of these vertices in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and the existence of e^∞ in $roll\text{-}out(A)$ imply that $e^\infty \in Induce\ \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that e^∞ is in $G_M{}^-$.

(ii) $y \in V(B)$.

We consider separately the two possibilities for e's membership in $Base$. First, if e is absent from $Base$ then $e^\infty \notin E(roll\text{-}out(Base))$. Because $e^\infty \in E(roll\text{-}out(A))$, this implies y^∞ is in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and thus in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. As in Case (1).(i), the b^∞ slice used to compute Δ^∞ includes x^∞ in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, which causes $e^\infty = x^\infty \longrightarrow y^\infty$ to be included in $Induce\ \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. This implies

that e^{∞} is in G_{M^-}.

Second, if e is present in *Base* then, because e is not in B but y is, $y \in DAP^S(B, Base) \subseteq \Delta^S(B, Base)$. However, $e \in E(G_M)$ and $e \notin E(B)$ imply that $y \in DAP^S(G_M, B)$; therefore, Type I interference exists because $y \in \Delta^S(B, Base) \cap DAP^S(G_M, B)$. This contradicts the assumption that no Type I interference exists.

(2) Call-site $Call_P$ is absent from B.

The vertex argument implies that, since x^{∞} and y^{∞} are in *roll-out*(M), they are in M^{∞}. Because $Call_P$'s absence from B implies that x^{∞} and y^{∞} are not in *roll-out*(B), they must be in M^{∞} because they are in Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$). Therefore, since $e^{\infty} = x^{\infty} \longrightarrow y^{\infty}$, e^{∞} is in *Induce* Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$), which implies that e^{∞} is in G_{M^-}.

□

6.2.2. Lemma 2

Lemma 1 represents half of the mutual-containment argument used to prove the Sufficiency Theorem. Lemma 2 proves the other half (*i.e.* that whenever no Type I or Type II interference exists, $G_{M^-} \subseteq G_{roll\text{-}out(M)}$). Together Lemmas 1 and 2 imply that if *Base*, A, B, and G_M pass the homogeneity test and no Type I or Type II interference exists then $G_{roll\text{-}out(M)} \approx G_{M^-}$ (*i.e.*, $G_{roll\text{-}out(M)}$ is isomorphic to G_{M^-}), which, since *roll-out*(M) is homogeneous, implies that M^{∞} is homogeneous.

The proof of Lemma 2 relies on the Δ Inclusion Lemma, which follows from four supporting results: the Delta Slice Lemma, the Valid Path Lemma, the Dual Path Lemma, and the Directly Affected Points Lemma. After proving these four lemmas and the Δ Inclusion Lemma, we then prove Lemma 2 and finally the Sufficiency Theorem. Before proving these four lemmas it is necessary to formalize the relationship between paths of edges in $G_{roll\text{-}out(A)}$ and paths of edges in G_A: every path in $G_{roll\text{-}out(A)}$ corresponds to a path in G_A; however, the converse (that every path in G_A corresponds to a path in $G_{roll\text{-}out(A)}$) is not true. The paths in G_A that correspond to paths in $G_{roll\text{-}out(A)}$ are *valid paths*, which are formalize below before proving the first lemma.

Valid Paths

A valid path is a sub-path of an *execution path*. Execution paths are composed of control, flow, meeting-point, and interprocedural edges such that the interprocedural edges correspond to a legal sequence of procedure calls and returns that begins in a procedure P and ends in the same activation of procedure P. By a "legal sequence of procedure calls and returns" we mean that an execution path does not contain edges that represent a procedure being called from one call-site but returning to a different call-site. Execution paths (and subsequently valid paths) are formalized below.

DEFINITION. (Execution Paths). Vertices x and y from system A are connected by a *execution path* iff

1) x and y are connected by a path of non-def-order and non-summary edges[3] in A, and
2) the interprocedural edges on the path are constrained by the following grammar (the second *Path* production is indexed by call-site):

[3] If x and y are connected by a path that contains summary edges then x and y are also connected by a path of non-summary edges. This path is obtained by (repeated) in-line substitution of the edges that induced the summary edges.

$Path \rightarrow Path\ Path$
$Path \rightarrow in_c\ Path\ out_c$
$\rightarrow \varepsilon$

where "in_c" represents either a parameter-in or call edge associated with call-site c, and "out_c" represents a parameter-out edge associated with call-site c.

Intuitively, because the language constraining the interprocedural edges of an execution path corresponds to the language of balanced parentheses, it represents the balanced execution of procedure calls and returns made during a system's execution.

A *valid path* is any sub-path of an execution path. Thus, valid paths also rule-out paths that enter a called procedure from one call-site and then exit the procedure to a different call-site. Intuitively, a valid path from x to y means that the computation represented by x has a potential influence on the computation represented by y. Formally, there is a valid path from x to y, denoted by $x \longrightarrow^{vp} y$, if the conditions of the following definition are satisfied.

DEFINITION. (Valid Paths). Vertices x and y from system A are connected by a *valid path* iff

1) x and y are connected by a path of non-def-order and non-summary edges in A, and
2) the interprocedural edges on the path are constrained by following grammar (the *Head* and *Tail* productions are indexed by call-site):

$Valid\text{-}Path \rightarrow Head\ Tail$
$Head \rightarrow out_{c1}\ Head$
$\rightarrow Path\ Head$
$\rightarrow \varepsilon$
$Tail \rightarrow in_{c2}\ Tail$
$\rightarrow Path\ Tail$
$\rightarrow \varepsilon$

where out_{c1} and in_{c2} represent *unmatched* interprocedural edges (out_{c1} is unmatched if no corresponding *in* edge in_{c1} occurs *before* out_{c1} on the path and in_{c2} is unmatched if no corresponding *out* edge out_{c2} occurs *after* in_{c2} on the path.)

For brevity, we use the phrase "the path is described by" in place of "the interprocedural edges on the path satisfy the constraints described by" and say that "a path p is in $\mathcal{L}(NonTerminal)$" (or equivalently $p \in \mathcal{L}(NonTerminal)$) if p is described by *NonTerminal* ($\mathcal{L}(NonTerminal)$ denotes the language generated by *NonTerminal*). For example, $x \longrightarrow^{*} y \in \mathcal{L}(Valid\text{-}Path)$ means $x \longrightarrow^{vp} y$.

Result 1: The Delta Slice Lemma

The Delta Slice Lemma, states that if valid paths beginning with vertices u and v share certain unmatched edges and meet at a vertex x then v is in the "delta-slice" with respect to u (the delta-slice is the slice used to compute Δ^S; thus, if u and v are in system A then $v \in b2\,b1\,f1\,(A, u) \cup b2\,f2\,f1\,(A, u)$). The proof follows from two other results: the Path Lemma, which demonstrates that summary edges correspond to paths in $\mathcal{L}(Path)$, and the Meeting-Point Lemma, which states that the first shared procedure (*i.e.*, the procedure that contains the head of the first shared unmatched edge on the paths from u and v) contains a meeting point vertex.

The Path Lemma

LEMMA. (PATH LEMMA). *If $x \longrightarrow^{*} y \in \mathcal{L}(Path)$ then a path composed of only control, flow, summary, and meeting-point edges connects x to y.*

PROOF. The proof is by induction over the grammar for (execution) paths. First, paths corresponding to the production $Path \rightarrow \varepsilon$ contain no interprocedural edges and therefore are composed solely of control, flow, and meeting-point edges. Second, for the right-hand side of each indexed production $Path \rightarrow in_c\ Path\ out_c$, the inductive hypothesis implies there is a path of control, flow, and summary edges through the procedure called by c. This path satisfies the conditions for a summary edge at c. Thus, replacing the edges $in_c, \cdots, out_c$ on the path from x to y with this summary edge leaves a path from x to y composed of control, flow, summary, and meeting-point edges. Finally, for $Path \rightarrow Path\ Path$, the path from x to y can be broken at an intermediate point z such that the paths from x to z and z to y correspond to the two right-hand-side occurrences of *Path*. The inductive hypothesis implies that paths composed of control, flow, summary, and meeting-point edges connect x to z and z to y; hence, the concatenation of these paths produces a path of control, flow, summary, and meeting-point edges from x to y. □

The Meeting-Point Lemma

The Meeting-Point Lemma, which considers two paths in $\mathcal{L}(Tail)$ that share a sequence of unmatched *in*-edges, describes a sufficient condition for the existence of a meeting-point vertex and edges (because the paths are in $\mathcal{L}(Tail)$, the only unmatched edges are unmatched *in*-edges). The proof is an induction over the number of unmatched *in*-edges on a path, which is formalized as the *distance* of the path.

DEFINITION. (Distance). The *distance* between the ends of path $v \longrightarrow^* x$, denoted by $dist(v \longrightarrow^* x)$, is the number of unmatched *in*-edges on the path from v to x.

The Meeting-Point Lemma is slightly more general than described above because it considers the possibility that $dist(v \longrightarrow^* x) = 0$, in which case the "meeting point" need not be a meeting-point vertex. This special case is included in the general case by stating that $v \in bl\,fl(P_v, u)$, which implies there is a vertex x, which may or may not be a meeting-point vertex, such that paths of intraprocedural edges connect v and u to x (*i.e.* $v \longrightarrow^*_{c,f,s,mp} x$ and $u \longrightarrow^*_{c,f,s,mp} x$).

LEMMA (MEETING-POINT LEMMA). *In system A, if* $v \longrightarrow^* x \in \mathcal{L}(Tail)$, $u \longrightarrow^* x \in \mathcal{L}(Tail)$, *and* $v \longrightarrow^* x$ *and* $u \longrightarrow^* x$ *have the same unmatched in-edges then* $v \in bl\,fl(A, u)$.

PROOF. Because $v \longrightarrow^* x$ and $u \longrightarrow^* x$ have the same unmatched *in*-edges $dist(v \longrightarrow^* x)$ must equal $dist(u \longrightarrow^* x)$; the proof is an induction over $dist(v \longrightarrow^* x)$.

Base Case: (the lemma holds when $dist(v \longrightarrow^* x) = 0$).

The Path Lemma implies that paths of intraprocedural edges (including summary edges) connect v and u to x; thus, $v \in bl\,fl(A, u)$ since these paths imply $v \in bl(A, x)$ and $x \in fl(A, u)$, respectively.

Inductive Step: (if the lemma holds for all z such that $dist(v \longrightarrow^* z) = k$ then it holds for an x such that $dist(v \longrightarrow^* x) = k+1$).

Assume $dist(v \longrightarrow^* x) = k+1$. The Path Lemma implies that paths of (only) intraprocedural edges (including summary edges) and (unmatched) *in*-edges connect v and u to x. Since $v \longrightarrow^* x$ and $u \longrightarrow^* x$ have the same unmatched *in*-edges they both begin in the same procedure P (with edges from P's program dependence graph) before exiting P at the same call-site through (possibly different) actual-in vertices. Dividing these paths at these actual-in vertices (*i.e.*, at their first unmatched *in*-edges) yields the following (recall that $\longrightarrow^*_{c,f,s}$ denotes a path of control, flow, and summary edges that are all in the same program dependence graph):

Subpath in P	First *in* edge	Remainder of path
$v \rightarrow^*_{c,f,s} Call_P.v_{in}$	$\rightarrow_{in,\,call}$	$Enter_P.v_{in} \rightarrow^* x$
$u \rightarrow^*_{c,f,s} Call_P.u_{in}$	$\rightarrow_{in,\,call}$	$Enter_P.u_{in} \rightarrow^* x$

Because $Enter_P.v_{in} \rightarrow^* x$ and $Enter_P.u_{in} \rightarrow^* x$ are in $\mathcal{L}(Tail)$, have the same unmatched *in*-edges, and $dist(Enter_P.v_{in} \rightarrow^* x) = dist(Enter_P.u_{in} \rightarrow^* x) = k$ the inductive hypothesis implies $Enter_P.v_{in} \in bl\,fl(A, Enter_P.u_{in})$. Thus, by definition, there is a meeting-point vertex and meeting-point edges at every call-site on P; in particular, at $Call_P$ there is a meeting-point vertex m and meeting-point edges $Call_P.v_{in} \rightarrow_{mp} m$ and $Call_P.u_{in} \rightarrow_{mp} m$. Finally, the paths obtained by extending $v \rightarrow^*_{c,f,s} Call_P.v_{in}$ and $u \rightarrow^*_{c,f,s} Call_P.u_{in}$ with these meeting-point edges imply $m \in fl(A, u)$ and $v \in bl(A, m)$; thus, $v \in bl\,fl(A, u)$.

□

The Delta Slice Lemma

Given two valid paths that begin with vertices v and u, the premise of the Delta Slice Lemma provides a sufficient condition for v to be included in the "delta-slice", taken with respect to u (*i.e.* $v \in b2\,f2\,fl(A, u) \cup b2\,bl\,fl(A, u)$). Later in this section, we show (in the Dual Path Lemma) that if paths from occurrences of u and v reach an occurrence of vertex x then the premise of the Delta Slice Lemma is satisfied. Combined with the Directly Affected Points Lemma, which (roughly) states that if an occurrence of u is a directly affected point then so is u, this provides a condition sufficient to show that v is in $\Delta^S(A, Base)$ because of directly affected point u. The Delta Slice Lemma has two cases which are illustrated in Figure 6.4.

LEMMA (DELTA SLICE LEMMA). *In system A, assume there are two valid paths $v \rightarrow^* t \rightarrow^* y \rightarrow^* x \in \mathcal{L}(Valid\text{-}Path)$ and $u \rightarrow^* z \rightarrow^* w \rightarrow^* x \in \mathcal{L}(Valid\text{-}Path)$ such that, for the first path, $v \rightarrow^* t \in \mathcal{L}(Head)$ and $t \rightarrow^* y \rightarrow^* x \in \mathcal{L}(Tail)$, and, for the second path, $u \rightarrow^* z \in \mathcal{L}(Head)$ and*

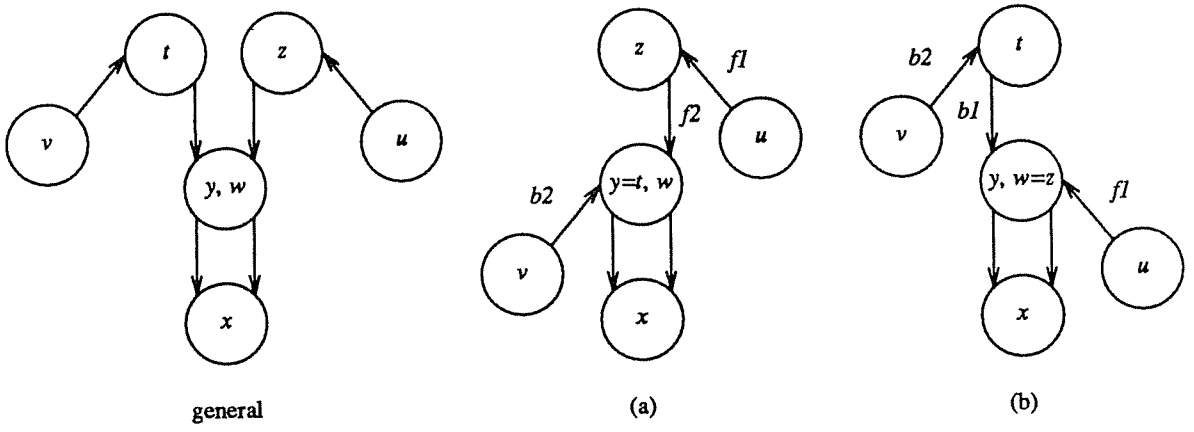


Figure 6.4. Illustrations for the Delta Slice Lemma. Circles represent procedures (the labels in the circles refer to the vertices mentioned in the Delta Slice Lemma) and lines represent transitive procedure calls. The first figure shows the conditions stated in the lemma without the restrictions of assertions (a) or (b); the second and third figures show the restrictions for (a) (*i.e.* $t = y$) and (b) (*i.e.* $z = w$), respectively. Line labels in Figures (a) and (b) indicate which slicing operator of Δ^S is used to "traverse" the edges on various parts of the paths $v \rightarrow^{vp} x$ and $u \rightarrow^{vp} x$.

$z \longrightarrow^* w \longrightarrow^* x \in \mathcal{L}(Tail)$. *If* $y \longrightarrow^* x$ *and* $w \longrightarrow^* x$ *have the same unmatched in-edges then both of the following hold:*

(a) if $y = t$ *then* $v \in b2\,f2\,f1(A, u)$, *and*

(b) if $w = z$ *then* $v \in b2\,b1\,f1(A, u)$.

PROOF. We first argue that the Meeting-Point Lemma implies $y \in bl\,fl(A, w)$, and then use the Path Lemma to prove a relationship between paths in $\mathcal{L}(Head)$ and $\mathcal{L}(Tail)$ and the slicing operators $f1, f2, b1$, and $b2$. These results are then combined to prove assertions (a) and (b) separately. First, the Meeting-Point Lemma implies that $y \in bl\,fl(A, w)$ because $t \longrightarrow^* y \longrightarrow^* x \in \mathcal{L}(Tail)$ implies that $y \longrightarrow^* x \in \mathcal{L}(Tail)$, $z \longrightarrow^* w \longrightarrow^* x \in \mathcal{L}(Tail)$ implies that $w \longrightarrow^* x \in \mathcal{L}(Tail)$, and, by assumption, $y \longrightarrow^* x$ and $w \longrightarrow^* x$ have the same unmatched *in*-edges.

We now turn to the relationship between paths in $\mathcal{L}(Head)$ and $\mathcal{L}(Tail)$ and the slicing operators $f1, f2$, $b1$, and $b2$. First, the Path Lemma implies that, for the path $v \longrightarrow^* t \in \mathcal{L}(Head)$, a path of intraprocedural edges (including summary edges) and (unmatched) *out*-edges connects v to t; because $f1$ and $b2$ slices traverse exactly these edges, $v \in b2(A, t)$ and $t \in f1(A, v)$. Similarly, for a path $u \longrightarrow^* z \in \mathcal{L}(Tail)$, the Path Lemma implies a path of intraprocedural edges and (unmatched) *in*-edges connect u to z; because $f2$ and $b1$ slices traverse exactly these edges, $u \in b1(A, z)$ and $z \in f2(A, u)$.

For the various subpaths of $v \longrightarrow^* t \longrightarrow^* y \longrightarrow^* x$ and $u \longrightarrow^* z \longrightarrow^* w \longrightarrow^* x$ this relationship with the slicing operators $f1, f2, b1$, and $b2$ is summarized in the following table. The table has two columns, which take into account the premises of Assertions (a) and (b), respectively, of the Lemma: column one summarizes these relationships for Assertion (a) where $y = t$ and column two summarizes them for Assertion (b) where $w = z$. (The slicing operators shown in the table are also marked in Figure 6.4.)

Assertion (a)		Assertion (b)	
Path	Slice	Path	Slice
$v \longrightarrow^* y \in \mathcal{L}(Head)$	$v \in b2(A, y)$	$v \longrightarrow^* t \in \mathcal{L}(Head)$	$v \in b2(A, t)$
$z \longrightarrow^* w \in \mathcal{L}(Tail)$	$w \in f2(A, z)$	$t \longrightarrow^* y \in \mathcal{L}(Head)$	$t \in b1(A, y)$
$u \longrightarrow^* z \in \mathcal{L}(Head)$	$z \in f1(A, u)$	$u \longrightarrow^* w \in \mathcal{L}(Tail)$	$w \in f1(A, u)$

(Recall that the Meeting-Point Lemma implies $y \in bl\,fl\,(A, w)$.) Composing the slices for Assertion (a) with $y \in bl\,fl(A, w)$ yields $v \in b2\,bl\,fl\,f2\,f1(A, u)$ and composing the slices for Assertion (b) with $y \in bl\,fl(A, w)$ yields $v \in b2\,b1\,bl\,fl\,f1(A, u)$. Because $b1 \circ bl = b1$, $b2 \circ bl = b2$, $fl \circ f2 = f2$, and $fl \circ f1 = f1$, these simplify to $v \in b2\,f2\,f1(A, u)$ and $v \in b2\,b1\,f1(A, u)$, respectively. □

Result 2: The Valid Path Lemma

The second result used in the proof of the Δ Inclusion Lemma is the Valid Path Lemma, which states that all the paths in *roll-out*(A) correspond to valid paths in A. The Lemma provides the link between the paths in *roll-out*(A) that cause *Integrate*$^\infty$ to include an occurrence of vertex v in Δ^∞(*roll-out*(A), *roll-out*(*Base*)) and the paths in A that cause *Integrate*^{S} to include v in $\Delta^S(A, Base)$.

LEMMA (VALID PATH LEMMA). *If* $x^\infty \longrightarrow^*_{c,f} y^\infty$ *in roll-out*(A) *then* $x \longrightarrow^{vp} y$ *in* A.

PROOF. To show that $x \longrightarrow^{vp} y$, we first show that a path of edges connects x and y, and then argue that this path is in $\mathcal{L}(Valid\text{-}Path)$.

A path of edges connects x and y because of the correspondence between edge occurrences in *roll-out*(A) and edges in A. Under this correspondence, every edge occurrence e^∞ in *roll-out*(A) identifies an edge e in A. In addition, if one of e^∞'s endpoints is a transfer-in, scope, or transfer-out vertex then e^∞

identifies—in addition to e—an interprocedural edge and if both of e^∞'s endpoints are transfer-in, scope, or transfer-out vertices then e^∞ identifies two interprocedural edges. For example, the edge $v^\infty \longrightarrow u^\infty$ where u^∞ is a transfer-in vertex identifies two edges: one from v to the actual-in vertex of which u^∞ is an occurrence and the other from this actual-in vertex to the formal-in vertex of which u^∞ is also an occurrence. By concatenating the edges in A identified by the edge occurrences in *roll-out*(A) that make up $x^\infty \longrightarrow^* y^\infty$, we obtain a path $x \longrightarrow^* y$.

To show that this path is a valid path, we must identify the *in* and *out* edges on the path and then demonstrate that these edges satisfy the conditions for begin a valid path. This can be accomplished by considering the string of scopes entered and exited by the path from x^∞ to y^∞ (hereafter referred to as the *scope string*). Let S_i denote a scope nested within i other scopes. The scope string corresponding to the path $x^\infty \longrightarrow^*_{c,f} y^\infty$ is the sequence of scopes entered into by the path from x^∞ to y^∞. For example, if the path enters enclosing scope S_{i-1} from subordinate scope S_i then S_i is followed by S_{i-1}. A scope string is used to determine the *in* and *out* edges on the path from x to y as follows: the sequence $S_j\, S_{j+1}$ corresponds to an *in* edge and the sequence $S_j\, S_{j-1}$ corresponds to an *out* edge.

To show that the path from x to y is in $\mathcal{L}(Valid\text{–}Path)$, we first consider a *balanced* scope string: a scope string of the form $S_j, S_{j+1}, \cdots, S_{j+n}, S_{j+n+1}, S_{j+n}, \cdots, S_{j+1}, S_j$. In *roll-out*$(A)$ each scope is nested within a single enclosing scope; therefore, the first and last interprocedural edges of a balanced scope string (*i.e.* those determined by S_j, S_{j+1} and S_{j+1}, S_j, respectively) must be associated with the same call-site. Because this is also true for the remaining edges determined by $S_{j+1}, \cdots, S_{j+1}$, the path corresponding to a balanced scope string is an execution path and hence in $\mathcal{L}(Path)$.

In a general scope string, all substrings of the form $S_j, S_{j+1}, \cdots, S_{j+1}, S_j$ correspond to balanced strings of *in* and *out* edges in $\mathcal{L}(Path)$. If we remove all such substrings, the remaining "skeletal" scope string is in one of the following forms:

(1) The empty string (*i.e.* the original string was balanced),
(2) $S_j, S_{j-1}, \cdots, S_{j-n}$,
(3) $S_j, S_{j+1}, \cdots, S_{j+n}$, or
(4) $S_j, S_{j-1}, \cdots, S_{j-n}, S'_{j-n+1}, \cdots, S'_{j-n+i}$.

To see that these represent all possible skeletal scope strings, consider building an arbitrary scope string by appending a new scope to the end of an existing scope string, beginning with the empty string. In each case for a string of length at least 1, the appended scope must either represent a subordinate scope or an enclosing scope; thus, its subscript may differ from its predecessor by either +1 or −1. Before considering general skeletal scope strings of Form 2, 3, and 4 that have length ≥ 2, we consider, as special cases, skeletal scope strings of length < 2.

First, the empty string is of Form 1. Now consider the length-one scope string S_j, which is of Forms 2, 3, and 4. Appending S_{j+1} to S_j produces a string of Form 3; appending S_{j-1} to S_j produces a string of Form 2.

We now consider appending a new scope to an arbitrary scope string of Forms 2, 3, and 4 that has length ≥ 2. For Form 2, appending S_{j-n-1} produces a skeletal scope string of Form 2, while appending S'_{j-n+1} produces a skeletal scope string of Form 4. Beginning with Form 3 and appending S_{j+n+1} produces a skeletal scope string of Form 3; appending S_{j+n-1} implies that the scope string ends with $S_{j+n-1}, S_{j+n}, S_{j+n-1}$, which is a balanced substring and therefore in $\mathcal{L}(Path)$. Such strings are removed because they are not part of a skeletal scope string; the remaining skeletal scope string, $S_j, S_{j+1}, \cdots, S_{j+n-1}$, is of Form 3 unless its length is zero, in which case it is of Form 1. Finally, for Form 4, appending $S'_{j-n+i+1}$ produces a skeletal scope string of Form 4, while appending $S'_{j-n+i-1}$ has the same affect as appending S_{j+n-1} to a string of Form 3. Therefore, Forms 1, 2, 3, and 4 represent all possible skeletal scope strings.

We now prove that the paths corresponding to skeletal scope strings are valid paths and then extend this argument to general scope strings. For Form 1, the empty string is trivially in $\mathcal{L}(Valid\text{-}Path)$. For Form 2, the scope string $S_j, S_{j-1}, \cdots, S_{j-n}$ identifies a sequence of *unmatched in*-edges in $\mathcal{L}(Tail)$ and therefore in $\mathcal{L}(Valid\text{-}Path)$. Similarly, a scope string $S_j, S_{j+1}, \cdots, S_{j+n}$ of Form 3 identifies a sequence of *unmatched out*-edges in $\mathcal{L}(Head)$ and therefore in $\mathcal{L}(Valid\text{-}Path)$. And finally, a skeletal scope string of Form 4 can be broken into $S_j, S_{j-1}, \cdots, S_{j-n}$ and $S'_{j-n+1}, \cdots, S'_{j-n+i}$ which identify sequences in $\mathcal{L}(Head)$ and $\mathcal{L}(Tail)$, respectively; thus, their concatenation is in $\mathcal{L}(Valid\text{--}Path)$.

To complete the proof, we show that replacing the substrings removed from a general scope string to obtain a skeletal scope string does not affect the acceptance of a path in $\mathcal{L}(valid\text{-}Path)$. The key observation is that a string in $\mathcal{L}(Valid\text{-}Path)$ results whenever the non-terminal *Path* is added between any two symbols of a string in $\mathcal{L}(Valid\text{-}Path)$. Thus, having sequences of balanced scope strings interspersed in a skeletal scope string does not affect the argument that the corresponding path is a valid path. Therefore, x and y are connected by a valid path. □

Result 3: The Dual Path Lemma

The Dual Path Lemma puts together the results of the Delta Slice Lemma and the Valid Path Lemma to show that if two paths $v^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$ and $u^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$ exist in *roll-out*(A) then v is in the delta-slice taken with respect to u (*i.e.* $v \in b2f2f1\ (A, u) \cup b2\,b1f1\ (A, u)$). In the proof, the vertex names are the same as in the proof of the Delta Slice Lemma and Figure 6.4.

LEMMA (DUAL PATH LEMMA). *If* $v^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$ *and* $u^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$ *in roll-out*(A) *then* $v \in b2f2f1\ (A, u) \cup b2\,b1f1\ (A, u)$.

PROOF. The Valid Path Lemma implies $v \rightarrow^{vp} x$ and $u \rightarrow^{vp} x$. The path $v \rightarrow^{vp} x$ can be divided at a vertex t such that $v \rightarrow^{*} t \in \mathcal{L}(Head)$ and $t \rightarrow^{*} x \in \mathcal{L}(Tail)$ and the path $u \rightarrow^{vp} x$ can be divided at a vertex z such that $u \rightarrow^{*} z \in \mathcal{L}(Head)$ and $z \rightarrow^{*} x \in \mathcal{L}(Tail)$. The first half of the proof shows either (1) that P_z (transitively) calls P_t and the path from u to x goes through P_t (shown in Figure 6.4a) or (2) that P_t (transitively) calls P_z and the path from v to x goes through P_z (shown in Figure 6.4b); the second half of the proof shows that these paths satisfy the premise of the Delta Slice Lemma. The result then follows from the Delta Slice Lemma.

The proof of the first half begins by showing that either P_z (transitively) calls P_t or P_t (transitively) calls P_z. This can be done by contradiction: P_z and P_t both call P_x, but if neither transitively calls the other then the expansion of the call-sites by which P_z calls P_x and those by which P_t calls P_x include separate copies of P_x in the scopes resulting from the expansion of calls in P_z and P_t. Thus, the paths from v^{∞} and u^{∞} reach different occurrences of x, which contradicts the assumption that $v^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$ and $u^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$; therefore, either P_t calls P_z or P_z calls P_t.

To complete the first half of the proof, we show (again by contradiction) that when P_z calls P_t, the path from u to x goes through P_t (the proof that when P_t calls P_z, the path from v to x goes through P_z is virtually identical). The contradiction used to prove this step is similar to the one above: if the path from u and x does not go through P_t then some procedure, P, called between P_z and P_t transitively calls both P_t and P_x from different call-sites. The expansion of these call-sites includes separate copies of P_x in the scopes produced form P, which means the paths from v^{∞} and u^{∞} reach different occurrences of x—a contradiction.

Given that either P_z calls P_t and the path from u to x goes through P_t or P_t calls P_z and the path from v to x goes through P_z, the second half of the proof is to show that $v \rightarrow^{vp} x$ and $u \rightarrow^{vp} x$ satisfy the premise of part (a) or (b) of the Delta Slice Lemma. In the P_z calls P_t case, there is a vertex w in P_t that divides $z \rightarrow^{*} x$ such that $z \rightarrow^{*} w \rightarrow^{*} x \in \mathcal{L}(Tail)$ and, by assumption, $u \rightarrow^{*} z$ and $v \rightarrow^{*} t$ are in $\mathcal{L}(Head)$ and $y \rightarrow^{*} x$ is in $\mathcal{L}(Tail)$. If we let $y = t$ then $y \rightarrow^{*} x$ becomes $t \rightarrow^{*} y \rightarrow^{*} x$. Finally, because $v^{\infty} \rightarrow^{*}_{c,f} x^{\infty}$

and $u^\infty \rightarrow^*_{c,f} x^\infty$ reach the same occurrence of x, $y \rightarrow^* x$ and $w \rightarrow^* x$ must have the same unmatched *in*-edges; thus, part (a) of the Delta Slice Lemma implies that $v \in b2f2f1(A, u) \subseteq b2f2f1(A, u) \cup b2b1f1(A, u)$.

In the P_t calls P_z case the proof is similar. First, there is a vertex y in P_z such that $t \rightarrow^* y \rightarrow^* x \in \mathcal{L}(Tail)$ and, by assumption, $u \rightarrow^* z$ and $v \rightarrow^* t$ are in $\mathcal{L}(Head)$ and $w \rightarrow^* x$ is in $\mathcal{L}(Tail)$. If we let $w = z$ then $w \rightarrow^* x$ becomes $z \rightarrow^* w \rightarrow^* x$. Finally, because $v^\infty \rightarrow^*_{c,f} x^\infty$ and $u^\infty \rightarrow^*_{c,f} x^\infty$ reach the same occurrences of x, $y \rightarrow^* x$ and $w \rightarrow^* x$ must have the same unmatched *in*-edges; thus, part (b) of the Delta Slice Lemma implies that $v \in b2b2b1(A, u) \subseteq b2f2f1(A, u) \cup b2b1f1(A, u)$. □

Result 4: Directly Affected Points Lemma

The fourth and final result used to prove the Δ Inclusion Lemma is the Directly Affected Points Lemma. Recall that the goal of the Δ Inclusion Lemma is to prove that if vertex occurrence v^∞ is in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ then v is in $\Delta^S(A, Base)$. The sets $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $\Delta^S(A, Base)$ are constructed from the sets $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $DAP^S(A, Base)$, respectively. Ideally, we would like to prove that if vertex occurrence u^∞ is in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ then u is in $DAP^S(A, Base)$. The Δ Inclusion Lemma would then follow immediately from the Dual Path Lemma. Unfortunately, $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $DAP^S(A, Base)$ are not related in this way.

The Directly Affected Points Lemma proves that a weaker relationship holds between $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $DAP^S(A, Base)$, which can be explained as follows: Let U^∞ be the set of occurrences from $roll\text{-}out(A)$ of vertices in $DAP^S(A, Base)$. The lemma first shows that U^∞ is a subset of $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. However, the containment in the other direction does not hold (*i.e.* $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base)) \not\subseteq U^\infty$). The lemma does show that $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ is a subset of the vertices in *the forward slice* of $roll\text{-}out(A)$ taken with respect to U^∞. This weaker form of containment relationship is sufficient to prove that $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $b^\infty f^\infty(roll\text{-}out(A), U^\infty)$ contain the same vertex occurrences, which is sufficient to prove the Δ inclusion Lemma.

LEMMA (THE DIRECTLY AFFECTED POINTS LEMMA).
Let $U^\infty = \{ u^\infty \in occurrences(roll\text{-}out(A), u) \mid u \in DAP^S(A, Base) \}$.
(a) $U^\infty \subseteq DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$,
(b) $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base) \subseteq f^\infty(roll\text{-}out(A), U^\infty)$, *and*
(c) $b^\infty f^\infty(roll\text{-}out(A), U^\infty) = \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

PROOF.

(a) By definition, $u \in DAP^S(A, Base)$ iff it is in A but not in $Base$ or it has different incoming edges in A and $Base$ (recall that a def-order edge is viewed as a hyper-edge that is an incoming edge of the witness vertex). If u is in A but not in $Base$ then no occurrences of u exist in $roll\text{-}out(Base)$; therefore, all occurrences of u in U^∞ are in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

Now suppose that, u is in A and $Base$, but has different incoming edges. Because parameter-in, call, and parameter-out edges are ignored when computing $DAP^S(A, Base)$, we need only consider control, flow, and def-order edges. For these kinds of edges, a vertex occurrence's incoming edge occurrences in $roll\text{-}out(A)$ are occurrences of the vertex's incoming edges in A. Thus, determining if u is a directly affected point requires considering only the incoming edges for u. This local information is not affected by the expansion of call-sites, so, if $u \in DAP^S(A, Base)$ then every occurrence of u has different incoming edge occurrences in $roll\text{-}out(A)$ and $roll\text{-}out(Base)$, and is therefore, in $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$. This implies that $U^\infty \subseteq DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$.

(b) Besides occurrences of vertices from $DAP^S(A, Base)$ (*i.e.*, besides U^∞), $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ includes occurrences of vertices that are in $roll\text{-}out(A)$ but not $roll\text{-}out(Base)$ because of the expansion of call-sites from A that are not in $Base$. While these vertex occurrences are not necessarily occurrences of vertices in $DAP^S(A, Base)$, the call-site vertices of the call-sites in A that are not in $Base$ are in $DAP^S(A, Base)$. Since the vertex occurrences in $roll\text{-}out(A)$ because of the expansion of these call-sites are in the f^∞ slice of $roll\text{-}out(A)$ taken with respect to occurrences of these call-site vertices, they are in the f^∞ slice of $roll\text{-}out(A)$ taken with respect U^∞. Thus, $DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base) \subseteq f^\infty(roll\text{-}out(A), U^\infty)$.

(c) The proof that $b^\infty f^\infty(roll\text{-}out(A), U^\infty) = \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ is by simple formula manipulation.

(1)	$f^\infty(roll\text{-}out(A), U^\infty) \subseteq f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	Apply f^∞ to both sides of the result in part (a)
(2)	$f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq f^\infty f^\infty(roll\text{-}out(A), U^\infty)$	Apply f^∞ to both sides of the result in part (b)
(3)	$f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq f^\infty(roll\text{-}out(A), U^\infty)$	f^∞ is idempotent
(4)	$f^\infty(roll\text{-}out(A), U^\infty) = f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	Combine (1) and (3)
(5)	$b^\infty f^\infty(roll\text{-}out(A), U^\infty) = b^\infty f^\infty DAP^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	Apply b^∞ to both sides of (4)
	$= \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	by definition

□

The Δ Inclusion Lemma

The Δ Inclusion Lemma, which brings together the Directly Affected Points Lemma and the Dual Path Lemma, captures a critical relationship between $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ and $\Delta^S(A, Base)$.

LEMMA (Δ INCLUSION LEMMA). *If* $v^\infty \in \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$ *then* $v \in \Delta^S(A, Base)$.

PROOF. Part (c) of the Directly Affected Points Lemma implies $v^\infty \in b^\infty f^\infty(roll\text{-}out(A), u^\infty)$ where $u^\infty \in \{ occurrences(roll\text{-}out(A), u) \mid u \in DAP^S(A, Base) \}$. We can divide $b^\infty f^\infty(roll\text{-}out(A), u^\infty)$ at a vertex x^∞ such that $v^\infty \in b^\infty(roll\text{-}out(A), x^\infty)$ and $x^\infty \in f^\infty(roll\text{-}out(A), u^\infty)$. By definition these slices imply that paths $v^\infty \rightarrow^*_{c,f} x^\infty$ and $u^\infty \rightarrow^*_{c,f} x^\infty$ exist in $roll\text{-}out(A)$, and therefore, by Dual Path Lemma, $v \in b2\,f2\,f1\,(A, u) \cup b2\,b1\,f1\,(A, u)$. Thus, because $u \in DAP^S(A, Base)$, it follows that $v \in \Delta^S(A, Base)$ because $\Delta^S(A, Base) = b2\,f2\,f1\,DAP^S(A, Base) \cup b2\,b1\,f1\,DAP^S(A, Base)$. □

Lemma 2

Using the Δ Inclusion Lemma, it is possible to prove Lemma 2, the second half of the mutual-containment argument used to prove the Sufficiency Theorem.

LEMMA 2. *If no Type I or Type II interference exists in the integration of Base, A, and B then* $G_{M^-} \subseteq G_{roll\text{-}out(M)}$.

PROOF. The proof considers first the vertices and then the edges of G_{M^-}.

Vertices

For each occurrence v^∞ in G_{M^-}, we must show that $v^\infty \in V(roll\text{-}out(M))$. By definition, v^∞ must come from $roll\text{-}out(Base)$, $roll\text{-}out(A)$, or $roll\text{-}out(B)$ where it is associated with a vertex v and a sequence of call-sites S. If v and the call-sites from S are in G_M then v^∞ is in $roll\text{-}out(M)$. Because G_{M^-} is constructed from the three terms $Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$, $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, and

$\Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$, we consider three cases:

$v^{\infty} \in Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$.

If $v^{\infty} \in Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ then it must be in $roll\text{-}out(A)$, $roll\text{-}out(Base)$, and $roll\text{-}out(B)$; thus, A, $Base$, and B all contain v and the sequence of call-sites S. Because $b^{\infty}(roll\text{-}out(A), v^{\infty}) = b^{\infty}(roll\text{-}out(Base), v^{\infty}) = b^{\infty}(roll\text{-}out(B), v^{\infty})$, $b2(A, v)$, $b2(Base, v)$ and $b2(B, v)$ must all be equal, which implies $v \in Pre^{S}(A, Base, B) \subseteq V(G_M)$. To show that the call-sites in S are also in G_M, we show that they too are in $Pre^{S}(A, Base, B)$. This follows from the definition of Pre^{∞}, since all the vertices in $b^{\infty}(roll\text{-}out(A), v^{\infty})$ must have the same slice in $roll\text{-}out(A)$, $roll\text{-}out(Base)$ and $roll\text{-}out(B)$ and these slices include the scope vertices that correspond to the call-sites in S (*i.e.* the scope vertices that represent the scope statements that replace the call statements represented in S). Having v and the call-sites from S in G_M implies $v^{\infty} \in V(G_{roll\text{-}out(M)})$.

$v^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

The Δ Inclusion Lemma implies v is in $\Delta^{S}(A, Base)$. Thus, it remains to be shown that the call-sites in S are also in M. These call-sites correspond to scope vertices in $b^{\infty}(roll\text{-}out(A), v^{\infty})$. Because Δ^{∞} is backwards closed (*i.e.* if $v^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $x^{\infty} \in b^{\infty}(roll\text{-}out(A), v^{\infty})$ then $x^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$), these scope vertices are also in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Thus, the Δ Inclusion Lemma implies that the corresponding call-sites in S are in $\Delta^{S}(A, Base)$; thus, in G_M, which implies $v^{\infty} \in V(G_{roll\text{-}out(M)})$.

$v^{\infty} \in \Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$.

The argument here is the same as that for $v^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

Edges

For each edge $e^{\infty} = x^{\infty} \longrightarrow y^{\infty}$ in $E(G_{M^{\infty}})$, we must to show that $e^{\infty} \in E(G_{roll\text{-}out(M)})$. As in the vertex case, e^{∞} must exists in $roll\text{-}out(Base)$, $roll\text{-}out(A)$, or $roll\text{-}out(B)$ where it is associated with an edge e and a sequence of call-sites S. If e and the call-sites from S are in M then e^{∞} is in $G_{roll\text{-}out(M)}$. Because $G_{M^{\infty}}$ is constructed from the three terms $Induce\ Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$, $Induce\ \Delta^{\infty}(roll\text{-}out(A), roBase)$, or $Induce\ \Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$, we consider three cases:

$e^{\infty} = x^{\infty} \longrightarrow y^{\infty} \in E(Induce\ Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B)))$.

The vertex argument implies x and y (and the witness vertex of a def-order edge) are in $Pre^{S}(A, Base, B)$. Therefore $e \in E(Induce\ Pre^{S}(A, Base, B)) \subseteq E(G_M)$. As in the vertex argument, the call-sites corresponding to the scopes enclosing e^{∞} are also in $Pre^{S}(A, Base, B)$ and hence in G_M. Together these imply $e^{\infty} \in E(G_{roll\text{-}out(M)})$.

$e^{\infty} = x^{\infty} \longrightarrow y^{\infty} \in E(Induce\ \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)))$.

The Δ Inclusion Lemma implies x, y (and the witness vertex of a def-order edge) are in $\Delta^{S}(A, Base)$. Therefore e is in $Induce\ \Delta^{S}(A, Base) \subseteq E(G_M)$. As in the vertex argument, the call-sites corresponding to the scopes enclosing e^{∞} are also in $\Delta^{S}(A, Base)$ and hence in G_M. Together these imply $e^{\infty} \in E(G_{roll\text{-}out(M)})$.

$e^{\infty} = x^{\infty} \longrightarrow y^{\infty} \in E(Induce\ \Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base)))$.

The argument here is the same as that for $e^{\infty} \in Induce\ \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

□

The Sufficiency Theorem

We use the results of Lemma 1 and Lemma 2 to prove that the homogeneity test is sufficient. In other words, whenever the integration of *Base*, *A*, and *B* is successful and produces system *M* and *Base*, *A*, *B*, and G_M pass the homogeneity test then $G_{M^{\infty}}$ is equivalent to $G_{roll\text{-}out(M)}$ (which by definition implies that M^{∞} is

homogeneous). In Section 6.4 we show that the integration of the rolled-out systems is in fact successful, but first in Section 6.3 we show that the homogeneity test is a necessary test, that is whenever M^∞ is homogeneous, *Base*, A, B, and G_M pass the homogeneity test.

THEOREM (SUFFICIENCY THEOREM). *If Base, A, B and G_M pass the homogeneity test and no Type I or Type II interference exists in the integration of Base, A, and B then M^∞ is homogeneous and $G_{roll\text{-}out(M)} \approx G_{M^\infty}$.*

PROOF. The theorem follows immediately from Lemmas 1 and 2. □

6.3. The Necessity Theorem

Whereas, the Sufficiency Theorem proves that the homogeneity test is sufficient, the Necessity Theorem proves its necessity (*i.e.*, it proves that whenever G_{M^∞} is homogeneous, *Base*, A, B, and G_M pass the homogeneity test). The proof makes use of the Witness Lemma, which states that every vertex in G_M is witnessed in G_{M^∞}. A first glance this result may appear to be subsumed by Lemma 1. However, Lemma 1 assumes that *Base*, A, B, and G_M pass the homogeneity test; this assumption cannot be made in the Necessity Theorem (or the Witness Lemma) because the argument would then be circular: in the Necessity Theorem we are trying to prove that *Base*, A, B, and G_M pass the homogeneity test from assumption that G_{M^∞} is homogeneous.

LEMMA (WITNESS LEMMA). *For every vertex v in G_M, G_{M^∞} contains an occurrence of v.*

PROOF. Assume v is in the procedure dependence graph for procedure P. Vertex v is in G_M because it is in $Pre^S(A, Base, B)$, $\Delta^S(A, Base)$, or $\Delta^S(B, Base)$. We consider separately these three possibilities. First, if $v \in Pre^S(A, Base, B)$ then $b2(A, v)$, $b2(Base, v)$, and $b2(B, v)$ are, by definition, equivalent. In this case, let v^∞ be the depth-zero occurrence of v (recall that this occurrence, as defined in Section 6.1, is the one associated with the expansion of the empty sequence of call-sites). The equivalence of $b2$ slices implies $b^\infty(roll\text{-}out(A), v^\infty) = b^\infty(roll\text{-}out(Base), v^\infty) = b^\infty(roll\text{-}out(B), v^\infty)$; thus, v^∞ is in $Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ and hence in G_{M^∞}.

If $v \in \Delta^S(A, Base)$ (the proof for $\Delta^S(B, Base)$ is identical) then, by definition, a vertex u exists in $DAP^S(A, Base)$ such that v is either in $b2\,b1\,fl\,(A, u)$ or $b2\,f2\,fl\,(A, u)$; because the proofs for these two cases are virtually identical, we assume that $v \in b2\,b1\,fl\,(A, u)$. The first column of the following table shows the breakdown of this slice into its component slices (the intermediate vertex names are the same as those in Figure 6.4b). The second column of the table shows the corresponding slices in $roll\text{-}out(A)$ (note that for $y \in bl\,fl\,(A, z)$ in line 2, y and z are in the same procedure; thus, the relationship $y^\infty \in b^\infty f^\infty(roll\text{-}out(A), z^\infty)$ holds for *all* occurrences of y and z).

Breakdown of $b2\,b1\,fl\,(A, u)$		Corresponding slice in $roll\text{-}out(A)$
$z \in fl(A, u)$	$\Rightarrow$	$\forall z^\infty\ \exists u^\infty$ such that $z^\infty \in f^\infty(roll\text{-}out(A), u^\infty)$
$y \in bl\,fl(A, z)$	$\Rightarrow$	$\forall y^\infty\ \forall z^\infty$ such that $y^\infty \in b^\infty f^\infty(roll\text{-}out(A), z^\infty)$
$t \in b1(A, y)$	$\Rightarrow$	$\forall t^\infty\ \exists y^\infty$ such that $t^\infty \in b^\infty(roll\text{-}out(A), y^\infty)$
$v \in b2(A, t)$	$\Rightarrow$	$\forall t^\infty\ \exists v^\infty$ such that $v^\infty \in b^\infty(roll\text{-}out(A), t^\infty)$

For example, to see why $z \in fl(A, u)$ in the first line corresponds to $\forall z^\infty \exists u^\infty$ such that $z^\infty \in f^\infty(roll\text{-}out(A), u^\infty)$, let z^∞ be an arbitrary occurrence of z in $roll\text{-}out(A)$. Because the only interprocedural edges considered by an fl slice are parameter-out edges, procedure P_z transitively calls procedure P_u; therefore, a P_u-scope is expanded into every P_z-scope, which means that every occurrence of z can be associated with an occurrence of u. Assume that u^∞ is the occurrences of u associated with z^∞. Because u^∞ is connected to z^∞ by occurrences of the edges that connect z and u, $z \in fl(A, u)$ implies

$z^{\infty} \in f^{\infty}(roll\text{-}out(A), u^{\infty})$. Similar relationships hold for the other lines of the table.

The combination of the slices from the second column of the table, imply some occurrence of v^{∞} is in $b^{\infty} b^{\infty} b^{\infty} f^{\infty} f^{\infty}(roll\text{-}out(A), u^{\infty})$. Because b^{∞} and f^{∞} are idempotent and, by the Directly Affected Points Lemma, $u^{\infty} \in DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, this implies that v^{∞} is in $b^{\infty} f^{\infty} DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which, by definition, implies $v^{\infty} \in \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Therefore, an occurrence of v exists in G_{M^-}.

□

COROLLARY. *The scope containing v^{∞} (as identified in the Witness Lemma) is also in G_{M^-}.*

PROOF. Since the final operator of Pre^{∞} and Δ^{∞} is b^{∞} and the scope vertex for the scope containing v^{∞} is in the b^{∞} slice with respect to v^{∞}, the scope containing v^{∞} is also in G_{M^-}. □

THEOREM (NECESSITY THEOREM). *If G_{M^-} is homogeneous then Base, A, B, and G_M pass the homogeneity test.*

PROOF. We prove the contrapositive: if *Base*, *A*, *B*, and G_M fail the homogeneity test then G_{M^-} is inhomogeneous. Recall that, by definition, G_{M^-} is inhomogeneous if there is no system S such that $G_{roll\text{-}out(S)} \approx G_{M^-}$. One way that this can happen is if G_{M^-} contains two P-scope's such that one contains an occurrence of some vertex v and the other does not. For each point of failure in the homogeneity test (see Figure 5.10), we show that this situation occurs (*i.e.*, that G_{M^-} is, in fact, inhomogeneous).

Base, *A*, *B*, and G_M fail the homogeneity test if either call on the function *ExtraOccurrences* in Figure 5.10 returns **true**. Assume, without loss of generality, that this happens when *ExtraOccurrences* is called with the set of vertices in G_M but not B. The proof breaks down into three cases corresponding to the three points at which the function *ExtraOccurrences* may return **true**:

Case I: The absent-vertex test determines failure in line [4] in Figure 5.10.

If the absent-vertex determines failure then there must be a vertex v from G_M that is not in B and not in $blfl\,DAP^S(A, Base)$. Assume that v is in procedure P and let v^{∞} be the depth-zero occurrence of v. Since $v \notin blfl\,DAP^S(A, Base)$, any directly affected point that causes $Integrate^S$ to include v in $\Delta^S(A, Base)$ must do so because it is connected to v by a path that includes a call-site on P. Because none of the scopes corresponding to these call-sites enclose $roll\text{-}out(A, P)$, v^{∞} is not in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Furthermore, since v^{∞} is not in $roll\text{-}out(B)$ (recall that v is not in B), it is not in $\Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$ or $Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$. Thus, it is impossible for v^{∞} is be in G_{M^-}.

The preceding argument implies that v^{∞}, the depth-zero occurrence of v, is not in G_{M^-}. If the scope for this occurrence is in G_{M^-} then G_{M^-} is inhomogeneous because the Corollary to the Witness Lemma implies that G_{M^-} contains a P-scope with an occurrence of v.

On the other hand, if the scope for the depth-zero occurrence of v is not in G_{M^-}, we conclude that G_{M^-} is inhomogeneous by the following argument: (1) the Corollary to the Witness Lemma implies that G_{M^-} contains a P-scope; therefore, any system S whose roll-out is isomorphic to G_{M^-} must contain a procedure P. (2) Any system S that contains a procedure P contains $roll\text{-}out(S, P)$, the scope for the depth-zero occurrence of v. Together (1) and (2) imply that no system S exists such that $G_{roll\text{-}out(S)} \approx G_{M^-}$; consequently, G_{M^-} is inhomogeneous.

Case II: The subset-test of the absent-call-site test determines failure in line [7] of Figure 5.10.

In this case there is a vertex v in a procedure transitively callable from a call-site c_0 that is in G_M but not B such that v is not in A. If c_0 is not in $blfl\,DAP^S(A, Base)$ then, by Case I, G_{M^-} is inhomogeneous. Otherwise, if c_0 is in $blfl\,DAP^S(A, Base)$, then every occurrence of c_0 in $roll\text{-}out(A)$ is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and therefore in G_{M^-}. Now, consider the sequence of call-sites

$c_0, \cdots, c_n$ where c_n calls the procedure containing v. It is shown below that either (a) one of the scopes resulting from the expansion of a call-site c_i $(0 \le i \le n-1)$ does not contain an occurrence of c_{i+1} or that (b) the scope resulting from the expansion of call-site c_n does not contain an occurrence of v. Thus, G_{M^-} is inhomogeneous because the Corollary to the Witness Lemma implies that a scope with an occurrence of c_{i+1} and a scope with an occurrence of v exist in G_{M^-}.

Consider any scope produced by the expansion of c_0 in *roll-out*(A). Because c_0 is not in B the vertex occurrences in this scope in G_{M^-} must come from Δ^∞(*roll-out*(A), *roll-out*$(Base)$). If this scope does not contain an occurrence of c_1 then use c_0 for c_i in Case (a) above. If the occurrence of c_1 is in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) then repeat the preceding argument with c_1 and c_2 in place of c_0 and c_1, respectively. Continue repeating this argument until either an occurrence of c_i is not in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) or an occurrence of c_n is included in Δ^∞(*roll-out*(A), *roll-out*$(Base)$). In the later case, G_{M^-} contains a scope created from c_n without an occurrence of v, since v is not in A, which corresponds to Case (b) above.

Case III: The slice-need-set test of the absent-call-site test determines failure in line [13] of Figure 5.10.
In this case there is a call-site c_0 and a vertex v such that c_0 is in M but not B and *Linkage*(v) is transferred backward through a sequence of call-sites $c_0, \cdots, c_n$ to the slice-need set for the procedure called by c_0 where it is then mapped, at c_0, to a set of actual-in, call-site, and actual-out vertices that are not DAP-connected (see lines [8]-[10] of Figure 5.10). Let P_i denote the procedure that contains call-site c_i. The proof considers the expansion of $c_0 \cdots c_n$ that includes the root occurrence of c_0 (this expansion takes place in *roll-out*(A, P_0)). As shown below, either there exists a scope without an occurrence of c_i or there exists a scope without an occurrence of v. Thus, as in Case II, G_{M^-} is inhomogeneous because the Corollary to the Witness Lemma implies that a scope with an occurrence of c_i and a scope with an occurrence of v exist in G_{M^-}.

We now prove that a scope without an occurrence of c_i or a scope without an occurrence of v exists in G_{M^-}. First, the argument for call-sites is the same as in Case II: starting from the scope produced by the expansion of c_0, if the scope produced by the expansion of c_i does not contain an occurrence of c_{i+1} then G_{M^-} is inhomogeneous.

On the other hand, if the scopes produced by the expansion of $c_0 \cdots c_n$ are all in G_{M^-} then we prove, by contradiction, that v^∞, the occurrence of v produced by the expansion of $c_0, \cdots, c_n$, is not in G_{M^-}. Recall that v^∞ is the occurrence of v in *roll-out*(A, P_0), and that, since c_0 is not in B, v^∞ is in G_{M^-} iff it is in Δ^∞(*roll-out*(A), *roll-out*$(Base)$). Thus, we assume $v^\infty \in \Delta^\infty$(*roll-out*$(A)$, *roll-out*$(Base)$) and prove that this contradicts the assumption of Case III that v causes the slice-need-set test to determine failure.

To begin with, $v^\infty \in \Delta^\infty$(*roll-out*$(A)$, *roll-out*$(Base)$) implies that v^∞ is in $b^\infty f^\infty DAP^\infty$(*roll-out*$(A)$, *roll-out*$(Base)$) and thus, by the Directly Affected Points Lemma, v^∞ is in $b^\infty f^\infty$(*roll-out*$(A), u^\infty$) where u^∞ is in DAP^∞(*roll-out*(A), *roll-out*$(Base)$) and u is in $DAP^S(A, Base)$. In addition, since in order for u^∞ to cause v^∞ to be included in Δ^∞(*roll-out*(A), *roll-out*$(Base)$), it must be in *roll-out*(A, P_0), u must be in a procedure (transitively called from) either P_i, for $0 \le i \le n$, or P_v. Finally, the observation that $v^\infty \in b^\infty f^\infty$(*roll-out*$(A), u^\infty$) implies there exists a vertex x^∞ and paths $v^\infty \rightarrow^*_{c,f} x^\infty$ and $u^\infty \rightarrow^*_{c,f} x^\infty$ in *roll-out*(A).

First, consider the case where u is in P_v (the procedure containing v) or a procedure transitively called by procedure P_v. In this case the paths $v^\infty \rightarrow^*_{c,f} x^\infty$ and $u^\infty \rightarrow^*_{c,f} x^\infty$ imply that v is in $b1\,f1\,DAP^S(A, Base)$ which means that v is excluded from begin considered by the slice-need-set test and thus, cannot cause this test to determine failure. This contradicts the assumption of Case III that v causes the slice-need-set test to determine failure.

Before considering the general case of u being in P_i (or a procedure transitively called by procedure P_i) for $0 \le i \le n$, we consider the case where u is in P_n (or a procedure transitively called by procedure P_n). In this case, either $v^\infty \rightarrow^*_{c,f} x^\infty$ or $u^\infty \rightarrow^*_{c,f} x^\infty$ must include a transfer-in, scope, or transfer-out vertex associated with the scope produced by the expansion of c_n. The portion of these paths in the same scope as v^∞ implies that *Linkage*(v) contains the corresponding formal-in, entry, or formal-out vertex; the remainder of these paths implies that, at c_n, the corresponding actual-in, entry, or actual-out vertex is DAP-connected. This contradicts the assumption that *Linkage*(v) is propagated to c_0 through c_n (and $c_1, \cdots, c_{n-1}$).

Now, consider the general case where u is in P_i or a procedure transitively called by procedure P_i, for $0 \le i \le n$. The proof of this case is identical to the case where u is in P_n of a procedure transitively called by procedure P_n, except that the portions of $v^\infty \rightarrow^*_{c,f} x^\infty$ and $u^\infty \rightarrow^*_{c,f} x^\infty$ in the scopes produced by the expansion of $c_i, \cdots, c_{n-1}$ imply that the transfer functions for $P_i, \cdots, P_{n-1}$ transfer *Linkage*(v) to c_i (where it is mapped to a DAP-connected vertex).

The above argument shows that the assumption of Case III leads to a contradiction; thus, it must be the case that v^∞ is not in $\Delta^\infty(\textit{roll-out}(A), \textit{roll-out}(\textit{Base}))$ and therefore not in G_M-.

□

6.4. The Type I Interference Theorem

The final theorem used in the proof of the Syntactic Correctness Theorem is the Type I Interference Theorem, which proves the equivalence of Type I interference in *Integrate*^{S} and *Integrate*$^\infty$ under certain assumptions given below. The proof makes use of three results about directly affected points (lemmas I1, I2, and I3), which because they apply to both systems and rolled-out systems, are stated without superscript designators. The proofs of lemmas I1, I2, and I3 make use of the four ways a vertex may be in $DAP(X, Y)$, which are shown in Figure 6.5.

LEMMA I1. $DAP(X_1 \cup X_2, Y) \subseteq DAP(X_1, Y) \cup DAP(X_2, Y)$.

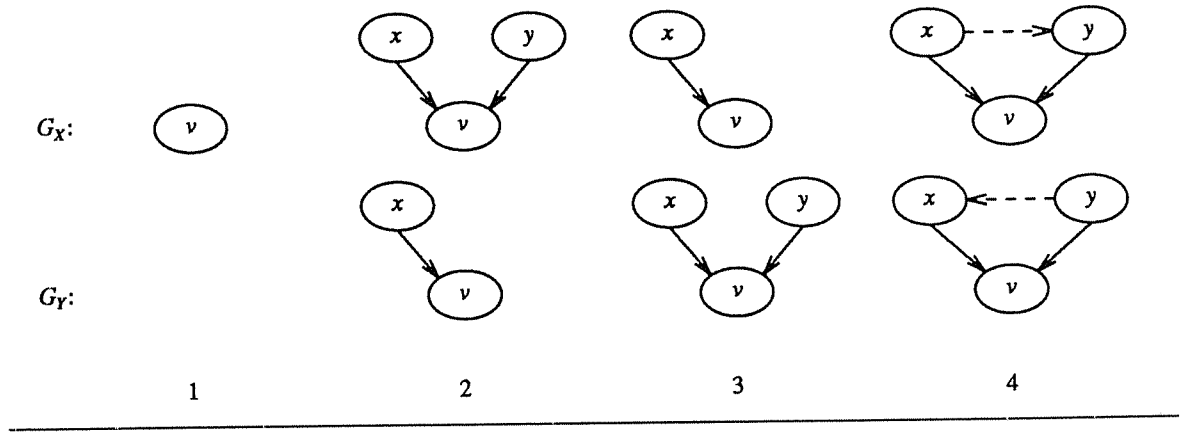


Figure 6.5. The four possible ways a vertex v can be in $DAP(X,Y)$ are (1) if v exists in X but not Y, (2) if v has an edge $(y \rightarrow v)$ in X but not Y, (3) if v has an edge $(y \rightarrow v)$ in Y but not X, and (4) if the direction of a def-order edge between two of v's predecessors is reversed (*i.e.*, v has different incoming def-order edges).

PROOF. The proof refers to the four cases shown in Figure 6.5.

(1) Without loss of generality assume v exists in X_1; vertex v is not in Y; therefore, $v \in DAP(X_1, Y)$.

(2) Without loss of generality assume $y \longrightarrow v$ exists in X_1; edge $y \longrightarrow v$ does not exist in Y; therefore, $v \in DAP(X_1, Y)$.

(3) $y \longrightarrow v$ does not exist in X_1 (or X_2) and exists in Y; therefore, $v \in DAP(X_1, Y)$.

(4) Without loss of generality assume $x \longrightarrow_{do(v)} y$ exists in X_1; edge $y \longrightarrow_{do(v)} x$ exists in Y; therefore, $v \in DAP(X_1, Y)$.

□

LEMMA I2. $DAP(Induce\ \Delta(A, Base), A) = DAP(Induce\ Pre(A, Base, B), A) = \emptyset$.

PROOF. The proofs for $\Delta(A, Base)$ and $Pre(A, Base, B)$ are virtually identical. Thus, we prove only the case for $\Delta(A, Base)$; let v be a vertex from $\Delta(A, Base)$.

(1) If v exists in $\Delta(A, Base)$ then, by definition, it exists in A; therefore, Case (1) cannot arise.

(2), (3), and (4)

(For system dependence graphs A and $Base$, we do not need to consider formal-in, entry, and actual-out vertices, because, by definition, such vertices are in $DAP(X, Y)$ iff they are in X but not Y.) If v is not a formal-in, entry, or actual-out vertex then all of v's immediate predecessors from A are included in $\Delta(A, Base)$ by the backward slice that includes v. Therefore, cases (2), (3), and (4) cannot arise.

□

LEMMA I3. $DAP(G_M, A) \subseteq DAP(Induce\ \Delta(B, Base), A)$.

PROOF. Because G_M is the union of $\Delta(A, Base)$, $\Delta(B, Base)$, and $Pre(A, base, B)$, the lemma follows from Lemmas I1 and I2. □

The Type I Interference Theorem

For reference, the definition of Type I interference in both $Integrate^S$ and $Integrate^\infty$ is repeated before the proof the Type I Interference Theorem.

DEFINITION. (Type I Interference).

$Interference^S(A, Base, B) \triangleq$
$\Delta^S(A, Base) \cap DAP^S(G_M, A) \neq \emptyset$ or
$\Delta^S(B, Base) \cap DAP^S(G_M, B) \neq \emptyset$.

$Interference^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B)) \triangleq$
$\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base)) \cap DAP^\infty(G_{M^-}, roll\text{-}out(A)) \neq \emptyset$ or
$\Delta^\infty(roll\text{-}out(B), roll\text{-}out(Base)) \cap DAP^\infty(G_{M^-}, roll\text{-}out(B)) \neq \emptyset$.

LEMMA (TYPE I INTERFERENCE LEMMA).

(1) *Assume* M^∞ *is homogeneous, if* $Interference^S(A, Base, B)$ *then* $Interference^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$.

(2) *If* $Interference^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *then* $Interference^S(A, Base, B)$ *or* G_M *is feasible.*

Note that this lemma indicates that $Interference^S$ is "weaker" than $Interference^\infty$ because $Interference^S$ cannot test differences in interprocedural edges. Such differences arise when in G_M there is *either* a call-site such that two actual-in vertices are connected to a single formal-in vertex *or* two formal-out vertices are connected to a single actual-out vertex. Although $Interference^S$ cannot detect these situations, in both

cases

G_M is infeasible (and so *Integrate*^{S} reports Type II interference rather than Type I interference).

PROOF. (1). We first show that when *Interference*$^S(A, Base, B)$ holds, there is a vertex v in $\Delta^S(A, Base)$ and $\Delta^S(B, Base)$ that has different incoming edges in A and B. We then argue that occurrences of v with different incoming edge occurrences exist in *roll-out*(A) and *roll-out*(B), and from this that *Interference*$^\infty$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) holds. First, by definition, the existence of *Interference*^{S} implies $\Delta^S(A, Base) \cap DAP^S(G_M, A) \neq \emptyset$ or $\Delta^S(B, Base) \cap DAP^S(G_M, B) \neq \emptyset$; without loss of generality assume the former and let v be a vertex in both $\Delta^S(A, Base)$ and $DAP^S(G_M, A)$. Thus, by Lemma I3, v is in DAP^S(*Induce* $\Delta^S(B, Base), A)$, which implies v is in $\Delta^S(B, Base)$. Finally, because v is in DAP^S(*Induce* $\Delta^S(B, Base), A)$ and also in A, it must have different incoming edges in A and B (cases 2-4 of Figure 6.5). (Note this rules out v being a formal-in, entry, or actual-out vertex because these vertices are in DAP^S(*Induce* $\Delta^S(B, Base), A)$ only if they are not in A).

We now demonstrate that *Interference*$^\infty$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) holds because of an occurrence of v. By the Witness Lemma, $v \in \Delta^S(A, Base)$ and $v \in \Delta^S(B, Base)$ imply that there are occurrences of v, v_A^∞ and v_B^∞, in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) and Δ^∞(*roll-out*(B), *roll-out*$(Base)$), respectively. Because v_A^∞ and v_B^∞ have different incoming edge occurrences in *roll-out*(A) and *roll-out*(B) (this follows from v having different incoming edges in A and B), and because M^∞ is homogeneous (and therefore all occurrences in G_{M^∞} have the same incoming edges), either $v_A^\infty \in DAP^\infty(G_{M^\infty}$, *roll-out*$(A))$ or $v_B^\infty \in DAP^\infty(G_{M^\infty}$, *roll-out*$(B))$. Consequently, because $v_A^\infty \in \Delta^\infty$(*roll-out*$(A)$, *roll-out*$(Base)$) and $v_B^\infty \in \Delta^\infty$(*roll-out*$(B)$, *roll-out*$(Base)$), *Interference*$^\infty$(*roll-out*$(A)$, *roll-out*$(Base)$, *roll-out*$(B)$) must hold.

(2). Similar to Case 1, if *Interference*$^\infty$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) holds, then without loss of generality there is a vertex occurrence v^∞ in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) and $DAP^\infty(G_{M^\infty}$, *roll-out*$(A))$. Also similar to Case 1, this implies that v^∞ is in Δ^∞(*roll-out*(B), *roll-out*$(Base)$) and has different incoming edge occurrences in *roll-out*(A) and *roll-out*(B).

Our goal is to demonstrate that *Interference*$^S(A, Base, B)$ holds because v in is $\Delta^S(B, Base)$ and $DAP^S(G_M, B)$. The argument makes use of the fact that v^∞ is in Δ^∞(*roll-out*(A), *roll-out*$(Base)$) and Δ^∞(*roll-out*(B), *roll-out*$(Base)$) and has different incoming edge occurrences in *roll-out*(A) and *roll-out*(B) but does not use the fact that v^∞ is in $DAP^\infty(G_{M^\infty}$, *roll-out*$(A))$. Thus, we can assume, again without loss of generality, that the difference in incoming edge occurrences for v^∞ is an edge $y^\infty \longrightarrow v^\infty$ that exists in *roll-out*(A) but not *roll-out*(B).

Because $v^\infty \in \Delta^\infty$(*roll-out*$(B)$, *roll-out*$(Base)$), the Δ Inclusion Lemma implies that v is in $\Delta^S(B, Base)$. Thus, what remains to be shown is that $v \in DAP^S(G_M, B)$. The latter argument is complicated by the fact that $y^\infty \longrightarrow v^\infty$ does not identify a unique edge in G_A. This is because transfer-in, scope, and transfer-out vertices correspond to pairs of vertices in A; therefore, $y^\infty \longrightarrow v^\infty$ may identify two or even three edges in A. We first prove $v \in DAP^S(G_M, B)$ under the restrictions that y^∞ and v^∞ are not transfer-in, scope, or transfer-out vertices and then remove these restrictions.

First, under the restriction that the edge occurrence $y^\infty \longrightarrow v^\infty$ identifies a unique edge $y \longrightarrow v$ in A, the edge $y \longrightarrow v$ is in *Induce* $\Delta^S(A, Base)$ and thus in G_M, because the Δ Inclusion Lemma implies that y and v are in $\Delta^S(A, Base)$. In general, an edge occurrence e^∞ exists in $G_{\text{roll-out}(B)}$ if there is an edge e in G_B and a sequence of call-sites from B whose expansion produces occurrences of the statements represented by the endpoints of e. Thus, because $y^\infty \longrightarrow v^\infty$ does not exist in *roll-out*(B), either $y \longrightarrow v$ is not in G_B or the necessary sequence of call-sites is not in B. However, since having v^∞ in *roll-out*(B) implies that B contains the necessary call-sites, we conclude that $y \longrightarrow v$ is not in B. Together , having $y \longrightarrow v$ in G_M (from *Induce* $\Delta^S(A, Base)$) but not B, implies v is in $DAP^S(G_M, B)$.

The preceding argument shows that Part (2) of the Type I Interference Theorem holds if y^∞ and v^∞ are not transfer-in, scope, or transfer-out vertices we now consider removing these restrictions, by first removing the restriction on v^∞ and then the restriction on y^∞. If v^∞ is a transfer-in, scope, or transfer-out vertex then v^∞ corresponds to two vertices v_1 and v_2 in A (for example, if v^∞ is a transfer-in vertex then v_1 is an actual-in vertex and v_2 is a formal-in vertex). Thus, $y^\infty \longrightarrow v^\infty$ identifies edges $y \longrightarrow v_1$ and $v_1 \longrightarrow v_2$ in A. In general, $y^\infty \longrightarrow v^\infty$ is not in $G_{roll\text{-}out(B)}$ because either $y \longrightarrow v_1$ or $v_1 \longrightarrow v_2$ are not in G_B, or one of the necessary call-sites is not in B. By definition, because v^∞ exists in *roll-out*(B), the necessary call-sites are in B and v_1 and v_2 are all in G_B. Since interprocedural edges are determined solely by their endpoints, this implies $v_1 \longrightarrow v_2$ is in G_B and thus for $y^\infty \longrightarrow v^\infty$ to be absent from *roll-out*(B), $y \longrightarrow v_1$ must be absent from B. Similar to the restricted case, the Δ Inclusion Lemma implies that $y \longrightarrow v_1$ is in *Induce* $\Delta^S(A, Base)$ and thus G_M; therefore, $v \in DAP^S(G_M, B)$.

Lifting the restriction on y^∞ is slightly more complicated because y^∞ is not guaranteed to exist in *roll-out*(B); we use the assumption that G_M is feasible (in particular that each parameter position is occupied by exactly one parameter) to show that $v \in DAP^S(G_M, B)$. If y^∞ is a transfer-in, scope, or transfer-out vertex then $y^\infty \longrightarrow v^\infty$ identifies the two edges $y_1 \longrightarrow y_2$ and $y_2 \longrightarrow v$ in A (if v^∞ is a transfer-in, scope, or transfer-out vertex then let v be v_1 from the preceding paragraph). Because $y^\infty \longrightarrow v^\infty$ is not in *roll-out*(B) but v^∞ is, either $y_1 \longrightarrow y_2$, $y_2 \longrightarrow v$, or both are not in G_B. We consider the second the third possibilities together before considering the first:

a) $y_2 \longrightarrow v$ is not in G_B.
The Δ Inclusion Lemma implies that $y_2 \longrightarrow v$ is in *Induce* $\Delta^S(A, Base)$ and thus in G_M; therefore, $v \in DAP^S(G_M, B)$.

b) G_B contains $y_2 \longrightarrow v$ but not $y_1 \longrightarrow y_2$
In any feasible system dependence graph (*i.e.* an system dependence graph that corresponds to some system), the vertices representing a call-site are always in 1-1 correspondence with the linkage vertices of the called procedure (otherwise, there would be two actual parameters corresponding to a single formal parameter or two formal parameters corresponding to a single actual parameter). Because y_2 exists in B and $y_1 \longrightarrow y_2$ does not, the 1-1 correspondence of the vertices representing a call-site with linkage vertices implies there must be a vertex y' such that $y' \longrightarrow y_2$ in G_B. Let y'^∞ be the vertex in *roll-out*(B) that represents the condensation of y' and y_2. Because $y'^\infty \longrightarrow v^\infty$ is in $G_{roll\text{-}out(B)}$ and v^∞ is in $\Delta^\infty(roll\text{-}out(B), roll\text{-}out(Base))$, y'^∞ is also in $\Delta^\infty(roll\text{-}out(B), roll\text{-}out(Base))$; therefore, by the Δ Inclusion Lemma, y' and y_2 are in $\Delta^S(B, Base)$ and thus $y' \longrightarrow y_2$ is in G_M. Since $y_1 \longrightarrow y_2$ is also in G_M (from *Induce* $\Delta^S(A, Base)$), G_M contains $y_1 \longrightarrow y_2$ and $y' \longrightarrow y_2$, which makes it infeasible (both y_1 and y' correspond to the parameter position associated with y_2). This contradicts the assumption that G_M is feasible; therefore Case (b) cannot arise.

□

6.5. Syntactic Correctness Theorem

Putting together the pieces in the preceding sections, it is now possible to establish that *Integrate*^{S} satisfies the requirements on I^S from Version 2 of the Revised Model of Program Integration (See Chapter 4).

THEOREM (SYNTACTIC CORRECTNESS THEOREM). *Integrate*^{S} *satisfies Properties (2)(ii) and (2)(iii) of Version 2 of the Revised Integration Model of Program Integration. In other words,*

(a) *If Integrate*$^S(A, Base, B)$ *succeeds and produces M, and Base, A, B and* G_M *pass the homogeneity test then Integrate*$^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *succeeds and produces* M^∞ *such that* M^∞ *is homogeneous and* $M^\infty = roll\text{-}out(M)$.

(b) *If Integrate*$^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *succeeds and produces* M^∞ *and* M^∞ *is homogeneous then Integrate*$^S(A, Base, B)$ *succeeds and produces M such that Base, A, B and* G_M *pass the*

homogeneity test, and roll-out$(M) = M^{\infty}$.

PROOF.

(a) To show that *Integrate*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) succeeds requires showing that *Interference*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) is **false** and that $G_{M^{\infty}}$ is feasible. By assumption, *Integrate*$^{S}(A, Base, B)$ succeeds, which implies that *Interference*$^{S}(A, Base, B)$ is **false** and G_M is feasible. Thus, the contrapositive of Part 2 of the Type I Interference Lemma implies that *Interference*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) is **false**. By definition, $G_{M^{\infty}}$ is feasible if there is a set of programs P such that $G_P \approx G_{M^{\infty}}$. The Sufficiency Theorem implies that $G_{\text{roll-out}(M)} \approx G_{M^{\infty}}$; thus, *roll-out*$(M)$ is one set of programs whose set of procedure dependence graphs is isomorphic to $G_{M^{\infty}}$. This implies that no Type II interference exists and thus *Integrate*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) succeeds in producing set of merged programs M^{∞}. Finally, because the Sufficiency Theorem implies that $G_{\text{roll-out}(M)} \approx G_{M^{\infty}}$, $G_{M^{\infty}}$ is, by definition, homogeneous and $M^{\infty} =$ *roll-out*(M).

(b) To show the *Integrate*$^{S}(A, Base, B)$ succeeds requires showing that *Interference*$^{S}(A, Base, B)$ is **false** and that G_M is feasible. By assumption, *Integrate*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) succeeds, which implies that *Interference*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)) is **false** and $G_{M^{\infty}}$ is feasible. Thus, the contrapositive of Part 1 of the Type I Interference Lemma implies that *Interference*$^{S}(A, Base, B)$ is **false**. By definition, M^{∞} is homogeneous iff system N exists such that $G_{\text{roll-out}(N)} \approx G_{M^{\infty}}$. We show below that G_N is isomorphic to G_M; thus, G_M is feasible (*i.e.*, there is no Type II interference) since N is a system whose system dependence graph is isomorphic to G_M. Therefore *Integrate*$^{S}(A, Base, B)$ succeeds in producing a system M. Finally, because the Necessity Theorem implies that *Base*, A, B, and G_M pass the homogeneity test and no Type I or Type II interference exists in the integration of *Base*, A, and B, the Sufficiency Theorem implies that $G_{\text{roll-out}(M)} \approx G_{M^{\infty}}$ (and thus, *roll-out*$(M) = M^{\infty}$).

We now prove that $G_N \approx G_M$; the proof is by mutual containment.

$G_N \subseteq G_M$:

Vertices

For a vertex $v \in V(G_N)$, an occurrence v^{∞} of v exists in $G_{\text{roll-out}(N)}$ and therefore in $G_{M^{\infty}}$. To be in $G_{M^{\infty}}$, v^{∞} must be in at least one of Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$), *Pre*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B)), or Δ^{∞}(*roll-out*(B), *roll-out*$(Base)$). For Δ^{∞}, the Δ Inclusion Lemma implies that $v \in \Delta^S$. For *Pre*$^{\infty}$, v^{∞} is in *roll-out*(A), *roll-out*$(Base)$, and *roll-out*(B) where b^{∞}(*roll-out*(A), v^{∞}) = b^{∞}(*roll-out*$(Base)$, v^{∞}) = b^{∞}(*roll-out*(B), v^{∞}). This implies v is in A, *Base* and B and further that $b2(A, v)$, $b2(Base, v)$ and $b2(B, v)$ are all equal; thus, $v \in Pre^S(A, Base, B)$. Finally, since v is in at least one of $\Delta^S(A, Base)$, $Pre^S(A, Base, B)$, or $\Delta^S(B, Base)$, it is in G_M.

Edges

First, the interprocedural edges of G_N can be determined from the vertices of G_N; therefore, the vertex argument implies that all the interprocedural edges from G_N are in G_M. Second, for an intraprocedural edge $e \in E(G_N)$, an occurrence e^{∞} of e exists in $G_{\text{roll-out}(N)}$ and therefore in $G_{M^{\infty}}$. The remainder of the argument is similar to the argument for vertices. Occurrence e^{∞} is in at least one of *Induce* Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$), *Induce Pre*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B))), or *Induce* Δ^{∞}(*roll-out*(B), *roll-out*$(Base)$) and by the vertex case the end points of e (and the witness vertex for def-order edges) are in $\Delta^S(A, Base)$, $Pre^S(A, Base, B)$, or $\Delta^S(B, Base)$. Because, an occurrence of e exists in *Induce* Δ^{∞}(*roll-out*(A), *roll-out*$(Base)$), *Induce Pre*$^{\infty}$(*roll-out*(A), *roll-out*$(Base)$, *roll-out*(B))), or *Induce* Δ^{∞}(*roll-out*(B), *roll-out*$(Base)$)) iff e exists in A, *Base*, or B, respectively, e must be in at least

one of *Induce* $\Delta^S(A, Base)$), *Induce* $Pre^S(A, Base, B)$, or *Induce* $\Delta^S(A, Base)$), which implies $e \in E(G_M)$.

$G_M \subseteq G_N$:

Vertices

For $v \in V(G_M)$, the Witness Lemma implies an occurrence of v exists in G_{M^-}. Because $G_{roll\text{-}out(N)} \approx G_{M^-}$, this occurrence is in $G_{roll\text{-}out(N)}$ and therefore v must be in $V(G_N)$.

Edges

First, the interprocedural edges of G_M can be determined from the vertices of G_M; therefore, the vertex argument implies that all the interprocedural edges from G_M are in G_N. Second, consider an intraprocedural edge $e = x \longrightarrow y \in E(G_M)$. While the Witness Lemma implies that an occurrence of y, y^∞ exists in G_{M^-}, the proof of the Witness Lemma actual proves a stronger result:

(1) if y is in $Pre^S(A, Base, B)$ then y^∞ is in $Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$,
(2) if y is in $\Delta^S(A, Base)$ then y^∞ is in $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$, and
(3) if y is in $\Delta^S(B, Base)$ then y^∞ is in $\Delta^\infty(roll\text{-}out(B), roll\text{-}out(Base))$,

We use this stronger result to prove that e is in G_N. By definition, e is in (at least one of) *Induce* $\Delta^S(A, Base)$, *Induce* $\Delta^S(B, Base)$, or *Induce* $Pre^S(A, Base, B)$. Therefore, we consider three cases (for a def-order edge $a \longrightarrow_{do(c)} b$, c replaces y, and a and b replace x in the following arguments).

1) $e = x \longrightarrow y \in E$ (*Induce* $Pre^S(A, Base, B)$).

(1) $y \in Pre^S(A, Base, B)$	by assumption
(2) $\exists y^\infty$ *s.t.* $y^\infty \in Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$	by the proof of the Witness Lemma
(3) $y^\infty \in V(roll\text{-}out(Base))$	by definition of Pre^∞
(4) $x \longrightarrow y \in E(Base)$	by assumption
(5) $\exists x^\infty$ *s.t.* $x^\infty \longrightarrow y^\infty \in E(roll\text{-}out(Base))$	follows from (3) and (4)
(6) $x^\infty \in Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$	follows from (2) and (5)
(7) $x^\infty \longrightarrow y^\infty \in$ *Induce* $Pre^\infty(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$	from (2), (5), and (6) by definition of Pre^∞
(8) $x^\infty \longrightarrow y^\infty \in G_{M^-}$	from (7) by definition of G_{M^-}

2) $e = x \longrightarrow y \in E$ (*Induce* $\Delta^S(A, Base)$).

(1) $y \in \Delta^S(A, Base)$	by assumption
(2) $\exists y^\infty$ *s.t.* $y^\infty \in \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	by the proof of the Witness Lemma
(3) $y^\infty \in V(roll\text{-}out(A))$	by definition of Δ^∞
(4) $x \longrightarrow y \in E(A)$	by assumption
(5) $\exists x^\infty$ *s.t.* $x^\infty \longrightarrow y^\infty \in E(roll\text{-}out(A))$	follows from (3) and (4)
(6) $x^\infty \in \Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	follows from (2) and (5)
(7) $x^\infty \longrightarrow y^\infty \in$ *Induce* $\Delta^\infty(roll\text{-}out(A), roll\text{-}out(Base))$	from (2), (5), and (6) by definition of Δ^∞
(8) $x^\infty \longrightarrow y^\infty \in G_{M^-}$	from (7) by definition of G_{M^-}

3) $e = x \longrightarrow y \in E$ (*Induce* $\Delta^S(B, Base)$).

The proof is the same as the proof for $x \longrightarrow y \in E$ (*Induce* $\Delta^S(A, Base)$).

The above argument implies an occurrence of e exists in G_{M^-}. By assumption, this occurrence of is in $G_{roll\text{-}out(N)}$ and therefore e is in G_N. □

CHAPTER 7

ROLL-OUT IS A SEMANTICS-PRESERVING TRANSFORMATION

This chapter demonstrates that roll-out is a semantics-preserving transformation and thus justifies the use of roll-out in Version 2 of the Revised Model of Program Integration given in Chapter 4. Recall that we used the concept of roll-out in this model to relate the sequences of values produced by a statement s in different contexts during the execution of system S with the sequences of values produced by the occurrences of s in the evaluation of *roll-out*(S). This relationship allows us to prove that M (the result of integrating A and B with respect to *Base*) captures the changed and preserved behavior of A and B with respect to *Base*, by proving that M^{∞} (the result of integrating *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*)) captures the changed and preserved behavior of *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*).

In addition to showing that roll-out is a semantics-preserving transformation, two other results are required. First, we must show that the diagram in Figure 7.1 commutes (*i.e.*, that $Integrate^{\infty} \circ roll\text{-}out$ equals $roll\text{-}out \circ Integrate^{S}$); this was done in Chapter 6 where we demonstrated that $roll\text{-}out(M) = M^{\infty}$ holds if either integration is successful. The second result we need to show is that M^{∞} captures the changed and preserved behavior of *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*); this is done in Chapter 8. The results of Chapters 6 and 8, when combined with the result from this chapter, imply that M captures the changed and preserved behavior of A and B with respect to *Base*: (1) M^{∞} captures the changed and preserved behavior of *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*), (2) $M^{\infty} = roll\text{-}out(M)$, and (3) *roll-out* is a semantics-preserving transformation.

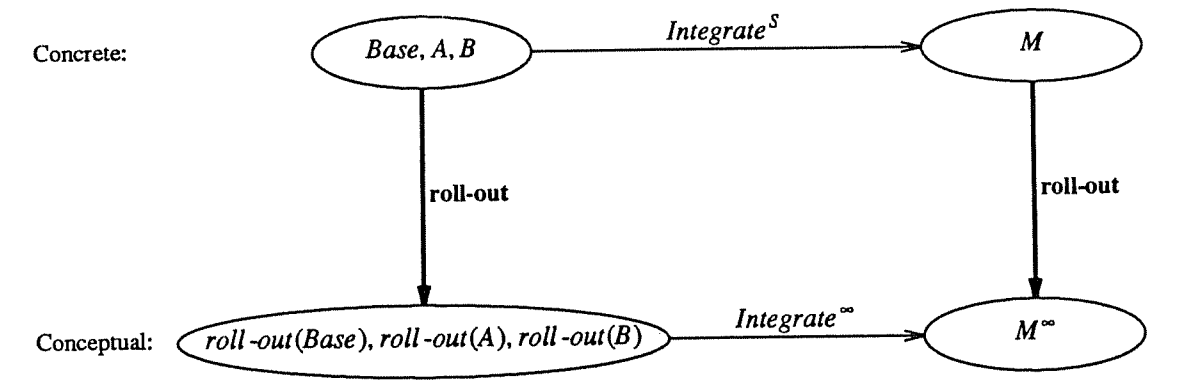


Figure 7.1. The commutative square that captures Version 2 of the Revised Model for Multi-Procedure Integration. This chapter deals with the transition from the concrete level to the conceptual level, which is highlighted in bold.

The proof that roll-out is a semantics-preserving transformation requires first proving that call-site expansion is a semantics-preserving transformation. Before tackling these proofs, we present a denotational semantics for systems, finite rolled-out systems, and infinite rolled-out systems. It is assumed that the reader has some familiarity with the denotational description of languages; those lacking such background are referred to [Gordon79] or [Schmidt86].

To simplify matters, the following assumptions are made in this chapter: procedures have a single parameter and this parameter is (always) potentially modified by the procedure (*i.e.* in *GMOD*). The first assumption avoids the added complexity of an induction over the number of parameters. Augmenting a denotational semantics to include procedures having more than one parameter is discussed in [Gordon79]. The second assumption simplifies the equations for call and scope statements. For example, without this assumption two kinds of scope statements must be handled: those with transfer-out statements and those without. Neither assumption affects the technical results contained in this chapter.

7.1. Meaning Functions for Systems, Finite Rolled-out Systems, and Infinite Rolled-out Systems

Figure 7.3 defines the auxiliary functions used in the equations, shown in Figure 7.2, that define the meaning function $\mathcal{M}$ for systems and finite rolled-out systems;[1] the meaning function $\mathcal{M}^{\infty}$ for infinite rolled-out system is given at the end of this section. $\mathcal{M}$'s arguments are a list of statements; a procedure environment, which maps a procedure name to the procedure's formal parameter and body (denoted by *Penv* through-out this chapter); a (calling-)context, which is a sequence of call-site tags[2]; a state, which is a mapping from variables to values; and a bucket function, which is explained below. Suppose that *SL* is the list of statement that make up the main procedure of a system or finite rolled-out system. The result of $\mathcal{M}$ applied to *SL*, *Penv*, the empty context (*i.e.* <>), an initial state and the initial bucket function (defined below), is a final state and a final bucket function.

A bucket function maps a context and a program point to the sequence of values computed at the program point in the context. (For single-procedure programs all contexts are empty; hence, the bucket function degenerates into a mapping from program points to sequences of values.) Informally, a bucket function can be thought of as a two dimensional array of buckets, where rows are labeled by contexts, columns by program points, and each bucket contains a sequence of values. As a formal device, this array contains too many entries. For example, if point p is in procedure P and context C is $<Call_Q>$ then the array has an entry for the pair (C, p), but this entry cannot be assigned any values because p cannot execute in context C (only statements from procedure Q can execute in context C). Thus, for a given system S, we are only interested in those context-point pairs that may be associated with a sequence of values. These pairs are captured in the following definition.

DEFINITION (Legal Context-Point Pairs). For system S, context-point pair (C, p) is *legal* if

(1) For all sub-sequences t_i, t_{i+1} of C the call-site with tag t_i calls the procedure containing the call-site with tag t_{i+1}, and

(2) point p can execute in the context given by C (*e.g.*, if the last call-site in C is "call $P(a)$" then p must be in procedure P).

---

[1] The function $\mathcal{M}$ does not define the values computed at call-site and scope, entry, formal-in, and formal-out points because these values can be determined from the values computed at the following points: the enclosing predicate, the set of corresponding call-site points, the set of corresponding actual-in points, and the set of corresponding actual-out points, respectively.

[2] As discussed in Chapters 2 and 5, each program component has a unique tag. While tags are necessary for identification of corresponding components in integration, they are not required for the results in this chapter. All that is required here is that a unique identifier be associated with each program point; the point's tag is a convenient choice.

$\mathcal{M}$: statement → procedure-environment → context → state$_\perp$ → bucket-function$_\perp$ → (state$_\perp$ × bucket-function$_\perp$)

$\mathcal{M}[\![s1; s2]\!]\ Penv\ C\ \sigma\ \beta =$
  let
    $(\sigma_1, \beta_1) = \mathcal{M}[\![s1]\!]\ Penv\ C\ \sigma\ \beta$
  in
    $\mathcal{M}[\![s2]\!]\ Penv\ C\ \sigma_1\ \beta_1$
  ni

$\mathcal{M}[\![i := exp]\!]\ Penv\ C\ \sigma\ \beta =$
  let
    $v = \mathcal{E}[\![exp]\!]\ \sigma$
    $b = (C, point(i := exp))$
  in
    $(\sigma[i/v], update\ \beta\ b\ v)$
  ni

$\mathcal{M}[\![$**if** *expression* **then** *s1* **else** *s2*$]\!]\ Penv\ C\ \sigma\ \beta = \cdots$

$\mathcal{M}[\![$**while** *expression* **do** *s1*$]\!]\ Penv\ C\ \sigma\ \beta = \cdots$

$\mathcal{M}[\![$**call** $P(a)]\!]\ Penv\ C\ \sigma\ \beta =$
  let
    $f,\ body = Penv\ [\![P]\!]$
    $b_1 = (C, point(a\text{-}before))$
    $b_2 = (C, point(a\text{-}after))$
    $C^+ = C \mid point(Call_P)$
    $\beta_1 = update\ \beta\ b_1\ (\mathcal{E}[\![a]\!]\ \sigma)$
    $\sigma_1 = zerostate\ [f/\mathcal{E}[\![a]\!]\ \sigma]$
  in
    let
      $\sigma_2, \beta_2 = \mathcal{M}[\![body]\!]\ Penv\ C^+\ \sigma_1\ \beta_1$
    in
      $(\sigma[a/\mathcal{E}[\![f]\!]\ \sigma_2], update\ \beta_2\ b_2\ (\mathcal{E}[\![f]\!]\ \sigma_2))$
    ni
  ni

$\mathcal{M}[\![$**scope** $P(f := a; a := f)\ body]\!]\ Penv\ C\ \sigma\ \beta =$
  let
    $b_1 = (C, point([\![f := a]\!]))$
    $b_2 = (C, point([\![a := f]\!]))$
    $\beta_1 = update\ \beta\ b_1\ (\mathcal{E}[\![a]\!]\ \sigma)$
    $\sigma_1 = zerostate\ [f/\mathcal{E}[\![a]\!]\ \sigma]$
  in
    let
      $\sigma_2, \beta_2 = \mathcal{M}[\![body]\!]\ Penv\ C\ \sigma_1\ \beta_1$
    in
      $(\sigma[a/\mathcal{E}[\![f]\!]\ \sigma_2], update\ \beta_2\ b_2\ (\mathcal{E}[\![f]\!]\ \sigma_2))$
    ni
  ni

$\mathcal{E}$: expression → state → value$_\perp$

$\mathcal{E}[\![exp]\!]\ \sigma = \cdots$

Figure 7.2. $\mathcal{M}$ is the meaning function for finite statement lists. The definitions for **if** and **while** statements are left unspecified; they differ from a standard definition (see [Gordon79]) only in the updating of a bucket function. $\mathcal{E}$, the semantic function for expressions, is also left unspecified; informally, given an expression *exp* and a state σ the result of $\mathcal{E}[\![exp]\!]\sigma$ is either the value of the expression or $\perp$ if an error occurs. Finally, the above functions are assumed to all be strict: if one of their arguments is $\perp$ then they produce $\perp$ as a result. (Formally the function $\mathcal{M}$ produces $\perp_{\mathcal{M}}$ (the undefined element for the domain of program meanings, which is defined as $(\lambda v.\perp_{sequence}, \lambda C.\lambda p.\perp_{sequence})$; however, for the sake of clarity, we omit subscripts from $\perp$ in the remainder of this chapter.)

Auxiliary Functions		
function	type	definition
| (infix)	$list \to element \to list$	$list \mid element = append(list, [element])$
[/]	$(a \to b) \to a \to b \to (a \to b)$	$x[y/z] = \lambda w.$ if $w = y$ then z else $x\ w$
,	$a \to b \to a \times b$	$a, b = (a, b)$
Penv	*procedure-name* $\to$ *formal-parameter* $\times$ *body*	procedure environment
zerostate	*variable* $\to$ *value*	$\lambda v.\, 0$
point	*program-component* $\to$ *program-point*	(see below)
update	*bucket-functions* $\to$ *bucket* $\to v \to$ *bucket-function*	$update\ \beta\ b\ v = \beta[b/(\beta\ b) \mid v]$

Figure 7.3. Auxiliary functions for the semantic equations shown in Figure 7.2. The function *point* returns the program-point (in most cases the tag) used to identify the program component (statement) that is its argument (for actual parameters the suffixes "-before" and "-after" are used to identify the point for an actual parameter before and after the call, respectively—these correspond to the actual-in and actual-out vertices for the parameter, respectively). The function *update* appends the new value v to the current contents of bucket b.

In this chapter, β, β', etc., denote bucket functions, $\beta_{initial}$ denotes the initial bucket function that maps every context-point pair to the empty sequence (*i.e.* $\beta_{initial} \triangleq \lambda C.\lambda p.\langle\rangle$). To be complete, the bucket functions used in the chapter should be "wrapped" in a function that checks, for a given system S, that the arguments to the function are legal (*i.e.*, the function

$$check\text{-}legality(\beta, S, C, p) \triangleq \textbf{if } legal(S, C, p) \textbf{ then } \beta(C, p) \textbf{ else } \bot \textbf{ fi}).$$

However, since we only apply bucket functions to legal context-point pairs, this "wrapper" function is omited to simplify the presentation.

Example. Figure 7.4 shows a system S and the final bucket function obtained when $\mathcal{M}$ is applied to (the statement list of) the main procedure of S, the context $\langle\rangle$, any initial state, and $\beta_{initial}$.

tags[†]	System S
	procedure *Main*
1	$i := 1$
2 4	**call** P(i)
	end
5 7	**procedure** $P(x)$
8	$x := x+2$
9	**if** $x < 3$ **then**
10 12	**call** $P(x)$
	fi
	return

Final bucket function for S

	points		
context	1	2	4
$\langle\rangle$[‡]	<1>	<1>	<3>

	points			
context	8	9	10	12
$\langle tag\ (Call_P)\rangle$	<3>	<F>	$\langle\rangle$	$\langle\rangle$

[†] The tags for actual-in and actual-out vertices, and formal-in and formal-out vertices appear on the same line as the corresponding call statement or procedure declaration, respectively. The tags for call-site and entry vertices are not shown; they are 3, for the call-site in *Main*, 11, for the call-site in P, and 6, for P's enter vertex.

[‡] $\langle\rangle$ denotes both the empty context (*i.e.* the empty sequence of call-site tags) and the empty sequence of values, which is computed by a statement that is not executed in a given context.

Figure 7.4. An example system and its final bucket function.

A Meaning Function for Infinite Rolled-out Systems

The meaning of an infinite rolled-out system, given by the function $\mathcal{M}^\infty$, is the least upper bound of the meanings of all finite approximations to the infinite rolled-out system. These approximations are obtained using a pruning operation that removes from a program all statements having depth greater than k (*i.e.*, all statements nested within more than k scope statements). Furthermore, an *abort* statement is added to all scope statements at depth k (those whose bodies have been removed); executing an abort statement causes a program to terminate abnormally.

DEFINITION. (Pruned Programs) The operation *prune* applied to a program *roll-out*(S) and a integer k, denoted by $prune_k(S)$, is *roll-out*(S) with all statements nested within more than k scope statements replaced by an *abort* statement.

The least upper bound that defines $\mathcal{M}^\infty$ uses the following least upper bounds.

DEFINITION. (Least Upper Bounds).

Type	Least Upper Bound
simple values	$v_1 \sqcup v_2 \triangleq v_1$ if $v_1 = v_2$ or $v_2 = \perp$, v_2 if $v_1 = \perp$ and is undefined otherwise.[3]
sequences of values	$s_1 \sqcup s_2 \triangleq s_1$ if $s_1 = s_2$ or $s_2 = \perp$, s_2 if $s_1 = \perp$ and is undefined otherwise.[3]
states	$(\sigma_1 \sqcup \sigma_2)(x) \triangleq \sigma_1(x) \sqcup \sigma_2(x)$
bucket functions	$(\beta_1 \sqcup \beta_2)(C, p) \triangleq \beta_1(C, p) \sqcup \beta_2(C, p)$.
pairs	$(p_1, p_2) \sqcup (q_1, q_2) \triangleq (p_1 \sqcup q_1, p_2 \sqcup q_2)$

DEFINITION. (Meaning of an Infinite Rolled-out System).

$$\mathcal{M}^\infty[\![\textit{roll-out}(S)]\!]\ \textit{Penv} <> \sigma\ \beta_{\textit{initial}} \triangleq \mathcal{M}^\infty[\![\textit{roll-out}(S, \textit{Main})]\!]\ \textit{Penv} <> \sigma\ \beta_{\textit{initial}}$$

and

$$\mathcal{M}^\infty[\![\textit{roll-out}(S, \textit{Main})]\!]\ \textit{Penv} <> \sigma\ \beta_{\textit{initial}} \triangleq \bigsqcup_{i=0}^{\infty} \mathcal{M}[\![\textit{prune}_i(S)]\!]\ \textit{Penv} <> \sigma\ \beta_{\textit{initial}}.$$

Example. For system S shown in Figure 7.4, Figure 7.5 shows (the main programs of) $prune_0(S)$ and $prune_1(S)$, and (part of) the infinite program *roll-out*(S, *Main*). Let β_i denotes the final bucket function produced $prune_i(S)$. The figure also shows β_0, β_1, and $\bigsqcup_{i=0}^{\infty} \beta_i$ ($\bigsqcup_{i=0}^{\infty} \beta_i$ is the final bucket function for $\mathcal{M}^\infty[\![\textit{roll-out}(S)]\!]$, which can be simplified to β_1 since $\beta_i = \beta_1$ for all $i \geq 1$ and β_0 is everywhere undefined).

7.2. Call-site Expansion is a Semantics-Preserving Transformation

The Expansion Lemma, stated below, shows that call-site expansion is a semantics-preserving transformation: *i.e.* it shows that call-site expansion does not alter the sequences of values computed by a system. The only effect an expansion has on the computation of a system is to change the context-point pair that maps to a particular sequences of values (thought of as an array of buckets, the affect of an expansion is to delete the row for the context associated with the expanded call-site and to create new columns for the points of the scope added to the main procedure; the expansion also relabels those buckets containing the expanded call-site in their context). The preservation of sequences of values (semantics), modulo this change in the context-point pair associated with a given sequence of values, is captured by the notion of a *bucket-function congruence* or simply a *congruence*.

[3] Because values and sequences of values each form a flat domain (*i.e.* $s_1 \sqsubseteq s_2$ iff $s_1 = s_2$ or $s_1 = \perp$) and if $\mathcal{M}[\![\textit{prune}_i(S)]\!]$ is not $\perp$ then, for all $j \geq i$, $\mathcal{M}[\![\textit{prune}_j(S)]\!] = \mathcal{M}[\![\textit{prune}_i(S)]\!]$ and, for all $j < i$, $\mathcal{M}[\![\textit{prune}_j(S)]\!] = \mathcal{M}[\![\textit{prune}_i(S)]\!]$ or $\perp$, the least upper bounds used in this chapter are never undefined.

tags †	Programs *prune*(*S*, 0)	*prune*(*S*, 1)	*roll-out*(*S*, *Main*)
	procedure *Main*	**procedure** *Main*	**procedure** *Main*
1	$i := 1$	$i := 1$	$i := 1$
2:5 4:7	**scope** $(x := i ; i := x)$	**scope** $(x := i ; i := x)$	**scope** $(x := i ; i := x)$
3:8	**abort**	$x := x+2$	$x := x+2$
3:9	**epocs**	**if** $x < 3$ **then**	**if** $x < 3$ **then**
3:10 3:12	**end**	**scope** $(x := x ; x := x)$	**scope** $(x := x ; x := x)$
		abort	...
		epocs	**epocs**
		fi	**fi**
		epocs	**epocs**
		end	**end**

† The tags for transfer-in and transfer-out vertices appear on the same line as the corresponding scope statement. The tags for scope vertices are not shown; they are 3:6, for the singly nested scope in *prune*(*S*, 0), *prune*(*S*, 1), and *roll-out*(*S*, *Main*), and 3:11:6, for the double nested scope in *prune*(*S*, 1) and *roll-out*(*S*, *Main*).

The final bucket functions for *prune*(*S*, 0), *prune*(*S*, 1), and *roll-out*(*S*, *Main*) (all contexts are <>).

Program	points						
	1	2:5	4:7	3:8	3:9	3:10	3:12
prune(*S*, 0)	⊥	⊥	⊥				
prune(*S*, 1)	<1>	<1>	<3>	<3>	<F>	<>	<>
roll-out(*S*, *Main*)	<1>	<1>	<3>	<3>	<F>	<>	<>

Figure 7.5. Three programs derived from system *S* of Figure 7.4 and their final bucket functions. The program are *prune*(*S*, 0), *prune*(*S*, 1), and *roll-out*(*S*, *Main*) (only the main programs are shown).

DEFINITION. (Congruence). Bucket functions β_1 and β_2 are *congruent* iff there exists a bijection ψ such that for all context-point pairs (C, p), $\beta_1(C, p) = \beta_2(\psi(C, p))$.

The Expansion Lemma also makes use of the following definition of a compound tag.

DEFINITION. (Compound Tag). Tag $t_1{:}t_2$ denotes the *compound* tag created from tags t_1 and t_2.

Compound tags are introduced when call statement "**call** $P(a)$" is expanded as follows (assume P has formal parameter f):

(1) The new scope statement is given the compound tag $tag(Call_P){:}tag(Enter_P)$.
(2) The transfer-in statement "$f := a$" is given the compound tag $tag(Call_P.f_{in}){:}tag(Enter_P.f_{in})$.
(3) The transfer-out statement "$a := f$" is given the compound tag $tag(Call_P.f_{out}){:}tag(Enter_P.f_{out})$.
(4) The copy of statement s from the body of P is given the compound tag $tag(Call_P){:}tag(s)$.

For the example shown in Figure 7.5, the occurrence of the statement "$x := x+2$" in the main procedure of $prune_1(S)$, which was introduced by the expansion of call-site $Call_P$ ("**call** $P(i)$") in the main program of system S, is given the compound tag $tag(Call_P){:}tag(\text{"}x := x+2\text{"})$, *i.e.*, 3:8.

LEMMA. (EXPANSION LEMMA). *Let S' be the result of expanding call-site "**call** $P(a)$" from the main procedure of system S, Penv the procedure environment for S, and ψ the bijection between the context-point pairs of S and S' given by the following table (where P is assumed to have formal parameter f and because $Call_P$ is in the main procedure and there are no recursive calls on the main procedure, it must be the first call-site of any context in which it appears):*

ψ Table $\psi(C,p)=(C',p')$				
C	p	C'	p'	
$tag\,(Call_P) \notin C$	$p \notin P$	C	p	
$tag\,(Call_P) \notin C$	$p \in P$	C	p	
$<tag\,(Call_P), t_1, \cdots, t_n>$	$p \notin P$	$<tag\,(Call_P):t_1, \cdots, t_n>$	p	
$<tag\,(Call_P), t_1, \cdots, t_n>$	$p \in P$	$<tag\,(Call_P):t_1, \cdots, t_n>$	p	($t_1, \cdots, t_n$ *is not empty*)
$<tag\,(Call_P)>$	$p \in P$	$<>$	$\hat{p}^{\dagger}$	($t_1, \cdots, t_n$ *is empty*)

$^{\dagger}\hat{p}$ *denotes a point associated with the new scope; the tag of* $\hat{p}$ *is compound tag determined by the expansion of* $Call_P$.

Table 7.1

Assume that

1) σ *is a state,*
2) β *and* β' *are congruent, and*
3) *s is a statement and C a context from S, and s′ is a statement and C′ a context from S′ such that* $\psi(C, point(s))=(C', point(s'))$ *(if more than one point is associated with s, as in the case of the two points (before and after) for a call statement or the n points of a statement list, then the equivalence (given by* ψ*) must hold for corresponding points of s and s′).*

If *s terminates when initiated on* σ *then s′ terminates when initiated on* σ *and* $\mathcal{M}[\![s]\!]Penv\ C\ \sigma\ \beta$ *and* $\mathcal{M}[\![s']\!]Penv\ C'\ \sigma\ \beta'$ *yield the same final state and congruent final bucket functions.*

PROOF. The proof is an induction over the number of applications of $\mathcal{M}$ used to derive the final meaning of s (operationally, this is approximately an induction over the number of intermediate states entered into by the computation). The base case is trivial: if zero applications of $\mathcal{M}$ are used then s and s' are empty statement lists, in which case s' terminates and σ, β, and β' are unchanged; therefore $\mathcal{M}[\![s]\!]Penv\ C\ \sigma\ \beta$ and $\mathcal{M}[\![s']\!]Penv\ C'\ \sigma\ \beta'$ yield the same final state (σ) and congruent final bucket functions (β and β').

Proving the inductive step requires showing that if the lemma holds when k applications of $\mathcal{M}$ are required to determine the final meaning of $\mathcal{M}[\![s]\!]Penv\ C\ \sigma\ \beta$ then it holds when $k+1$ applications of $\mathcal{M}$ are required. For all but one case, s and s' are the same syntactic statement; in these cases, the inductive step is trivial. The one exception to this is when s is "**call** $P\,(a)$"—the call statement from S that was expanded to obtain S'. In this case, s', is the scope statement that replaces "**call** $P\,(a)$"; assuming procedure P has formal parameter f and body *body* this scope statement is "**scope** $P(f := a\,;a := f)$ *body*".

Given that "**call** $P\,(a)$" terminates and its meaning is obtained using $k+1$ applications of $\mathcal{M}$, we demonstrate below that "**scope** $P(f := a\,;a := f)$ *body*" also terminates and that $\mathcal{M}[\![$**call** $P\,(a)]\!]Penv\ C\ \sigma\ \beta$ and $\mathcal{M}[\![$**scope** $P(f := a\,;a := f)$ *body*$]\!]Penv\ C'\ \sigma\ \beta'$ yield the same state and congruent bucket functions. The remainder of the proof refers to the values shown in the following table (in which the contexts C and C' are empty, *i.e.*, equal to $<>$, because "**call** $P\,(a)$" is in the main procedure).

Auxiliary Variable Values (variables names are the same as in Figure 7.2)			
For $\mathcal{M}[\![\textbf{call}\,P\,(a)]\!]Penv <> \sigma\,\beta$		For $\mathcal{M}[\![\textbf{scope}\,P(f:=a\,;a:=f)\;body]\!]Penv <> \sigma\,\beta'$	
f	P's formal parameter		
body	the body of procedure P		
b_1	$(<>, point\,(a\text{-}before))$	b'_1	$(<>, point\,(f:=a))$
b_2	$(<>, point\,(a\text{-}after))$	b'_2	$(<>, point\,(a:=f))$
C^+	$<> \mid point(Call_P) = <point(Call_P)>$		
σ_1	$zerostate\,[f/\mathcal{E}[\![a]\!]\,\sigma]$	σ'_1	$zerostate\,[f/\mathcal{E}[\![a]\!]\,\sigma]$
β_1	$update\;\beta\;b_1\;(\mathcal{E}[\![a]\!]\,\sigma)$	β'_1	$update\;\beta'\;b'_1\;(\mathcal{E}[\![a]\!]\,\sigma)$
σ_2, β_2	$\mathcal{M}[\![body]\!]\;Penv\;C^+\;\sigma_1\;\beta_1$	σ'_2, β'_2	$\mathcal{M}[\![body]\!]\;Penv <>\;\sigma'_1\;\beta'_1$

The proof has three parts: (i) showing that "$f:=a$" terminates, $\sigma_1=\sigma'_1$, and β_1 and β'_1 are congruent; (ii) showing that *body* terminates, $\sigma_2=\sigma'_2$, and β_2 and β'_2 are congruent; and (iii) showing that "$a:=f$" terminates, the final states are equal, and the final bucket functions are congruent.

(i) To begin with, since **call** $P\,(a)$ terminates, $\mathcal{E}[\![a]\!]\,\sigma \neq \perp$; thus, the evaluation of "$f:=a$" terminates. Let $v=\mathcal{E}[\![a]\!]\,\sigma$. States σ_1 and σ'_1 are both $zerostate[f/v]$ and therefore equal. Finally, to show that β_1 and β'_1 are congruent requires showing that they map corresponding context-point pairs to the same sequence of values (*i.e.* $\beta(C,p)=\beta'\;\psi(C,p)$). By definition,

$$\beta_1 = update\;\beta\;b_1\;v\;=\beta\,[b_1/((\beta\;b_1))\mid v] = \lambda w.\;\text{if } w=b_1 \text{ then } (\beta\;b_1)\mid v \text{ else } \beta\;w\text{, and}$$
$$\beta'_1 = update\;\beta'\;b'_1\;v=\beta'[b'_1/((\beta'\;b'_1))\mid v] = \lambda w.\;\text{if } w=b'_1 \text{ then } (\beta'\;b'_1)\mid v \text{ else } \beta'\;w$$

First consider $\beta_1\;b_1$, which simplifies to $(\beta\;b_1)\mid v$, and $\beta'_1\;\psi(b_1)$, which, because assumption 3 implies $\psi(b_1)=b'_1$, simplifies to $\beta'_1\;b'_1=(\beta'\;b'_1)\mid v$. Because β and β' are congruent and $\psi(b_1)=b'_1$, $\beta\;b_1$ and $\beta'\;b'_1$ are equal (*i.e.* the same sequence of values); appending v to these sequences yields equal sequences of values. Now consider any legal context-point pair b other than b_1 and let $\psi(b)=b'$. $\beta_1\;b=\beta\;b$ and $\beta'_1\psi(b)=\beta'_1\;b'=\beta'\;b'$ (because ψ, as given by Table 7.1, is a bijection $b'\neq b'_1$). By assumption β and β' are congruent; therefore, $\beta\;b$ and $\beta'\;b'$ are equal. Finally, consider any illegal context-point pair b. Since ψ maps illegal context-point pairs in S to illegal context-point pairs in S' and, when applied to an illegal context-point pair, any bucket function produces $\perp$, β_1 and $\beta_1{}'$ yield the same sequence of values when applied to b and $\psi(b)$, respectively. Hence, β_1 and $\beta_1{}'$ are congruent, since $\beta_1 b$ and $\beta_1{}'\psi(b)$ yield the same sequence of values, for all possible context-point pairs b.

(ii) For the second step we must show that the body of the scope statement terminates on σ'_1, that $\sigma_2=\sigma'_2$, and that β_2 and β'_2 are congruent. Step (ii) follows from the inductive hypothesis because only k applications of $\mathcal{M}$ are required to derive the final meaning of $\mathcal{M}[\![body]\!]\;Penv\;C^+\;\sigma_1\;\beta_1$. The inductive hypothesis can be applied because the three assumptions of the Expansion Lemma are satisfied by $\mathcal{M}[\![body]\!]\;Penv\;C^+\;\sigma_1\;\beta_1$ and $\mathcal{M}[\![body]\!]\;Penv <>\;\sigma'_1\;\beta'_1$: Step (i) establishes that assumptions 1 and 2 are satisfied and, as shown below, assumption 3 is satisfied; therefore, because only k applications of $\mathcal{M}$ are required to derive the final meaning of $\mathcal{M}[\![body]\!]\;Penv\;C^+\;\sigma_1\;\beta_1$, the inductive hypothesis implies that the evaluation of *body* terminates, $\sigma_2=\sigma'_2$, and β_2 and β'_2 are congruent.

Assumption 3 states that for each point p in the body of procedure P, $\psi(C^+,p)$ must equal $(<>,p')$, where p' is the corresponding point in the body of "**scope** $P(f:=a\,;a:=f)\;body$". By construction, p' is assigned the compound tag $tag(Call_P):tag\,(p)$ during the expansion of "**call** $P\,(a)$". Because the last line in Table 7.1 maps (C^+,p) to $(<>,\hat{p})$ where the tag of $\hat{p}$ is $tag(Call_P):tag\,(p)$, $\psi(C^+,p)$ equals $(<>,p')$ (recall that $C^+=<tag\,(Call_P)>$); thus, assumption 3 is satisfied.

(iii) For the final step, we must show that "$a := f$" terminates, that the final states are equal, and that the final bucket functions are congruent. First, since **call** $P(a)$ terminates, $\mathcal{E}[\![f]\!]\ \sigma_2 \neq \perp$; thus, the evaluation of "$a := f$" terminates. Let $v = \mathcal{E}[\![f]\!]\ \sigma_2$. The final states are both $\sigma[a/v]$ and are therefore equal; the final bucket functions are *update* $\beta_2\ b_2\ v$ and *update* $\beta_2'\ b_2'\ v$. The argument that these two are congruent is similar to the argument that the updates that produced β_1 and β_1' are congruent (see Part (i)).

□

The Expansion Theorem extends the Expansion Lemma to non-terminating executions of a system.

THEOREM. (EXPANSION THEOREM). *Let Penv be the procedure environment for S. If S′ is the result of expanding a call-site in the main procedure of S then, for any state* σ, $\mathcal{M}[\![S]\!]\ Penv <> \sigma\ \beta_{initial}$ *and* $\mathcal{M}[\![S']\!]\ Penv <> \sigma\ \beta_{initial}$ *yield the same final state and congruent final bucket functions.*

PROOF. Let *body* be the body of the main procedure of S, *body′* the body of the main procedure of S'. The proof breaks down into two cases: one where S terminates when run on σ and one where S fails to terminate when run on σ. To begin with, if S terminates when run on σ then, by the Expansion Lemma, $\mathcal{M}[\![body]\!]\ Penv <> \sigma\ \beta_{initial}$ and $\mathcal{M}[\![body']\!]\ Penv <> \sigma\ \beta_{initial}$ yield the same state and congruent bucket functions.

If S fails to terminate when run on σ but S' does not then an induction similar to the one in the Expansion Lemma would imply that S also terminates when run on σ, which contradicts the assumption that S fails to terminate when run on σ. Thus, for a state σ on which S fails to terminate, S' must also fail to terminate when run on σ. This implies $\mathcal{M}[\![S]\!]\ Penv <> \sigma\ \beta_{initial}$ and $\mathcal{M}[\![S']\!]\ Penv <> \sigma\ \beta_{initial}$ both equal $(\lambda v.\perp, \lambda C.\lambda p.\perp)$ and therefore yield the same state and congruent bucket functions. □

COROLLARY. *If S′ is the result of expanding a series of call-sites from S (where each call-site is in the main procedure after the preceding expansions) then, for any state* σ, $\mathcal{M}[\![S]\!]\ Penv <> \sigma\ \beta_{initial}$ *and* $\mathcal{M}[\![S']\!]\ Penv <> \sigma\ \beta_{initial}$ *yield the same final state and congruent final bucket functions.*

PROOF. The Expansion Theorem proves both the base case and inductive step of an induction over the number of call-sites expanded to produce S' from S. □

7.3. Roll-out is a Semantics-Preserving Transformation

The Expansion Theorem demonstrates that call-site expansion is a semantics-preserving transformation; the Consistent Semantics Theorem, stated below, extends this to roll-out. The proof makes use of a partial or *limited* roll-out in which only some of the call-sites in system are expanded. This is formalized in the following definition.

DEFINITION. (Limited Roll-out). The *i-limited-roll-out* of system S to depth i, denoted by $\bar{S}_i$, is obtained from S by (repeatedly) expanding call statements having depth $\leq i$ (*i.e.*, in $\bar{S}_i$ there are no call statements having depth $\leq i$).

$\bar{S}_i$ is related to the pruning of S at depth i (*i.e.*, $prune_i(S)$) in two ways. First, syntactically $\bar{S}_i$ and $prune_i(S)$ are identical except where the scopes that contain abort statements in $prune_i(S)$ are replaced in $\bar{S}_i$ by scopes that contain call statements. Second, this implies semantically that $\bar{S}_i$ and $prune_i(S)$ compute identical sequences of values provided $prune_i(S)$ does not execute an abort statement.

THEOREM. (CONSISTENT SEMANTICS THEOREM). *Roll-out is a semantics-preserving transformation:*

if

$$\sigma_1, \beta_1 = \mathcal{M}[\![S]\!]Penv <> \sigma\ \beta_{initial} \text{ and}$$
$$\sigma_2, \beta_2 = \mathcal{M}^{\infty}[\![roll\text{-}out(S)]\!]Penv <> \sigma\ \beta_{initial}$$

then $\sigma_1 = \sigma_2$ *and* β_1 *and* β_2 *are congruent.*

PROOF. First the finite case. If *roll-out*(S) is finite then, for a sufficiently large value of i, *roll-out*(S) and $\bar{S}_i$ are syntactically identical programs. This implies that $\mathcal{M}[\![roll\text{-}out(S)]\!]Penv <> \sigma\, \beta_{initial} = \mathcal{M}[\![\bar{S}_i]\!]Penv <> \sigma\, \beta_{initial}$; therefore, $\mathcal{M}[\![\bar{S}_i]\!]Penv <> \sigma\, \beta_{initial} = \sigma_2, \beta_2$. Because $\bar{S}_i$ is obtained from S by a series of call-site expansions the Corollary to the Expansion Theorem implies $\sigma_1 = \sigma_2$ and β_1 and β_2 are congruent. Thus, in the finite case, roll-out is a semantics-preserving transformation.

If *roll-out*(S) is infinite then consider the execution of S on σ. If S terminates when run on σ then, for a sufficiently large value of i, *prune*$_i$(S) does not execute an abort statement but *prune*$_{i-1}$(S) does (i is one larger then the maximum depth of the call stack). This has three implications; first, for all legal context-point pairs in *prune*$_i$(S) (since the only legal context in *prune*$_i$(S) is the empty context, *i.e.*, <>, this is equivalent to "for all points in *prune*$_i$(S)"),

> for all $j \geq i$, $\mathcal{M}[\![prune_j(S)]\!]\, Penv <> \sigma\, \beta_{initial} = \mathcal{M}[\![prune_i(S)]\!]\, Penv <> \sigma\, \beta_{initial}$
>
> (Note that those context-point pairs legal in *prune*$_j$(S) but not *prune*$_i$(S) produce the empty sequence because no statements having depth greater than i are executed.)

Second,

> for all $j < i$, $\mathcal{M}[\![prune_j(S)]\!]\, Penv <> \sigma\, \beta_{initial} = \perp$.

And third,

$$\mathcal{M}[\![prune_i(S)]\!]\, Penv <> \sigma\, \beta_{initial} = \mathcal{M}[\![\bar{S}_i]\!]\, Penv <> \sigma\, \beta_{initial}.$$

Since $\mathcal{M}^{\infty}[\![roll\text{-}out(S)]\!]\, Penv <> \sigma\, \beta_{initial}$ is defined as $\bigsqcup_{j=0}^{\infty} \mathcal{M}[\![prune_j(S)]\!]\, Penv <> \sigma\, \beta_{initial}$, these three implications imply that, for all legal context-point pairs in *prune*$_i$(S),

$$\begin{aligned}\mathcal{M}^{\infty}[\![roll\text{-}out(S)]\!]\, Penv <> \sigma\, \beta_{initial} &= \bigsqcup_{j=0}^{\infty} \mathcal{M}[\![prune_j(S)]\!]\, Penv <> \sigma\, \beta_{initial} \\ &= \mathcal{M}[\![prune_i(S)]\!]\, Penv <> \sigma\, \beta_{initial} \\ &= \mathcal{M}[\![\bar{S}_i]\!]\, Penv <> \sigma\, \beta_{initial}.\end{aligned}$$

Let $\mathcal{M}[\![\bar{S}_i]\!]\, Penv <> \sigma\, \beta_{initial}$ equal $\bar{\sigma}, \bar{\beta}$ and recall that $\mathcal{M}^{\infty}[\![roll\text{-}out(S)]\!]\, Penv <> \sigma\, \beta_{initial}$ equals σ_2, β_2. The above equality establishes that σ_2 equals $\bar{\sigma}$ and that, for legal context-point pairs in *prune*$_i$(S), β and $\bar{\beta}$ are congruent. To extend this congruence to all context-point pairs in $\bar{S}_i$ and *roll-out*(S) require three observations. First, because only the statements in *prune*$_i$(S) are executed, $\bar{\beta}(\bar{b}) = \beta_2(b_2) = <>$, for legal context-point pairs $\bar{b}$ from $\bar{S}_i$ and b_2 from *roll-out*(S) that are not legal in *prune*$_i$(S). Second, for all illegal context-point pairs, β and $\bar{\beta}$ produce $\perp$. Third, the composition of the isomorphisms given by Table 7.1 for the remaining expansions required to obtain *roll-out*(S) from $\bar{S}_i$ produces an isomorphism that maps legal context-point pairs from $\bar{S}_i$ that are not legal in *prune*$_i$(S) to legal context-point pairs from *roll-out*(S) that are not legal in *prune*$_i$(S) and illegal context-point pairs in $\bar{S}_i$ to illegal context-point pairs in *roll-out*(S). Together these observations extend the congruence between β and $\bar{\beta}$ for legal context-point pairs in *prune*$_i$(S) to all context-point pairs; thus, β and $\bar{\beta}$ are congruent.

Finally, as in the finite case, the Corollary to the Expansion Lemma implies that $\sigma_1 = \bar{\sigma}$ and that β_1 and $\bar{\beta}$ are congruent bucket functions. Thus, $\sigma_1 = \sigma_2$ because they both equal $\bar{\sigma}$, and β_1 and β_2 are congruent because they are both congruent to $\bar{\beta}$ (the isomorphism between the context-point pairs of β_1 and β_2 is the composition of the isomorphism for β_1 and $\bar{\beta}$ and the isomorphism for β_2 and $\bar{\beta}$, which is itself an isomorphism). Therefore, in the case where S terminates on σ, roll-out is a semantics-preserving transformation.

Finally, if S fails to terminate when run on σ then the meaning of S is the everywhere-undefined state and the everywhere-undefined bucket function (*i.e.* $\sigma_1 = \lambda v.\perp$ and $\beta_1 = \lambda C.\lambda p.\perp$). For all values i, this is also the meaning of *prune*$_i$(S) because *prune*$_i$(S) either executes an abort statement or enters into the same infinite loop as S. Therefore, $\sigma_2, \beta_2 = \mathcal{M}^{\infty}[\![roll\text{-}out(S)]\!]\, Penv <> \sigma\, \beta_{initial} =$

$\bigsqcup_{i=0}^{\infty} \mathcal{M}[\![prune_i(S)]\!]\, Penv <> \sigma\, \beta_{initial} = (\lambda v.\bot, \lambda C.\lambda p.\bot)$ and consequently, $\sigma_1 = \sigma_2$ and β_1 and β_2 are congruent. Thus, in all cases, roll-out is a semantics-preserving transformation. □

CHAPTER 8

Integrate$^{\infty}$ SATISFIES THE REQUIREMENTS ON I^{∞}

This chapter proves the Infinite Integration Theorem, which demonstrates that *Integrate*$^{\infty}$ satisfies the requirements on I^{∞} from Version 2 of the Revised Model of Program Integration given in Section 4.3. In terms of the commutative square shown in Figure 8.1, the Infinite Integration Theorem deals with the bottom line labeled "Conceptual:". That is, it proves that M^{∞}—the result of applying *Integrate*$^{\infty}$ to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*)—captures the changed and preserved behavior of *roll-out*(*A*) and *roll-out*(*B*) with respect to *roll-out*(*Base*).

This theorem is used to establish that M—the result of applying *Integrate*^{S} to *Base*, *A*, and *B*—captures the changed and preserved behavior of *A* and *B* with respect to *Base* as follows. First, because as shown in Chapter 6, M^{∞} and *roll-out*(*M*) are syntactically equivalent, the Infinite Integration Theorem implies that *roll-out*(*M*) captures the changed and preserved behavior of *roll-out*(*A*) and *roll-out*(*B*) with respect to *roll-out*(*Base*). Finally, when combined with the result from Chapter 7, which states that roll-out is a semantics-preserving transformation, this implies that *M* captures the changed and preserved behavior of *A* and *B* with respect to *Base*.

The proof of the Infinite Integration Theorem is a reduction to a previously known result: the (Finite) Integration Theorem [Yang90], which proves that *Integrate*^{hpr} (the integration algorithm discussed in Chapter 2) satisfies the integration model from Section 1.1 for finite single-procedure programs. This

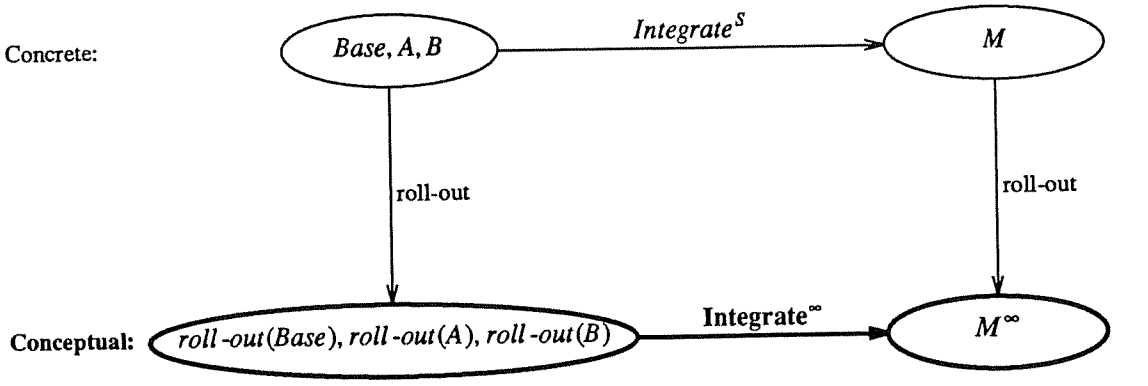


Figure 8.1. The commutative square that captures Version 2 of the Revised Model for Multi-Procedure Integration. This chapter deals with the bottom line of the figure, which is highlighted in bold.

reduction is possible because the requirements on I^∞ apply only to initial states σ for which *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) terminate. For such states, there exists an integer k such that $prune_k$(*Base*), $prune_k$(*A*), and $prune_k$(*B*) are semantically equivalent to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*), respectively. Let M_k be the program produced by $Integrate^{hpr}(prune_k(A), prune_k(Base), prune_k(B))$. Our goal in this chapter is to prove that M^∞ captures the changed and preserved behavior of *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) by proving that M^∞ is semantically equivalent to M_k, which by the Finite Integrate Theorem, captures the changed and preserved behavior of $prune_k$(*Base*), $prune_k$(*A*), and $prune_k$(*B*).

Unfortunately, there are two problems with the above argument. Both of these arise because $prune_k$ does not capture syntactic properties of *roll-out*(*A*) and *roll-out*(*B*) relevant to integration. The first problem, which we call the *severed-path problem*, is the failure of the function $prune_k$ to preserve certain paths. The second problem, which we refer to as the *cutout-slice problem*, is specific to integration. This problem arises because $prune_k(M^\infty)$ contains (pruned) slices that are not part of M_k.

In order to avoid confusion in the following discussion we formalize the *depth* of statements, vertices, and edges.

DEFINITION. (Depth). The *depth* of a statement is the number of scope statements that enclose it. The *depth* of vertex v, denoted by *depth*(v), is the depth of the statement represented by v (*i.e.* the statement labeling v). The *depth* of edge $x \longrightarrow y$ is the greater of *depth*(x) and *depth*(y). Finally, the *depth* of def-order edge $x \longrightarrow_{do(z)} y$ is the greatest of *depth*(x), *depth*(y), and *depth*(z).

Note that because transfer-in and transfer-out statements are not in the scope they parameterize, they have the same depth as the scope statement. For example, in *roll-out*(S) of Figure 8.3, the depth of the statement "$x := y+z$" is 1. The depth of all other statements—including the three transfer-in statements "$x := a$", "$y := b$", and "$z := c$", and transfer-out statement "$a := x$"—is 0.

The remainder of this chapter is organized as follows. We first solve the severed-path problem by introducing a replacement for *prune* called *cutoff*. We then solve the cutout-slice problem by *converting* the cutoffs of *roll-out*(*A*) and *roll-out*(*B*) into semantically equivalent programs that "reintroduce" cutout slices. We then use *cutoff* and *convert*, to reduce the proof of the Infinite Integration Theorem to that of the Finite Integration Theorem. Finally, in Section 8.4, we summarize how the principal results from this chapter and Chapters 6 and 7 are combined to prove that $Integrate^S$ is an acceptable multi-procedure

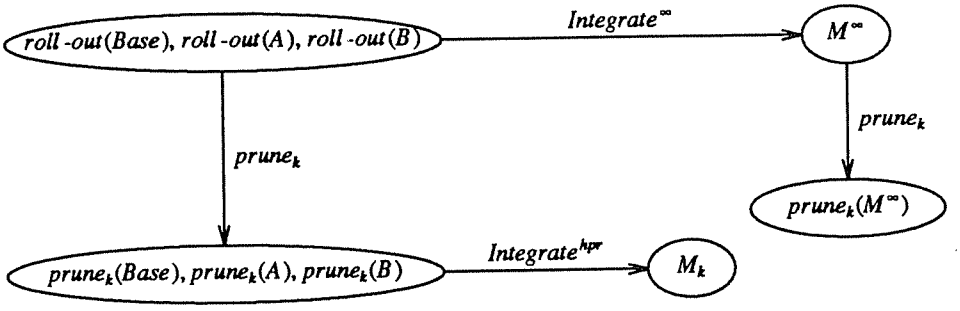


Figure 8.2. A pictorial outline for the proof of the Infinite Integration Theorem.

program-integration algorithm.

8.1.1. The Severed-Path Problem

The severed-path problem refers to the fact that paths between vertices having depth $\leq k$ can be severed in $prune_k(S)$. In particular, those paths that contain edges having depth greater than k are severed when these edges are pruned. This is illustrated in Figure 8.3 where the paths from vertices "$b := 1$" and "$c := 2$" to the vertex "$t := a$" do exist in $G_{prune_0(S)}$.[1]

To solve the severed-path problem, we introduce the operation *cutoff*, which is similar to *prune*. The major difference between *cutoff* and *prune* is that a cutoff program includes edges that splice together the paths severed in a pruned program. This splicing is done by replacing transfer-out statements having depth k with *dummy-out* statements that, by referencing variables defined by certain transfer-in statements of the same scope, induce flow dependence edges that replace pruned paths (this is illustrated in Figure 8.4). There is one special case for this replacement: if none of the transfer-in vertices are connected to a transfer-out vertex then the dummy-out vertex replacing it must not induce any edges into the cutoff program. This is accomplished by using the right-hand-side expression 0 in the dummy-out statement;

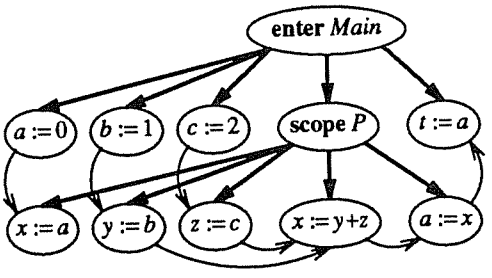
roll-out(S)

```
Procedure Main
  a := 0
  b := 1
  c := 2
  scope P(x := a, y := b, z := c ;
          a := x)
    x := y+z
  epocs
  t := a
end
```

$prune_0(S)$

```
Procedure Main
  a := 0
  b := 1
  c := 2
  scope P(x := a, y := b, z := c ;
          a := x)
    abort
  epocs
  t := a
end
```

$G_{roll\text{-}out(S)}$



$G_{prune_0(S)}$

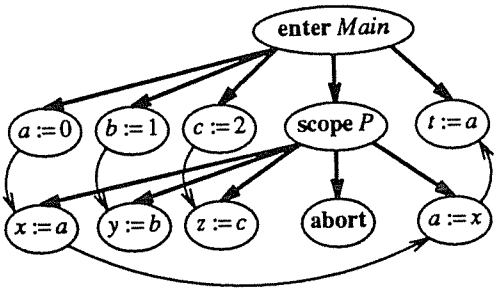


Figure 8.3. The severed-path problem. Unlike $G_{roll\text{-}out(S)}$, $G_{prune_0(S)}$ does not contain paths from the vertices labeled "$b := 1$" and "$c := 2$" to the vertex labeled "$t := a$". (In this figure, and the remaining figures in this chapter, only the main procedure of each program is shown.)

[1] Notice that in $prune_0(S)$ an unwanted path connects "$a := 0$" to "$t := a$".

because this right-hand-side references no variables, it induces to flow dependence edges and thus preserves the (lack of) paths from transfer-in vertices (this is illustrated in Figure 8.5).

The definition of cutoff also introduces *abort scopes*—the execution of an abort scope, like the execution of an abort statement, causes a program to terminate abnormally. *Cutoff* introduces abort scopes rather than using abort statements because the program M_k produced by applying $Integrate^{hpr}$ to the cutoffs of *roll-out*(*Base*), *roll-out*(A), and *roll-out*(B) must abort if it executes a scope statement at depth k. Unfortunately, when abort statements are used, the slices that make up M_k may include a scope at depth k but not the abort statement in the scope. In this case, it is impossible to prove that such scopes are not executed by a terminating execution of M_k. In contrast, abort scopes, which tie together the scope statement and the abort statement, avoid this separation; thus, using abort scopes, termination guarantees that M_k executes no scope statements having depth k.

DEFINITION. (Cutoff). The *cutoff* of *roll-out*(S) at depth k, denoted by $S!k$, contains all the statements of *roll-out*(S) that have depth $\leq k$ with the following modifications:

1) Suppose transfer-out vertex *t-out* represents a transfer-out statement "$a := \cdots$" at depth k in *roll-out*(S). Let T be the set of transfer-in statement associated with the same scopes as *t-out*. In $S!k$, *t-out* is replaced by the dummy-out statement "$a := \cdots \oplus x \oplus \cdots$", where the variables x on the right-hand-side are a function of the paths in *roll-out*(S) from elements of T to *t-out*. In particular, if the transfer-in vertex label "$x := \cdots$" is connected to *t-out* by a path of edges having depth greater than k or a single edge from the transfer-in vertex to *t-out* then x appears on the left-hand-site of the dummy-out statement. If, however, no such paths or edge exists then the statement "$a := 0$" replaces the transfer-out statement. Finally, for the purposes of integration, a dummy-out statement retains the tag of the transfer-out statement it replaces.

2) Scopes having depth k are *abort* scopes; executing an abort scope causes $S!k$ to terminate abnormally.

The actual operation denoted by $\oplus$ in a dummy-out statement is unimportant because the scope associated with a dummy-out statment is always an abort scope; thus, dummy-out statements are never executed.

Example. Figure 8.4 shows $G_{S!0}$, for system S shown in Figure 8.3. In $G_{S!0}$, the dummy-out vertex labeled "$a := y \oplus z$" replaces the transfer-out vertex labeled "$a := x$" of $G_{\text{roll-out}(S)}$. Because "$a := y \oplus z$" references the variables y and z, the dummy-out vertex is the target of flow dependence edges from the

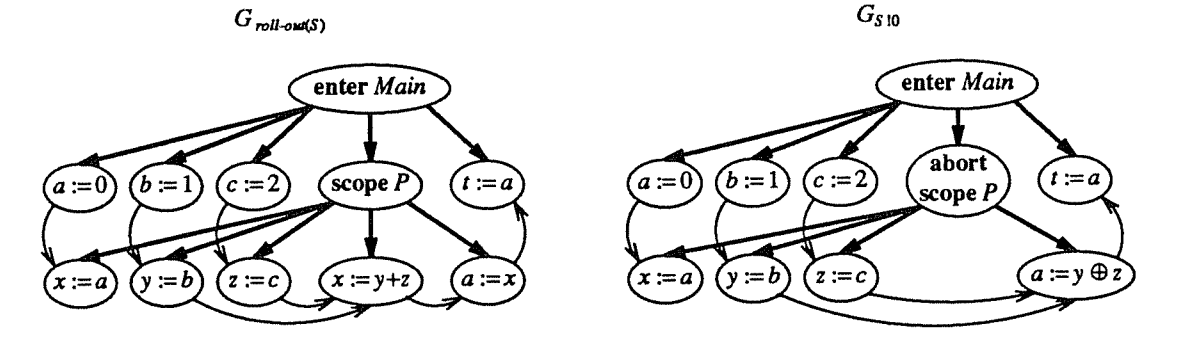


Figure 8.4. The solution to the severed-path problem applied to the example from Figure 8.3.

transfer-in vertices labeled "$y := b$" and "$z := c$". These flow dependence edges replace the paths in $G_{roll\text{-}out(S)}$ from these transfer-in vertices to the transfer-out vertex labeled "$a := x$".

8.1.2. The Cutout-Slice Problem

Since cutoff, unlike *prune*, preserves paths, one might think that M_k (the result of $Integrate^{hpr}(A\,!k, Base\,!k, B\,!k)$) would equal $M^{\infty}!k$. Unfortunately, as illustrated by Figure 8.5, this is not the case. The problem is that certain slices present in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ may not be present in $\Delta^{hpr}(A\,!k, Base\,!k)$. In effect, when cutoff removes program components having depth greater than k, it may also remove affected and directly affected points whose slices are part of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. In this case, we say the slices have been *cut out* of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Because cutout slices are not part of $\Delta^{hpr}(A\,!k, Base\,!k)$, M_k may not contain vertices that are in $M^{\infty}!k$.

There are two kinds of cutout slices. First, for an affected point *ap* having depth greater than k, the backward slice of $roll\text{-}out(A)$ taken with respect to *ap* is a cutout slice. Such a slice is part of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, but not $\Delta^{hpr}(A\,!k, Base\,!k)$ because *ap* is not in $A\,!k$. Second, for a directly affected point *dap* having depth greater than k, the forward slice of $roll\text{-}out(A)$ taken with respect to *dap* is a cutout slice. Such a slice (and backward slices taken with respect to each vertex in it) are part of $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, but not $\Delta^{hpr}(A\,!k, Base\,!k)$ because *dap* is not in $A\,!k$.

Example. The cutout-slice problem is illustrated in Figure 8.5 by the affected point v ($v \in f^{\infty}(roll\text{-}out(A), DAP1)$) and the directly affected point $DAP2$. $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ contains the statement "$a := 2$" because it is in $b^{\infty}(roll\text{-}out(A), v)$ and (the backward slice with respect to) the statement "$b := a$" because it is in $f^{\infty}(roll\text{-}out(A), DAP2)$). In contrast, because $Base\,!0$ and $A\,!0$ are identical $\Delta^{hpr}(A\,!k, Base\,!k)$ is empty (thus, the slices with respect to v and $DAP2$, which have depth greater their 0, have been cutout).

The conversion of $A\,!k$, which "reintroduces" the two kinds the cutout slices, is a two-step process (the conversion of $B\,!k$ is similar). First, for an affected point *ap* having depth greater than k, the portion of the slice $b^{\infty}(roll\text{-}out(A), ap)$ having depth $\leq k$ is reintroduced into $\Delta^{hpr}(A\,!k, Base\,!k)$ by adding a new *dummy-use* statement to certain scopes having depth k. (Dummy-use statements, which have depth $k+1$, are the only statements with depth greater than k in a converted cutoff program.) Because dummy-use statements do not exist in $Base\,!k$, they are directly affected points in the converted version of $A\,!k$; however, by construction, they have no outgoing edges and therefore contribute only a backward slice to Δ^{hpr}. This backward slice replaces the part of $b^{\infty}(roll\text{-}out(A), ap)$ having depth $\leq k$.

The second step of the conversion reintroduces slices cutout because directly affected points having depth greater than k are not part of $A\,!k$. Let *dap* be a directly affected point having depth greater than k. The portion of the slice $f^{\infty}(roll\text{-}out(A), dap)$ having depth $\leq k$ is reintroduced into $\Delta^{hpr}(A\,!k, Base\,!k)$ by assigning new tags to certain dummy-out statements. In particular, if a transfer-out vertex having depth k is in $f^{\infty}(roll\text{-}out(A), dap)$ then the dummy-out vertex that replaces it is assigned a new tag (the definition given below also treats the special case where a new tag must be assigned to the same vertex in the conversion of $A\,!k$ and $B\,!k$). This causes $Integrate^{hpr}$ to determine that the dummy-out vertex is a directly affected point and thus to take its forward slice as part of the construction of Δ^{hpr}. This forward slice replaces the part of $f^{\infty}(roll\text{-}out(A), dap)$ having depth $\leq k$.

The following definition formalizes the operation *convert*, which solves the cutout-slice problem. It is important to note that for an *a priori* fixed initial state σ on which no statements at depth greater than k are executed, we have the freedom to place additional program elements at depth $k+1$ without altering the program behavior on σ. These additional elements, however, affect the program produced by $Integrate^{hpr}$.

roll-out(Base)	*roll-out(A)*	*A !0*
Procedure *Main*	**Procedure** *Main*	**Procedure** *Main*
$a := 2$	$a := 2$	$a := 2$
scope $P(x := a;$	**scope** $P(x := a;$	**scope** $P(x := a;$
$a := x)$	$a := x)$	$a := 0)$
$t1 := 1$	$\boxed{t1 := 3}$	
$t2 := x+t1$	$t2 := x+t1$	
$x := 2$	$\boxed{x := 4}$	
epocs	**epocs**	**epocs**
$b := a$	$b := a$	$b := a$
end	**end**	**end**

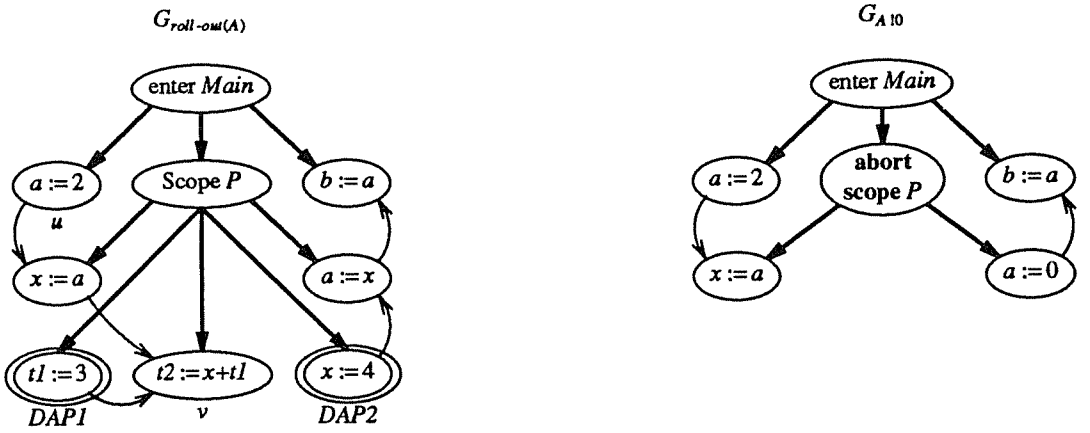


Figure 8.5. The cutout-slice problem is illustrated by the b^{∞} slice taken with respect to affected point v and the f^{∞} slice taken with respect to directly affected point *DAP2*. While both of these slices are part of Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)), neither is part of Δ^{hpr}(*A* !0, *Base* !0). (In this figure boxed statements indicate the modifications to *A* and encircled vertices denote directly affected points.)

DEFINITION (Convert). Formally the operation *convert* is applied to the five arguments $A!k$, $B!k$, *roll-out*(*A*), *roll-out*(*B*), and *roll-out*(*Base*) and produces, in tandem, the converted versions of $A!k$ and $B!k$; however, for brevity we write *convert* as a unary function that produces a single result (*e.g.*, we write *convert*($A!k$) in place of *SelectFirst*(*convert*($A!k$, $B!k$, *roll-out*(*A*), *roll-out*(*B*), *roll-out*(*Base*)))). The function *convert* syntactically modifies $A!k$ and $B!k$ as follows:

1) If a transfer-out vertex *t-out* having depth k is in the forward slice of *roll-out*(*A*) with respect to a directly affected point having depth greater than k then *d-out*, the dummy-out vertex replacing *t-out* in $A!k$, is given a new tag. This same conversion step is applied to $B!k$; however, if *d-out* is given a new tag in *convert*($A!k$) and *convert*($B!k$), and b^{∞}(*roll-out*(*A*), *t-out*) equals b^{∞}(*roll-out*(*B*), *t-out*) then *d-out* is assigned the same new tag in *convert*($A!k$) and *convert*($B!k$). (Note that, if *d-out* is given a new tag in *convert*($A!k$) and *convert*($B!k$), and b^{∞}(*roll-out*(*A*), *t-out*) $\neq b^{\infty}$(*roll-out*(*B*), *t-out*) then Type I interference exist and integration is unsuccessful. Because we are only interested in proving properties for successful integrations, this case is left undefined.)

2a) If a transfer-in vertex *t-in* having depth k is in the backward slice of *roll-out*(*A*) with respect to an affected point having depth greater than k then a new *dummy-use* statement (having depth $k+1$) is

placed in *t-in*'s scope. For the transfer-in statement "$x := \cdots$", this dummy-use statement is "$NewA := x$" where *NewA* is a new variable not otherwise used in *roll-out*(A), *roll-out*(B), *convert*$(A\,!k)$, or *convert*$(B\,!k)$ and this assignment statement is given a new tag. The same conversion step is applied to $B\,!k$.

2b) If a scope vertex having depth k is in the backward slice of *roll-out*(A) with respect to an affected point having depth greater than k and none of the transfer-in vertices of the scope are in such a slice, then the new *dummy-use* statement "$NewA := 0$" is placed in the scope (as in Case (2a), *NewA* is a variable not otherwise used in *roll-out*(A), *roll-out*(B), *convert*$(A\,!k)$, or *convert*$(B\,!k)$ and this assignment statement is given a new tag). The same conversion step is applied to $B\,!k$.

If multiple dummy-use statements are placed in a scope their order is irrelevant because they each assign to a different new variable. Also, because each dummy-use statement is assigned a new tag, the same dummy-use statement is never placed in *convert*$(A\,!k)$ and *convert*$(B\,!k)$.

Example. Figure 8.6 shows $G_{roll\text{-}out(A)}$ and $G_{convert(A\,!0)}$ for the program *roll-out*(A) shown in Figure 8.5. In this example, the conversion adds the dummy-use statement "$NewA := x$" to $A\,!0$ because the transfer-in vertex labeled "$x := a$" (which has depth 0) is in b^{∞}(*roll-out*(A), v) (v is a affected point having depth greater than 0). The conversion also assigns dummy-out vertex "$a := 0$" a new tag, because the transfer-out vertex it replaces (*i.e.*, "$a := x$") is in f^{∞}(*roll-out*(A), *DAP2*). Because of these conversions, Δ^{hpr}(*convert*$(A\,!k)$, *Base*$\,!k$) correctly includes the vertex u labeled "$a := 2$" because $u \in b^{hpr}$(*convert*$(A\,!k)$, "$NewA := A$") and (the b^{hpr} slice with respect to) the vertex labeled "$b := a$", which is an affected point because it is in f^{hpr}(*convert*$(A\,!k)$, "$a := 0$").

8.2. *Integrate*$^{\infty}$ Satisfies the Extended HPR Integration Model

This section proves *Integrate*$^{\infty}$ satisfies the HPR integration model, discussed in Section 1.1, extended to allow infinite programs. This is done by showing that whenever *Integrate*$^{\infty}$(*roll-out*(A), *roll-out*(*Base*), *roll-out*(B)) is successful in producing M^{∞}, *Integrate*^{hpr}(*convert*$(A'\,!k)$, *Base*$\,!k$, *convert*$(B'\,!k)$) is successful in producing M_k such that $M_k = M^{\infty}\,!k$.

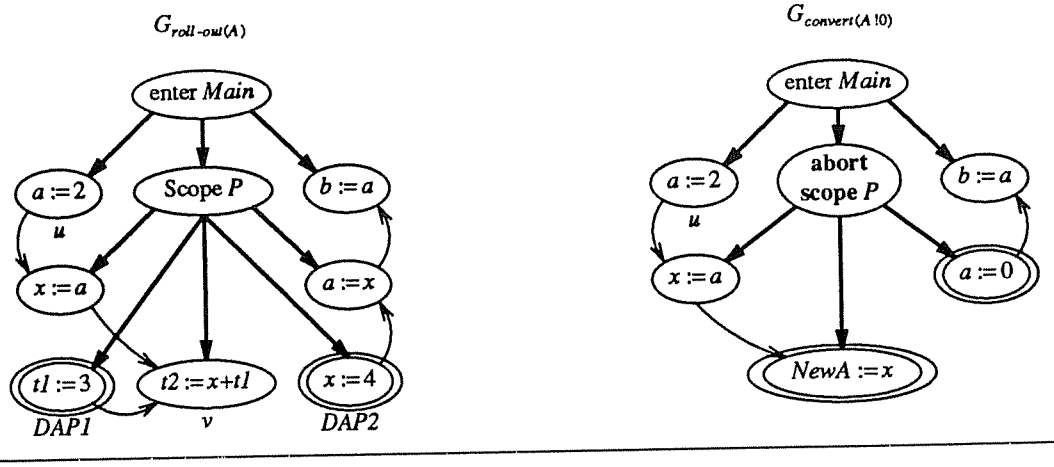


Figure 8.6. The solution to the cutout-slice problem applied to *roll-out*(A) in Figure 8.5 is illustrated in $G_{convert(A\,!0)}$. (Encircled vertices denote directly affected points.)

This reduces the proof for $Integrate^{\infty}$ to the proof for $Integrate^{hpr}$. (Thought the remainder of this section, we use the notation $A'!k$ and $B'!k$ to denote $convert(A'!k)$ and $convert(B'!k)$, respectively.)

The proof of the Infinite Integration Theorem in Section 8.2.4 is based on results from Sections 8.2.1 through 8.2.3. Section 8.2.1 proves that when restricted to vertices having depth $\leq k$, $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ equals $\Delta^{hpr}(A!k, Base!k)$; Section 8.2.2 proves the similar result for Pre^{∞} and Pre^{hpr}. Finally, two interference results are proven in Section 8.2.3. (An overview of the structure of the entire proof is provided in Figure 8.7, where the numbers indicate the order in which the theorems and lemmas are proven.)

8.2.1. The Δ Equivalence Lemma

The Δ Equivalence Lemma proves that $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $\Delta^{hpr}(A'!k, Base!k)$ contain the same vertices having depth $\leq k$. The proof is based on two supporting results: the Path Equivalence Lemma and the DAP Equivalence Lemma. The Path Equivalence Lemma demonstrates that vertices having depth $\leq k$ are connected by a path in $G_{roll\text{-}out(A)}$ iff they are connected by a path in $G_{A'!k}$. The DAP Equivalence Lemma demonstrates that $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ equals $DAP^{hpr}(A'!k, Base!k)$, when restricted to vertices having depth $\leq k$.

Path Equivalence Lemma

LEMMA. (PATH EQUIVALENCE LEMMA). *For all vertices x and y having depth $\leq k$ (thus excluding dummy-use vertices), $x \rightarrow^{*}_{c,f} y$ in $G_{roll\text{-}out(A)}$ iff $x \rightarrow^{*}_{c,f} y$ in $G_{A'!k}$ (if x or y is a dummy-out vertex in $G_{A'!k}$ then it is a transfer-out vertex in $G_{roll\text{-}out(A)}$).*

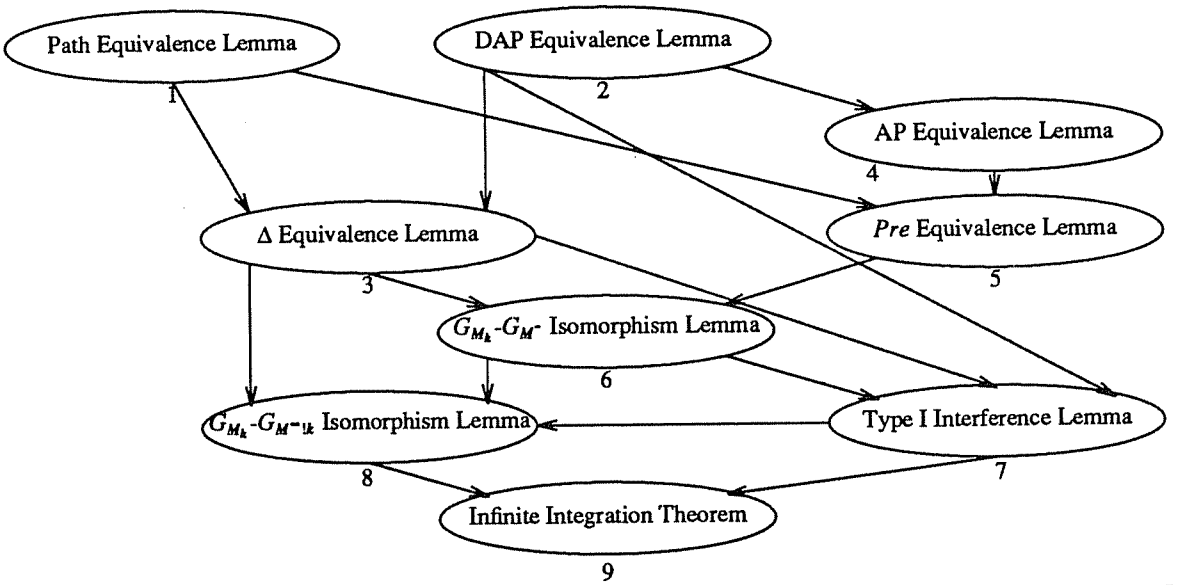


Figure 8.7. A pictorial breakdown of the proof that $Integrate^{\infty}$ satisfies the HPR model for single procedure integration. The proofs are given in the order indicated.

PROOF.

$\Rightarrow$: A path $x \rightarrow^*_{c,f} y$ in $G_{roll\text{-}out(A)}$ can be divided, at transfer-in and transfer-out vertices having depth k, into a series of sub-paths alternately containing edges having depth $\leq k$ and edges having depth $> k$. A sub-path of edges having depth $> k$ from a transfer-in vertex to a transfer-out vertex in $G_{roll\text{-}out(A)}$ is summarized in $G_{A'!k}$ by an edge from the transfer-in vertex to the dummy-out vertex that replaces the transfer-out vertex; a sub-path of edges having depth $\leq k$ in $G_{roll\text{-}out(A)}$ corresponds to a subpath in $G_{A'!k}$. (This correspondence, for each edge, is given by the identity function except for edges whose source or target is a transfer-out vertex having depth k; these edges in $G_{roll\text{-}out(A)}$ correspond to edges in $G_{A'!k}$ whose source or target, respectively, is the dummy-out vertex that replaced the transfer-out vertex.) Connecting the endpoints of these sub-paths with the endpoints of the edges from transfer-in vertices to dummy-out vertices identifies a path $x \rightarrow^*_{c,f} y$ in $G_{A'!k}$.

$\Leftarrow$: In the other direction, a path $x \rightarrow^*_{c,f} y$ in $G_{A'!k}$ can be broken into a series of sub-paths of edges having depth $\leq k$ interleaved with edges from transfer-in to dummy-out vertices. This second kind of edge in $G_{A'!k}$ exist because, in $G_{roll\text{-}out(A)}$, paths of edges having depth greater than k connect the transfer-in vertices to the transfer-out vertices replaced by the dummy-out vertices; as is the previous case, each sub-path of edges having depth $\leq k$ in $G_{A'!k}$ corresponds to a sub-path of edges having depth $\leq k$ in $G_{roll\text{-}out(A)}$. The concatenation of these alternating sub-paths produces a path $x \rightarrow^*_{c,f} y$ in $G_{roll\text{-}out(A)}$. □

DAP Equivalence Lemma

LEMMA. (DAP EQUIVALENCE LEMMA). *Restricted to vertices having depth $\leq k$, and excluding dummy-out vertices and the transfer-out vertices to which they correspond, $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ equals $DAP^{hpr}(A'!k, Base!k)$.*

PROOF. The proof is by mutual containment: it shows that, when restricted to non-dummy-out vertices having depth $\leq k$, $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq DAP^{hpr}(A'!k, Base!k)$ and $DAP^{hpr}(A'!k, Base!) \subseteq DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. First, let u be a directly affected point from $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ having depth $\leq k$; thus, by definition, u is in $G_{roll\text{-}out(A)}$ but not $G_{roll\text{-}out(Base)}$ or u has different incoming edges in $G_{roll\text{-}out(A)}$ and $G_{roll\text{-}out(Base)}$. In the former case, because $depth(u) \leq k$, u is in $G_{A'!k}$ and, because u is not in $G_{roll\text{-}out(Base)}$, u is not in $G_{Base!k}$; hence u is in $DAP^{hpr}(A'!k, Base!k)$. In the latter case, because, $depth(u) \leq k$ and u does not correspond to a dummy-out vertex in $G_{A'!k}$, the incoming edges of u in $G_{roll\text{-}out(A)}$ and $G_{roll\text{-}out(Base)}$ are the same as the incoming edges on u in $G_{A'!k}$ and $G_{Base!k}$, respectively; therefore, u has different incoming edges in $G_{A'!k}$ and $G_{Base!k}$, which implies it is in $DAP^{hpr}(A'!k, Base!k)$.

The second half of the mutual containment argument requires showing that all non-dummy-out vertices in $DAP^{hpr}(A'!k, Base!k)$ are in $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Let u be a non-dummy-out vertex in $DAP^{hpr}(A'!k, Base!k)$; thus, by definition, u is in $G_{A'!k}$ but not $G_{Base!k}$ or u has different incoming edges in $G_{A'!k}$ and $G_{Base!k}$. In the former case, u is in $G_{roll\text{-}out(A)}$ but not $G_{roll\text{-}out(Base)}$ and therefore in $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. In the latter case, because u is not a dummy-out vertex, the incoming edges on u in $G_{A'!k}$ and $G_{Base!k}$ are the same as the incoming edges on u in $G_{roll\text{-}out(A)}$ and $G_{roll\text{-}out(Base)}$, respectively; therefore, u has different incoming edges in $G_{roll\text{-}out(A)}$ and $G_{roll\text{-}out(Base)}$, which implies it is in $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. □

Δ Equivalence Lemma

LEMMA. (Δ EQUIVALENCE LEMMA). *Restricted to vertices having depth $\leq k$ (thus excluding dummy-use vertices), $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ equals $\Delta^{hpr}(A'!k, Base!k)$.*

PROOF. The proof is by mutual containment: it shows that for vertices having depth $\leq k$,

(I) $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq \Delta^{hpr}(A'!k, Base\,!k)$ and
(II) $\Delta^{hpr}(A'!k, Base\,!k) \subseteq \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

Case I. A vertex v in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ having depth $\leq k$ is, by definition, in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and therefore, for some $u \in DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, v is in $b^{\infty}f^{\infty}(roll\text{-}out(A), u)$. This implies the existence of a vertex x and two paths, $v \rightarrow^{*}_{c,f} x$ and $u \rightarrow^{*}_{c,f} x$ in $G_{roll\text{-}out(A)}$. The remainder of the proof for Case I breaks down into two cases: $depth(x) \leq k$ and $depth(x) > k$ (for $depth(x) \leq k$, there are two subcases).

1a) $depth(x) \leq k$, $depth(u) \leq k$, and u is not a transfer-out vertex having depth k.

The Path Equivalence Lemma implies $v \rightarrow^{*}_{c,f} x$ and $u \rightarrow^{*}_{c,f} x$ exist in $G_{A'!k}$; consequently, $v \in b^{hpr}f^{hpr}(A'!k, u)$. Because the DAP Equivalence Lemma implies u is in $DAP^{hpr}(A'!k, Base\,!k)$, v is in $b^{hpr}f^{hpr}DAP^{hpr}(A'!k, Base\,!k)$, which implies v is in $\Delta^{hpr}(A'!k, Base\,!k)$.

1b) $depth(x) \leq k$ and either $depth(u) > k$ or u is a transfer-out vertex having depth k.

The path from u to x must contain at least one transfer-out vertex having depth k. Let *t-out* be one of these transfer-out vertices; let *d-out* be the dummy-out vertex that replaces *t-out* in $G_{A'!k}$. Because *t-out* is on the path from u to x, it (like x) is in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$; therefore, the conversion operation assigns *d-out* a new tag, which causes *d-out* to be in $DAP^{hpr}(A'!k, Base\,!k)$. Similar to Case (1a), the Path Equivalence Lemma implies $v \rightarrow^{*}_{c,f} x$ and *d-out* $\rightarrow^{*}_{c,f} x$ exist in $G_{A'!k}$; thus, putting these paths together with *d-out* $\in DAP^{hpr}(A'!k, Base\,!k)$, v is in $b^{hpr}f^{hpr}DAP^{hpr}(A'!k, Base\,!k)$, which implies v is in $\Delta^{hpr}(A'!k, Base\,!k)$.

2) $depth(x) > k$.

The path from v to x must contain at least one transfer-in vertex or scope vertex having depth k; let *t-in* be one of these vertices. Because, like v, *t-in* is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, a dummy-use statement "*NewA* := $\cdots$" exists in $A'!k$ such that *d-use*, the vertex representing this statement, is the target of an edge from *t-in*. Because the Path Equivalence Lemma implies the path from v to *t-in* in $G_{roll\text{-}out(A)}$ exists in $G_{A'!k}$ and an edge connects *t-in* to *d-use*, v is in $b^{hpr}(A'!k, d\text{-}use)$. Finally, by construction, *d-use* is in $DAP^{hpr}(A'!k, Base\,!k)$; therefore $v \in b^{hpr}DAP^{hpr}(A'!k, Base\,!k) \subseteq b^{hpr}f^{hpr}DAP^{hpr}(A'!k, Base\,!k)$, which implies v is in $\Delta^{hpr}(A'!k, Base\,!k)$.

Case II. The preceding half of the mutual containment argument shows that when restricted to vertices having depth $\leq k$, $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq \Delta^{hpr}(A'!k, Base\,!k)$; the second half of the argument shows that, when restricted to vertices having depth $\leq k$, $\Delta^{hpr}(A'!k, Base\,!k) \subseteq \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. By definition, if $v \in \Delta^{hpr}(A'!k, Base\,!k)$ then there exists a directly affected point u and a vertex x such that $v \in b^{hpr}(A'!k, x)$ and $x \in f^{hpr}(A'!k, u)$. The proof breaks down into three cases, based on the type of u.

1) u is a dummy-out vertex given a new tag when $A\,!k$ is converted into $A'!k$.

Let *t-out* be the transfer-out vertex in $G_{roll\text{-}out(A)}$ replaced by dummy-out vertex u. The proof follows from two observations: first, the Path Equivalence Lemma ensures us that $v \rightarrow^{*}_{c,f} x$ and *t-out* $\rightarrow^{*}_{c,f} x$ exist in $G_{roll\text{-}out(A)}$. Second, u is given a new tag because *t-out* $\in f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Since *t-out* $\rightarrow^{*}_{c,f} x$ in $G_{roll\text{-}out(A)}$, this second observation implies $x \in f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$; therefore, because $v \rightarrow^{*}_{c,f} x$ in $G_{roll\text{-}out(A)}$ implies $v \in b^{\infty}(roll\text{-}out(A), x)$, v is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that v is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

2) u is a dummy-use vertex.

The proof for this step has three parts. First, by construction, a dummy-out vertex is the target of at most one flow dependence edge from a transfer-in vertex and a control dependence edge from a scope

vertex. If u has an incoming flow dependence edge then let *t-in* be the transfer-in vertex at the source of this edge; otherwise, let *t-in* be the scope vertex at the source of the control edge. Second, since v is in $b^{hpr}f^{hpr}(A'!k, u)$, but, by definition, dummy-use vertices have no outgoing edges, v is in $b^{hpr}(A'!k, u)$; therefore, a path, which ends with the edge *t-in* $\longrightarrow u$, connects v to u in $G_{A'!k}$. Third, the conversion operation places u in $A'!k$ because *t-in* is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Finally, since, the Path Equivalence Lemma implies that $v \longrightarrow^{*}_{c,f}$ *t-in* is in $G_{roll\text{-}out(A)}$, v, like *t-in*, is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that v is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

3) u is any other vertex.

The DAP Equivalence Lemma implies u is in $DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Therefore, because the Path Equivalence Lemma implies paths exist such that v is in $b^{\infty}f^{\infty}(roll\text{-}out(A), u)$, v is in $b^{\infty}f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that v is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

□

8.2.2. The *Pre* Equivalence Lemma

The second result used to prove the Infinite Integration Theorem is the *Pre* Equivalence Lemma, which states that when restricted to vertices having depth $\leq k$, $Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ and $Pre^{hpr}(A'!k, Base!k, B'!k)$ contain the same vertices. The proof is based on the Path Equivalence Lemma, the DAP Equivalence Lemma, and one new lemma: the Affected Points Equivalence Lemma, which shows that $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $AP^{hpr}(A'!k, Base!k)$ contain the same vertices having depth $\leq k$.

The Affected Points Equivalence Lemma

LEMMA. (AFFECTED POINTS EQUIVALENCE LEMMA). *Restricted to vertices having depth* $\leq k$, $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ *equals* $AP^{hpr}(A'!k, Base!k)$.

PROOF. The proof is by mutual containment. First, consider a vertex v in $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. By definition, a directly affected point u exists such that v is in $f^{\infty}(roll\text{-}out(A), u)$. If $depth(u) \leq k$ and u is not a transfer-out vertex having depth k then the Path Equivalence Lemma implies v is in $f^{hpr}(A'!k, u)$ and the DAP Equivalence Lemma implies $u \in DAP^{hpr}(A'!k, Base!k)$; therefore, v is in $f^{hpr}DAP^{hpr}(A'!k, Base!k)$, which implies that v is in $AP^{hpr}(A'!k, Base!k)$.

Otherwise, if $depth(u) > k$ or u is a transfer-out vertex having depth k then the path from u to v in $G_{roll\text{-}out(A)}$ contains at least one transfer-out vertex having depth $\leq k$; let *t-out* be one of these transfer-out vertices. Thus, paths connect u to *t-out* and *t-out* to v in $G_{roll\text{-}out(A)}$, which imply that *t-out* $\in f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $v \in f^{\infty}(roll\text{-}out(A),$ *t-out*$)$. Let *d-out* be the dummy-out vertex that replaces *t-out* in $G_{A'!k}$. By construction, *d-out* $\in DAP^{hpr}(A'!k, Base!k)$ and, by the Path Equivalence Lemma, v is in $f^{hpr}(A'!k,$ *d-out*$)$; therefore, v is in $f^{hpr}DAP^{hpr}(A'!k, Base!k)$, which implies that v is in $AP^{hpr}(A'!k, Base!k)$.

The preceding argument shows that when restricted to vertices having depth $\leq k$, $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)) \subseteq AP^{hpr}(A'!k, Base!k)$. The second half of the mutual containment argument shows that when restricted to vertices having depth $\leq k$, $AP^{hpr}(A'!k, Base!k) \subseteq AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. Let v be a vertex from $AP^{hpr}(A'!k, Base!k)$ (v is not a dummy-use vertex because these vertices have depth $k+1$, which is greater than k). By definition a directly affected point u exists such that v is in $f^{hpr}(A'!k, u)$. If u is not a dummy-out vertex having depth k then the DAP Equivalence Lemma implies that $u \in DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$; therefore, because the Path Equivalence Lemma implies that $v \in f^{\infty}(roll\text{-}out(A), u)$, v is in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies v is in $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

Otherwise, if u is a dummy-out vertex having depth k then, by definition, it replaces a transfer-out vertex *t-out* that is in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. The Path Equivalence Lemma implies $t\text{-}out \rightarrow^{*}_{c,f} v$ exists in $G_{roll\text{-}out(A)}$; therefore, v is also in $f^{\infty}DAP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, which implies that v is in $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. □

The Pre Equivalence Lemma

LEMMA. (*Pre* EQUIVALENCE LEMMA). *Restricted to vertices having depth* $\leq k$, $Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *equals* $Pre^{hpr}(A'!k, Base!k, B'!k)$.

PROOF. Because $AP^{hpr}(A, Base)$ contains all the vertices whose slices differ in A and $Base$, it must be the case that $V(G_A) - AP^{hpr}(A, Base)$ contain all the vertices of A whose slices are the same in A and $Base$. Together with the similar observation for B, this implies that Pre^{hpr} and Pre^{∞} can be defined as follows:

$$Pre^{hpr}(A, Base, B) \triangleq (V(G_A) - AP^{hpr}(A, Base)) \cap (V(G_B) - AP^{hpr}(B, Base)).$$

$$Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B)) \triangleq (V(G_{roll\text{-}out(A)}) - AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))) \cap (V(G_{roll\text{-}out(B)}) - AP^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))).$$

Therefore, when restricted to vertices having depth $\leq k$, $Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ equals $Pre^{hpr}(A'!k, Base!k, B'!k)$ if

(1) $V(G_{A'!k}) - AP^{hpr}(A'!k, Base!k)$ equals $V(G_{roll\text{-}out(A)}) - AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and
(2) $V(G_{B'!k}) - AP^{hpr}(B'!k, Base!k)$ equals $V(G_{roll\text{-}out(B)}) - AP^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$.

Because (1) and (2) have the same proof, only the proof of (1) is given below.

A vertex v is in $V(G_{A'!k}) - AP^{hpr}(A'!k, Base!k)$ iff it is in $G_{A'!k}$ and not in $AP^{hpr}(A'!k, Base!k)$. Since $G_{A'!k}$ and $G_{roll\text{-}out(A)}$ contain the same vertices having depth $\leq k$ (assuming dummy-out vertices are identified with the transfer-out vertices they replace) and the Affected Points Equivalence Lemma implies that when restricted to vertices having depth $\leq k$, $AP^{hpr}(A'!k, Base!k)$ equals $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, v is in $G_{A'!k}$ and not $AP^{hpr}(A'!k, Base!k)$ iff it is in $G_{roll\text{-}out(A)}$ and not $AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, or equivalently, iff v is in $V(G_{roll\text{-}out(A)}) - AP^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$. □

8.2.3. Two Interference Results

This section proves that the absence of Type I and Type II interference from the integration of $roll\text{-}out(Base)$, $roll\text{-}out(A)$, and $roll\text{-}out(B)$ implies the absence of Type I and Type II interference from the integration of $Base!k$, $A'!k$, and $B'!k$. (In the remainder of this chapter, let G_{M^-} denote the intermediate graph produced by $Integrate^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$; similarly, G_{M_k} denotes the intermediate graph produced by $Integrate^{hpr}(A'!k, Base!k, B'!k)$.)

The Type I Interference Lemma, which proves the absence of Type I interference, follows from the Δ Equivalence Lemma, the DAP Equivalence Lemma and a new lemma, the G_{M_k}-G_{M^-} Isomorphism Lemma. The absence of Type II interference follows by extending the G_{M_k}-G_{M^-} Isomorphism Lemma to the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma. A corollary of this second lemma proves the existence of a program P, derived from $M^{\infty}!k$, whose PDG is isomorphic to G_{M_k} (recall that Type II interference exists if no program exists whose dependence graph is G_{M_k}).

The kind of isomorphism proven to exist by the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma and the G_{M_k}-G_{M^-} Isomorphism Lemma are limited isomorphisms (*i.e.*, they apply only to a restricted portion of G_{M_k}). This limited type of isomorphism captures the relationship between $G_{roll\text{-}out(S)}$ and $G_{S!k}$ for any system S. In particular, when restricted to vertices having depth $\leq k$, $G_{roll\text{-}out(S)}$ and $G_{S!k}$ have identical vertex sets (*i.e.*,

$V(G_{roll\text{-}out(S)}) = V(G_{S!k})$). However, for edges having depth $\leq k$, $\mathcal{E} \subseteq \mathcal{E}$, but $\mathcal{E} \not\subseteq \mathcal{E}$ because edges between transfer-in vertices and dummy-out vertices are not in $G_{roll\text{-}out(S)}$. If we ignore these edges, *i.e.*, for edges having depth $\leq k$, excluding those whose source is a transfer-in vertex, then $\mathcal{E} = \mathcal{E}$. This limited kind of isomorphism is formalized as a *k-limited* isomorphism by the following definition.

DEFINITION. (k-limited isomorphism). A *k-limited* isomorphism is an isomorphism between $G_{roll\text{-}out(A)}$ and $G_{A!k}$ restricted to vertices and edges having depth $\leq k$ in which the following assumptions are made.

(1) The bijection for vertices maps transfer-out vertex *t-out* at depth k from $G_{roll\text{-}out(A)}$ to the dummy-out vertex *d-out* in $G_{A!k}$ that replaces *t-out*.

(2) Edges from transfer-in vertices having depth k are ignored.

In addition, since when restricted to vertices and edges having depth $\leq k$, $A'!k$ and $A!k$ differ only in the retagging of certain dummy-out statements, there is a k-limited isomorphism between $G_{roll\text{-}out(A)}$ and $G_{A'!k}$.

The G_{M_k}-G_{M^-} Isomorphism Lemma

Putting together the results from the previous two sections, it is now possible to prove the existence of a k-limited isomorphism between the merged graph G_{M^-} produced by $Integrate^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ and the merged graph G_{M_k} produced by $Integrate^{hpr}(A'!k, Base!k, B'!k)$.

LEMMA. (G_{M_k}-G_{M^-} ISOMORPHISM LEMMA). *G_{M_k} and G_{M^-} are k-limited isomorphic.*

PROOF. G_{M^-} and G_{M_k} are both the union of three sub-graphs (*i.e.*, $Induce\ \Delta(A, Base)$, $Induce\ \Delta(B, Base)$, and $Induce\ Pre(A, Base, B)$). If k-limited isomorphisms exist between corresponding subgraphs then a k-limited isomorphism exists between G_{M^-} and G_{M_k}. First, because a k-limited isomorphism exists between $G_{roll\text{-}out(A)}$ and $G_{A'!k}$ and the Δ Equivalence Lemma implies that $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $\Delta^{hpr}(A'!k, Base!k)$ contain the same vertices having depth $\leq k$, the restriction of the k-limited isomorphism between $G_{roll\text{-}out(A)}$ and $G_{A'!k}$ to these vertices produces a k-limited isomorphism between $Induce\ \Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$ and $Induce\ \Delta^{hpr}(A'!k, Base!k)$. A similar a k-limited isomorphism exists between $Induce\ \Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$ and $Induce\ \Delta^{hpr}(B'!k, Base!k)$. Finally, because the *Pre* Equivalence Lemma implies that $Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ and $Pre^{hpr}(A'!k, Base!k, B'!k)$ contain the same vertices having depth $\leq k$, the restriction of the k-limited isomorphism between $G_{roll\text{-}out(Base)}$ and $G_{Base!k}$ to these vertices produces a k-limited isomorphism between $Induce\ Pre^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ and $Induce\ Pre^{hpr}(A'!k, Base!k, B'!k)$. Thus, because k-limited isomorphisms exist between corresponding subgraphs, G_{M^-} and G_{M_k} are k-limited isomorphic. □

The Type I Interference Lemma

Before stating the Type I interference Lemma, the definitions of Type I interference, for $Integrate^{hpr}(A'!k, Base!k, B'!k)$ and $Integrate^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ are repeated for reference.

$Interference^{hpr}(A'!k, Base!k, B'!k)$ exists iff

$$\Delta^{hpr}(A'!k, Base!k) \cap DAP^{hpr}(G_{M_k}, A'!k) \neq \emptyset \text{ or}$$
$$\Delta^{hpr}(B'!k, Base!k) \cap DAP^{hpr}(G_{M_k}, B'!k) \neq \emptyset.$$

$Interference^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ exists iff

$$\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base)) \cap DAP^{\infty}(G_{M^-}, roll\text{-}out(A)) \neq \emptyset \text{ or}$$
$$\Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base)) \cap DAP^{\infty}(G_{M^-}, roll\text{-}out(B)) \neq \emptyset.$$

LEMMA. (TYPE I INTERFERENCE LEMMA). *If* $Interference^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *is* **false** *then* $Interference^{hpr}(A'!k, Base!k, B'!k)$ *is* **false**.

PROOF. Because it is more direct, a proof of the contrapositive is given (*i.e.*, it is shown that $Interference^{hpr}(A'!k, Base!k, B'!k)$ implies $Interference^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$). To this end, assume, without loss of generality, that $Interference^{hpr}(A'!k, Base!k, B'!k)$ returns **true** because vertex v is in both $\Delta^{hpr}(A'!k, Base!k)$ and $DAP^{hpr}(G_{M_k}, A'!k)$. The proof breaks down into three cases, based on the type of v.

1) v is neither a dummy-use nor a dummy-out vertex.
   First, because the G_{M_k}-G_{M^-} Isomorphism Lemma implies that G_{M^-} and G_{M_k} are k-limited isomorphic, (the argument in) the DAP Equivalence Lemma implies that v is in $DAP^{\infty}(G_{M^-}, roll\text{-}out(A))$. Second, the Δ Equivalence Lemma implies that v is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$; therefore, $Interference^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ returns **true**.

2) v is a dummy-use vertex.
   This case cannot arise because it is impossible for a dummy-use vertex to be in $\Delta^{hpr}(A'!k, Base!k)$ and $DAP^{hpr}(G_{M_k}, A'!k)$. In the case of vertex v, because v is in $G_{A'!k}$ (this is required for v to be in $\Delta^{hpr}(A'!k, Base!k)$) it cannot be in $B'!k$ (the same dummy-use statement is, by definition, not in both $A'!k$ and $B'!k$). Therefore, the only incoming edges on v in G_{M_k} are the incoming edges of v in $G_{A'!k}$, which implies that v is not in $DAP^{hpr}(G_{M_k}, A'!k)$.

3) v is a dummy-out vertex.
   When v is a dummy-out vertex, the transfer-out vertex that v replaces is not necessarily in $DAP^{\infty}(G_{M^-}, roll\text{-}out(A))$. However, to be in $DAP^{hpr}(G_{M_k}, A'!k)$, v must have an incoming edge e in G_{M_k} that is not in $G_{A'!k}$ (Case 2 from Figure 8.8). The remaining ways v can be in $DAP^{hpr}(G_{M_k}, A'!k)$, as shown in Figure 8.8, cannot arise: Case 1, where v is in G_{M_k} but not in $G_{A'!k}$, cannot arise because $v \in \Delta^{hpr}(A'!k, Base!k)$ implies that v is in $G_{A'!k}$. Case 3, where v has an incoming edge in $G_{A'!k}$ but not in G_{M_k}, cannot arise because $v \in \Delta^{hpr}(A'!k, Base!k)$ implies that any incoming edge on v in $G_{A'!k}$

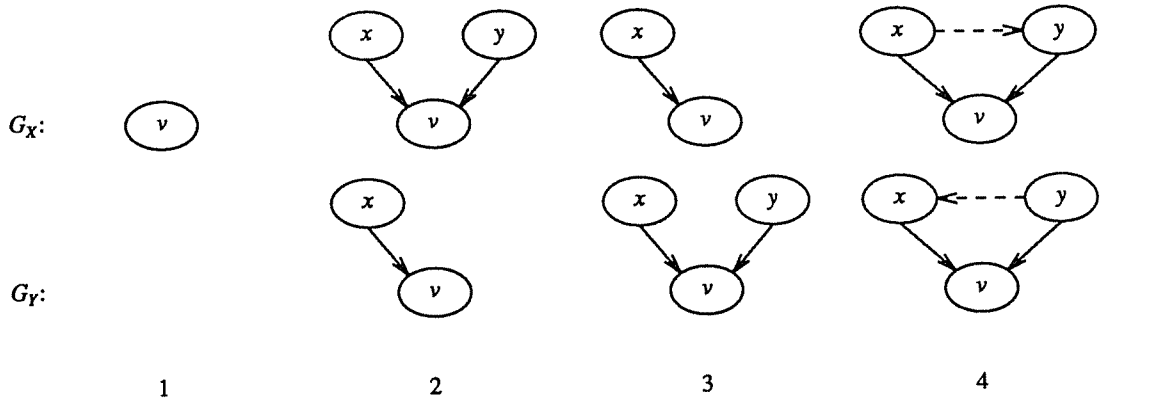


Figure 8.8. This figure repeats Figure 6.5. It shows the four ways a vertex v can be in $DAP(X,Y)$: (1) if v exists in X but not Y, (2) if v has an edge $(y \longrightarrow v)$ in X but not Y, (3) if v has an edge $(y \longrightarrow v)$ in Y but not X, and (4) if the direction of a def-order edge between two of v's predecessors is reversed (*i.e.*, v has different incoming def-order edges).

is in *Induce* $\Delta^{hpr}(A'!k, Base!k)$ and therefore in G_{M_k}. Finally, Case 4, where v has different incoming def-order edges, cannot arise because, by construction, dummy-out vertices (such as v) have no incoming def-order edges.

Thus, consider an edge $e = t\text{-}in \longrightarrow v$ that is in , but not $G_{A'!k}$. In general, the only way an edge not in $G_{A'!k}$ can be in G_{M_k} is if it is in *Induce* $\Delta^{hpr}(B'!k, Base!k)$, which, in the case of edge e, implies that v is in $\Delta^{hpr}(B'!k, Base!k)$. In $G_{roll\text{-}out(B)}$, e summarizes a path p, which ends with *t-out*, the transfer-out vertex replaced by v. Because the Δ Equivalence Lemma implies that v is in $\Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$, this path is in *Induce* $\Delta^{\infty}(roll\text{-}out(B), roll\text{-}out(Base))$ and therefore in $G_{M^{\infty}}$. Since path p is not in $G_{roll\text{-}out(A)}$ (e is not in $G_{A'!k}$), Type I Interference exists: first, following the edges of p backwards from v there must be an edge $x \longrightarrow y$ of p such that y is in $G_{roll\text{-}out(A)}$ but $x \longrightarrow y$ is not. Since $x \longrightarrow y$ is in $G_{M^{\infty}}$, this implies that y is in $DAP^{\infty}(G_{M^{\infty}}, roll\text{-}out(A))$; second, because a subpath of p connects y to v and, by the Δ Equivalence Lemma, v is in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$, y is also in $\Delta^{\infty}(roll\text{-}out(A), roll\text{-}out(Base))$.

□

The G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma

The second interference result is for Type II interference; recall that Type II interference exists for *Integrate*^{hpr} if no program exists whose procedure dependence graph is isomorphic to G_{M_k}. The G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma proves $G_{M^{\infty}!k}$ is isomorphic to G_{M_k} when restricted to vertices and edges having depth $\leq k$ (but including edges from transfer-in vertices having depth k). The corollary to the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma identifies a program P (derived from $M^{\infty}!k$) whose procedure dependence graph is isomorphism to G_{M_k} (without any restrictions). Thus, the corollary proves the absence of Type II interference when *Integrate*^{hpr} is applied to $Base!k$, $A'!k$, and $B'!k$.

The first assumption made in the statement of the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma is necessary because it implies program $M^{\infty}!k$ is well defined: it implies there exists a (set of) programs M^{∞} whose (set of) procedure dependence graphs is $G_{M^{\infty}}$.

LEMMA. (G_{M_k}-$G_{M^{\infty}!k}$ ISOMORPHISM LEMMA). *Assume $G_{M^{\infty}}$ is feasible and* $Interference^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$ *is* **false**. *Restricted to vertices and edges having depth $\leq k$, G_{M_k} and $G_{M^{\infty}!k}$ are isomorphic.*

PROOF. The proof first demonstrates that G_{M_k} and $G_{M^{\infty}!k}$ are k-limited isomorphic and then extends this k-limited isomorphism to include the edges from transfer-in vertices to dummy-out vertices.

By construction $G_{M^{\infty}}$ and $G_{M^{\infty}!k}$ are k-limited isomorphic; by the G_{M_k}-$G_{M^{\infty}}$ Isomorphism Lemma G_{M_k} and $G_{M^{\infty}}$ are k-limited isomorphic. The composition of these isomorphisms implies G_{M_k} and $G_{M^{\infty}!k}$ are k-limited isomorphic (note that dummy-out vertices $d\text{-}out_1$ and $d\text{-}out_2$ from G_{M_k} and $G_{M^{\infty}!k}$, respectively, correspond under the vertex bijection of this isomorphism iff they both replace the same transfer-out vertex in $G_{M^{\infty}}$).

Extending this k-limited isomorphism to an isomorphism for all vertices and edges having depth $\leq k$ requires showing that the same edges connect transfer-in vertices to dummy-out vertices in G_{M_k} and $G_{M^{\infty}!k}$. The proof is by mutual containment: it first proves that such edges in G_{M_k} are in $G_{M^{\infty}!k}$ and then that such edges in $G_{M^{\infty}!k}$ are in G_{M_k}. To begin with, let $t\text{-}in \longrightarrow d\text{-}out$ be an edge in G_{M_k} from transfer-in vertex *t-in* to dummy-out vertex *d-out*. Edge $t\text{-}in \longrightarrow d\text{-}out$ is in G_{M_k} because is it is in *Induce* $Pre^{hpr}(A'!k, Base!k, B'!k)$, *Induce* $\Delta^{hpr}(A'!k, Base!k)$, or *Induce* $\Delta^{hpr}(B'!k, Base!k)$. The proof that $t\text{-}in \longrightarrow d\text{-}out$ exists in $G_{M^{\infty}!k}$ considers each of these possibilities separately (recall that $t\text{-}in \longrightarrow d\text{-}out$ summarizes a path p of edges in the procedure dependence graph of one or more of *roll-out(Base)*,

roll-out(*A*), and *roll-out*(*B*)):

1) *t-in* $\longrightarrow$ *d-out* is in *Induce* $Pre^{hpr}(A'!k, Base!k, B'!k)$.
Because *d-out* is in $Pre^{hpr}(A'!k, Base!k, B'!k)$ it is not in $DAP^{hpr}(A'!k, Base!k)$ or $DAP^{hpr}(B'!k, Base!k)$; thus, *t-in* $\longrightarrow$ *d-out* exists in $G_{Base!k}$, $G_{A'!k}$, and $G_{B'!k}$ and path p exists in *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*). This implies that p exists in $G_{M^{\infty}}$; therefore, an edge *t-in* $\longrightarrow$ *d-out* summarizing p exists in $G_{M^{\infty}!k}$.

2) *t-in* $\longrightarrow$ *d-out* is in *Induce* $\Delta^{hpr}(A'!k, Base!k)$.
When *t-in* $\longrightarrow$ *d-out* is in *Induce* $\Delta^{hpr}(A'!k, Base!k)$ then it is in $G_{A'!k}$, which implies that in $G_{roll\text{-}out(A)}$ path p connects *t-in* to *t-out* (the vertex replaced by *d-out*). Because the Δ Equivalence Lemma implies that *t-out* is in Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)) and the final operator applied by Δ^{∞} is b^{∞}, the vertices and edges of p are in *Induce* Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)) and consequently in $G_{M^{\infty}}$; therefore, the edge *t-in* $\longrightarrow$ *d-out*, which summarizes p, is in $G_{M^{\infty}!k}$.

3) *t-in* $\longrightarrow$ *d-out* is in *Induce* $\Delta^{hpr}(B'!k, Base!k)$.
The proof of this case is identical to the proof of Case (2).

The preceding argument, which is the first half of the proof, demonstrates that the edges from transfer-in vertices to dummy-out vertices in G_{M_k} exist in $G_{M^{\infty}!k}$; The second half of the proof involves showing that such edges in $G_{M^{\infty}!k}$ exist in G_{M_k}. Let *t-in* $\longrightarrow$ *d-out* be an edge in $G_{M^{\infty}!k}$ from transfer-in vertex *t-in* to dummy-out vertex *d-out* and assume *t-out* be the transfer-out vertex in $G_{M^{\infty}}$ that is replaced by *d-out*. As in the first half of the argument, there are three cases to consider:

1) *t-out* is in Pre^{∞}(*roll-out*(*A*), *roll-out*(*Base*), *roll-out*(*B*)).
Because *t-out* is in Pre^{∞}(*roll-out*(*A*), *roll-out*(*Base*), *roll-out*(*B*)), the path of edges p that gives rise to *t-in* $\longrightarrow$ *d-out* in $G_{M^{\infty}!k}$ must exist in $G_{roll\text{-}out(Base)}$, $G_{roll\text{-}out(A)}$, and $G_{roll\text{-}out(B)}$. Therefore, the edge *t-in* $\longrightarrow$ *d-out* exists in $G_{Base!k}$, $G_{A'!k}$, and $G_{B'!k}$, which implies it exists in G_{M_k}.

2) *t-out* is in Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)).
When *t-out* is in Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)), the Δ Equivalence Lemma implies *d-out* is in $\Delta^{hpr}(A'!k, Base!k)$. If the path p that gives rise to *t-in* $\longrightarrow$ *d-out* in $G_{M^{\infty}!k}$ exists in $G_{roll\text{-}out(A)}$ then the edge *t-in* $\longrightarrow$ *d-out* that summarizes p must also exist in $G_{A'!k}$. This edge is in *Induce* $\Delta^{hpr}(A'!k, Base!k)$ and thus in G_{M_k} because its presence in $G_{A'!k}$ implies that *t-in* is also in $\Delta^{hpr}(A'!k, Base!k)$. Otherwise, if path p does not exist in $G_{roll\text{-}out(A)}$ then Type I Interference exists, which violates our assumption that no interference exists, interference exists because by following the edges of p backwards from *t-out* there must be an edge $x \longrightarrow y$ of p such that y is in $G_{roll\text{-}out(A)}$ but $x \longrightarrow y$ is not. Since p and thus $x \longrightarrow y$ are in $G_{M^{\infty}}$, this edge places y in $DAP^{\infty}(G_{M^{\infty}}, roll\text{-}out(A))$; y is also in Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)), because it is connected to *t-out* and *t-out* is in Δ^{∞}(*roll-out*(*A*), *roll-out*(*Base*)); thus, Type I interference exists.

3) *t-out* is in Δ^{∞}(*roll-out*(*B*), *roll-out*(*Base*)).
The proof of this case is identical to the proof of Case (2).

□

COROLLARY. *Let P be the program obtained by adding to* $M^{\infty}!k$ *the dummy-use statements represented by dummy-use vertices in* G_{M_k}. *(1)* G_{M_k} *is feasible, and (2)* G_P *is isomorphic to* G_{M_k}.

PROOF. To show that G_{M_k} is feasible requires finding a program whose procedure dependence graph is isomorphic to G_{M_k}; thus, (1) follows immediately from (2). To show that G_P is isomorphic to G_{M_k} requires extending the argument in the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma, which implies that G_P and G_{M_k} are isomorphic when restricted to vertices and edges having depth $\leq k$. Thus, we must show that G_P and G_{M_k}

contain the corresponding vertices and edges having depth $k+1$ (G_P and G_{M_k} contain no vertices or edges having depth greater than $k+1$).

First, the construction of P implies that G_{M_k} and G_P contain corresponding vertices having depth $k+1$ (*i.e.*, corresponding dummy-use vertices). Second, G_{M_k} and G_P contain corresponding edges having depth $k+1$ because they contain corresponding control, flow, and def-order edges having depth $k+1$. The cases for control edges and def-roder edges are straightforward. First, the only control edges in G_{M_k} and G_P having depth $k+1$ are from a scope vertex to a dummy-use vertex; therefore, since G_{M_k} and G_P have corresponding dummy-use vertices they must have corresponding control edges. Second, G_{M_k} and G_P contain no def-order edges having depth $k+1$.

For flow dependence edges, first consider a flow dependence edge in G_P from transfer-in vertex *t-in* to dummy-use vertex *d-use*. By construction *d-use* exists in G_{M_k}, where is must come from $G_{A'!k}$ or $G_{B'!k}$; without loss of generality, assume that *d-use* comes from $G_{A'!k}$. Thus, the conversion of $A!k$ added the statement represented by *d-use* to $A!k$, which, by construction, implies $t\text{-}in \longrightarrow_f d\text{-}use$ exist in $G_{A'!k}$. Finally, Because no dummy-use vertices exist in $Base!k$, *d-use* is in $DAP^{hpr}(A'!k, Base!k)$ and hence $t\text{-}in \longrightarrow_f d\text{-}use$ is in $Induce\ \Delta^{hpr}(A'!k, Base!k)$ since having $t\text{-}in \longrightarrow_f d\text{-}use$ in $G_{A'!k}$ implies *t-in* is in $\Delta^{hpr}(A'!k, Base!k)$.

Now consider a flow dependence edge $t\text{-}in \longrightarrow_f d\text{-}use$ in G_{M_k} from transfer-in vertex *t-in* to dummy-use vertex *d-use*. By construction, G_P contains *d-use* and, by the G_{M_k}-$G_{M^{\infty}!k}$ Isomorphism Lemma, it contains *t-in* (since the depth of *t-in* is k). For the rest of the we assume that *t-in* assigns to variable x. Edge $t\text{-}in \longrightarrow_f d\text{-}use$ exists in G_P provided there are no definitions of x (other than *t-in*) in the scope. No such definitions exists because for a given scope at most one transfer-in statement can assign to a variable, and dummy-use statements assign to new (previously unused) variables. □

8.2.4. The Infinite Integration Theorem

From the two results on interference proven in Section 8.2.3, it is now possible to prove that $Integrate^{\infty}$ satisfies the HPR integration model, extended to sets of infinite programs (hereafter referred to as the extended integration model). (The original HPR model from Section 1.1 is repeated below for reference.)

(Original) HPR Model of Program Integration

(1) Programs must be written in a simplified programming language that has only scalar variables and constants, assignment statements, conditional statements, while loops, and final output statements (called *end* statements); by definition, only those variables listed in the end statement have values in the final state. 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,[2] 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* in variant *A* that produces different sequences of values in *A* and *Base*, component *c* is in *M* and produces the same sequence of values as in *A* (*i.e.*, *M* agrees with *A* at component *c*).
   (iii) *M* captures the changed behavior of *B*: for any program component *c* in variant *B* that produces different sequences of values in *B* and *Base*, component *c* is in *M* and produces the same sequence of values as in *B* (*i.e.*, *M* agrees with *B* at component *c*).
   (iv) *M* captures the behavior of *Base* preserved in *A* and *B*: for any program component *c* that produces the same sequence of values in *Base*, *A*, and *B*, component *c* is in *M* and produces the same sequence of values as in *Base* (*i.e.*, *M* agrees with *Base*, *A*, and *B* at component *c*).

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

Integrate$^{\infty}$ satisfies the extended model if it satisfies Properties 1, 2, and 3. By definition, *Integrate*$^{\infty}$ satisfies Properties 1 and 3; thus, we must prove that it satisfies Property 2. We do this below by reducing the question of *Integrate*$^{\infty}$ satisfying Property 2 of the extended model to the question of *Integrate*^{hpr} satisfying Property 2 for the original model. This later question has been answered in the affirmative by the following theorem.

THEOREM. (FINITE INTEGRATION THEOREM [Yang90]).[2] *Integrate*^{hpr} satisfies the HPR integration model.

The key to the reduction is the following observation: for any arbitrary fixed initial state σ for which *roll-out*(*A*), *roll-out*(*Base*), and *roll-out*(*B*) terminate normally, there exists a finite integer *k* such that, when evaluated on σ, *Base*$!k$, *A*$'!k$, and *B*$'!k$ are semantically equivalent to (*i.e.* compute the same sequences of values as) *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*), respectively.

THEOREM. (INFINITE INTEGRATION THEOREM). *Integrate*$^{\infty}$ *satisfies Property 2 of the extended integration model.*

PROOF. Let σ be any arbitrary state for which *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) all terminate. Let *k* be one greater than the maximum depth of any executed statement when *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) are evaluated on σ. This choice of *k* implies that *Base*$!k$, *A*$'!k$, and *B*$'!k$ do not execute an abort scope and are therefore semantically equivalent to *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) (for state σ).

Now consider the integration of *Base*$!k$, *A*$'!k$, and *B*$'!k$. Property 2 of the extended model supposes that the integration of *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) is successful; thus, no Type I or Type II

[2] The proof in [Yang90] actually applies to finite single procedure programs without scope statements; however, because scope statements can be removed using variable renaming, the result applies to finite single procedure programs that contain scope statements.

interference exists in the integration of *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*). Therefore, the Type I Interference Lemma and the Corollary to the G_{M_k}-$G_{M^\infty!k}$ Isomorphism Lemma imply that no Type I or Type II interference exists in the integration of *Base*$!k$, *A*$'!k$, and *B*$'!k$. Consequently, *Integrate*^{hpr}(*A*$'!k$, *Base*$!k$, *B*$!k$) is successful and returns a program whose program dependence graph is isomorphic to G_{M_k}.

The remainder of the proof follows from the Finite Integration Theorem and the Strong Form of the Equivalence Theorem [Reps88], which proves that two (single-procedure) programs with isomorphic program dependence graphs are semantically equivalent (*i.e.*, compute the same sequences of values at corresponding program points). First, the Finite Integration Theorem implies that the program produced by *Integrate*^{hpr}(*A*$'!k$, *Base*$!k$, *B*$!k$) satisfies Property 2 of the HPR Model for Program Integration. Second, the Strong Form of the Equivalence Theorem implies that *P*, the program obtained by adding dummy-use statements to $M^\infty!k$ (as in the Corollary to the G_{M_k}-$G_{M^\infty!k}$ Isomorphism Lemma), also satisfies Property 2 of the HPR Model for Program Integration because its program dependence graph is isomorphic to G_{M_k} (this was shown in the Corollary to the G_{M_k}-$G_{M^\infty!k}$ Isomorphism Lemma).

Because *P* satisfies Property 2.i it terminates normally on σ and therefore executes no abort scopes. This implies M^∞ terminates normally on state σ since, when restricted to statements having depth $\leq k$, *P* is syntactically identical to M^∞ (this was proven by the G_{M_k}-$G_{M^\infty!k}$ Isomorphism Lemma); thus, M^∞ satisfies Property 2.i of the extended integration model. Furthermore, this syntactic equivalence also implies that, on state σ, *P* and M^∞ produce the same sequences of values at corresponding program points. Thus, because *P* satisfies Properties 2.i though 2.iv of the HPR Model for Program Integration, M^∞ satisfies Properties 2.i though 2.iv of the extended integration model.

The above argument applies for any arbitrary fixed initial state on which the evaluations of *roll-out*(*Base*), *roll-out*(*A*), and *roll-out*(*B*) terminate, and thus, implies that *Integrate*$^\infty$ satisfies Property 2 of the extended integration model. □

8.3. Chapters 6, 7, and 8 Prove that *Integrate*^{S} is an Acceptable Integration Algorithm

This section discusses how Chapters 6, 7, and 8 combine to prove that *Integrate*^{S} is an *acceptable* multi-procedure program-integration algorithm. Defining the term "acceptable"—especially in the presence of procedures—is a nontrivial task. Chapter 4 considers this problem; the result is Version 2 of the Revised Model for Program Integration. This model uses the concept of roll-out to relate the integration of programs with procedure calls to the (conceptually simpler) integration of programs without procedure calls (the Consistent Semantics Theorem of Chapter 7 justifies this use of roll-out). Because it uses roll-out, Version 2 of the Revised Model for Program Integration involves the concept of integration at two levels:

(1) The conceptual level concerns the integration of rolled-out (possibly infinite) programs.
(2) The concrete level concerns an actual *algorithm* for multi-procedure integration (*i.e.*, an operation that applies to finite representations of programs).

These two levels are syntactically related by roll-out. They are also semantically related by roll-out because the Consistent Semantics Theorem of Chapter 7 proves that roll-out is a semantics-preserving transformation; thus, the syntactic and semantic properties of an operator at the conceptual level can be used to determine the syntactic and semantic properties of the corresponding operator at the concrete level.

In Chapter 4, we introduced the conceptual integration operator *Integrate*$^\infty$ as the operator to which our concrete-level integration algorithm would be related; in Chapter 5, we developed algorithm *Integrate*^{S} as the concrete-level integration operator. The requirements of the two-level model, viewed in terms of operators *Integrate*$^\infty$ and *Integrate*^{S}, are as follows:

(1: a semantic requirement)

The "conceptual" integration operator $Integrate^{\infty}$ must satisfy the original HPR model (given in Section 1.1), extended to sets of infinite programs.

(2: a syntactic requirement)

The concrete integration operator $Integrate^{S}$ must

(i) deal with finite systems,

(ii) succeed in producing M iff $Integrate^{\infty}$ succeeds in producing M^{∞}, and

(iii) be consistent with $Integrate^{\infty}$ (*i.e.* $roll\text{-}out(M)$ must equal M^{∞}).

Property (1) places syntactic and semantic requirements on $Integrate^{\infty}$. Chapter 8 proves that $Integrate^{\infty}$ satisfies Property (1), which includes proving that the set of merged programs produced by $Integrate^{\infty}$ (semantically) captures the changed and preserved behavior of its three arguments. Property (2) is syntactic; in essence, it states that the system produced by operator $Integrate^{S}$ must be syntactically consistent with the set of programs produced by operator $Integrate^{\infty}$. Chapter 6 establishes that $Integrate^{\infty}$ and $Integrate^{S}$ satisfy Property (2). Thus, $Integrate^{\infty}$ and $Integrate^{S}$ satisfy Version 2 of the Revised Model for Program Integration.

To see why satisfying this model implies that $Integrate^{S}$ is an acceptable integration algorithm, let $M = Integrate^{S}(A, Base, B)$ and $M^{\infty} = Integrate^{\infty}(roll\text{-}out(A), roll\text{-}out(Base), roll\text{-}out(B))$. Now, assume that M^{∞} is homogeneous and that either integration is successful (it follows from the Syntactic Correctness Theorem of Chapter 6 that both integrations are successful). To begin with, Chapter 8 proves that M^{∞} (semantically) captures the changed and preserved behavior of $roll\text{-}out(A)$ and $roll\text{-}out(B)$ with respect to $roll\text{-}out(Base)$. Because Chapter 6 proves that M^{∞} (syntactically) equals $roll\text{-}out(M)$, $roll\text{-}out(M)$ also (semantically) captures the changed and preserved behavior of $roll\text{-}out(A)$ and $roll\text{-}out(B)$ with respect to $roll\text{-}out(Base)$. Finally, since Chapter 7 proves that roll-out is a semantics-preserving transformation, this implies that M (semantically) captures the changed and preserved behavior of A and B with respect to $Base$. Thus, because the merged systems it produces capture the changed and preserved behavior of its arguments, $Integrate^{S}$ is an acceptable integration algorithm.

CHAPTER 9

CONCLUSION

9.1. Work Accomplished

The research described in this dissertation covers three topics: the system dependence graph; precise interprocedural slicing; and the integration of multi-procedure programs. The algorithm *Integrate*^{S}, discussed in Chapter 5, is the first algorithm for semantics-based multi-procedure program integration. Semantics-based integration represents a fundamental advance over text-based approaches (*e.g.*, the UNIX utility *diff3*). While previous semantics-based integration algorithms treat only single-procedure programs the ability of *Integrate*^{S} to integrate multi-procedure programs is an essential step toward integrating programs written in a full-fledged programming language.

Unlike the integration of single-procedure programs, where the execution behavior of a program can be modeled by the sequences of values produced by program components, a model for multi-procedure integration must provide a finer level of granularity because it is necessary to account for the calling context in which each sequence of values is produced. The notion of roll-out is used in the two models for multi-procedure integration given in Chapter 4 to capture this finer level of granularity; in essence, the roll-out notion distinguishes between the different calling contexts in which a program component may be executed.

Algorithm *Integrate*^{S}, presented in Chapter 5, satisfies Version 2 of the Revised Model of Program Integration given in Chapter 4 and is thus the first algorithm for semantics-based multi-procedure program integration that takes into account the calling contexts of the changes made to *Base* in A and B. For example, it permits changes in both variants A and B to affect the computation of the same program point, provided they do so in different calling contexts (the direct extension of the HPR algorithm discussed in Chapter 4 fails to integrate such examples because it fails to take calling context into account).

As proven in Chapters 6, 7, and 8, *Integrate*^{S} satisfies the second Integration Model from Chapter 4 and therefore correctly accounts for calling context when determining the changed and preserved behavior of Variants A and B with respect to the *Base*. In particular, Chapter 6 demonstrates that *roll-out*(M) and M^∞ are syntactically identical programs. Because roll-out is a semantics-preserving transformation (as shown in Chapter 8) this implies that M and M^∞ produce the same sequence of values (where the sequence produced "in a context" in M is produced "by an occurrence" in M^∞). Finally, in order to conclude that M satisfactorily integrates Variants A and B with respect to *Base*, we have proven in Chapter 8 that M^∞ satisfactorily integrates *roll-out*(A) and *roll-out*(B) with respect to *roll-out*(*Base*); therefore, M, the merged program produced by *Integrate*^{S}(A, *Base*, B) is a *semantically* acceptable result for the integration of A and B with respect to *Base*.

Accounting for calling context in *Integrate*^{S} requires accounting for calling context in the various steps of *Integrate*^{S}; the results from these steps have uses outside program integration. For example, the term $\Delta^S(P, Q)$, which captures the differences between programs P and Q, can be used to determine a

subprogram of P that computes all the changed behavior of P when compared with the behavior of Q. Such information has obvious uses in software development and software maintenance. For example, if a program maintainer has produced P from Q as the result of a bug fix, then, rather than retesting all of Q's functionality, only that portion represented in $\Delta^S(P, Q)$ must be retested since the remainder of P is semantically equivalent to Q. This is particularly beneficial when $\Delta^S(P, Q)$ is small and P and Q are large.

Precise interprocedural slicing is an important part of *Integrate*^{S}. As with multi-procedure integration, the chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. Chapter 3 develops an algorithm for interprocedural slicing that correctly handles the calling context problem using the transitive dependence (summary) edges in a system dependence graph in conjunction with a two-pass traversal of the graph. This approach leads to more precise slices than previous interprocedural slicing algorithms and provides greater efficiency than other interprocedural slicing algorithms that have similar precision, particularly when computing multiple slices of a single program.

The underlying data structure used by our interprocedural slicing algorithm and *Integrate*^{S} is the system dependence graph. An important property of the system dependence graph is that it captures the meaning of a system in the sense that two systems with isomorphic system dependence graphs have the same meaning (as defined by the meaning function given in Chapter 7). In addition, the system dependence graph has two important features: the explicit representation of (direct) control and data dependence including the dependences that exist across procedure boundaries, and the transitive dependence (summary) edges added at call-sites. Direct dependence edges, which encode a *directly affected relation*, are useful in determining *transitive affected relations* (*e.g.*, an interprocedural slice, $\Delta^S(A, Base)$, etc.). In particular, the direct dependence edges between procedures allow interprocedural edges to be treated as any other edges during certain transitive computations.

Summary edges permit information about a called procedure to be obtained *without* "descending" into the called procedure. This is significant because it avoids the need to record calling-context information when "descending" into the called procedure. Consequently, algorithms that deal with connections between vertices (*e.g.*, interprocedural slicing) can be stated as simple vertex-marking algorithms, rather than more complicated algorithms in which the simple marks are replaced by more complex data structures that record calling context (*e.g.*, a stack of call-site names).

9.2. Future Work

This section discusses four areas for future work on (multi-procedure) program integration: a replacement for the homogeneity test, procedure specialization, algebraic properties of multi-procedure integration, and extending program integration to a full-fledged programming language.

9.2.1. A Replacement for the Homogeneity Test

If we omit the homogeneity test from *Integrate*^{S}, the resulting algorithm may produce an acceptable integrated program (both intuitively and as defined by the stronger integration model given below) even though the program would be rejected by the homogeneity test. For example, the program M shown in Figure 9.1 is determined to be unacceptable because the homogeneity test determines that M^∞, as produced by *Integrate*$^\infty$ and shown in Figure 9.1, is inhomogeneous.[1] However, *Integrate*$^\infty$ represents only one operation that satisfies the requirements on I^∞ as given by Version 2 of the Revised Model of Program Integration in Chapter 4. For this example, *Integrate*$^\infty$ fails to produce a homogeneous program; however, there

[1] The homogeneity test fails because "**call** $P(b)$" $\in$ $M-B$, "$t := x$" is in a procedure called by "**call** $P(b)$" in M and $\neg DAPConnected(Map(Call_P, Linkage(\text{"}t := x\text{"})))$.

Base	Variant *A*	Variant *B*	*M*

procedure *main*
  $a := 1$
  $b := 2$
  call $P(a)$
  call $P(b)$
 	
end
	procedure *main*
	  $a := 1$
	  $b := 2$
	  call $P(a)$
	  call $P(b)$
	  $\boxed{c := a}$
	end
		procedure *main*
		  $a := 1$
		  $b := 2$
		  call $P(a)$
		  $\boxed{}$
		end
			procedure *main*
			  $a := 1$
			  $b := 2$
			  call $P(a)$
			  call $P(b)$
			  $c := a$
			end

procedure $P(x)$
 	
  $t := x$
return
	procedure $P(x)$
	  $\boxed{t2 := x+2}$
	  $t := x$
	return
		procedure $P(x)$
		
		  $t := x$
		return
			procedure $P(x)$
			  $t2 := x+2$
			  $t := x$
			return

roll-out(*Base*, *Main*)	*roll-out*(*A*, *Main*)	*roll-out*(*B*, *Main*)	*Main* of M^{∞}

procedure *main*
  $a := 1$
  $b := 2$
  scope $P(x := a;)$
 	
  $t := x$
  epocs
  scope $P(x := b;)$
 	
  $t := x$
  epocs
 	
end
	procedure *main*
	  $a := 1$
	  $\boxed{b := 2}$
	  scope $P(x := a;)$
	  $t2 := x+2$
	  $t := x$
	  epocs
	  scope $\boxed{P}$ $\boxed{x := b}$;)
	  $t2 := x+2$
	  $\boxed{t := x}$
	  epocs
	  $c := a$
	end
		procedure *main*
		  $a := 1$
		  $b := 2$
		  scope $P(x := a;)$
		
		  $t := x$
		  epocs
		
		end
			procedure *main*
			  $a := 1$
			  $b := 2$
			  scope $P(x := a;)$
			  $t2 := x+2$
			  $t := x$
			  epocs
			  scope $P(x := b;)$
			  $t2 := x+2$
			
			  epocs
			  $c := a$
			end

Figure 9.1. The result of *Integrate*^{S}(*A*, *Base*, *B*) fails the homogeneity test; however, the merged program labeled *M* captures the changed and preserved execution behavior of *A* and *B* with respect to the execution behavior of *Base*. The boxes in *A* and *B* indicate the changes made in *A* and *B*; the boxes in *roll-out*(*A*) identify the slice that must be grafted onto M^{∞} to make it homogeneous.

could be other candidate operations for I^{∞} that—in addition to producing a program that satisfies all the other requirements of the integration model—also produce a homogeneous result. For the example shown in Figure 9.1, such an operation is the following:

> Apply *Integrate*$^{\infty}$, except *graft* onto (*i.e.*, union into) the merged graph $G_{M^{\infty}}$ the slice of *roll-out*(*A*, *Main*) with respect to the second occurrence of "$t := x$" (this slice is highlighted in *roll-out*(*A*, *Main*) of Figure 9.1).

Unfortunately, the idea of defining a better I^{∞} operation by augmenting *Integrate*$^{\infty}$ to graft additional slices from *roll-out*(*A*) (or *roll-out*(*B*)) onto the merged graph does not always work. For example, in Figure 9.2 grafting the slice of *roll-out*(*A*) with respect to the second occurrence of "$t := 1/x$" onto M^{∞} makes $G_{M^{\infty}}$ infeasible: an edge from the assignment "$b := 0$" to the transfer-in statement "$x := b$" already exists in $G_{M^{\infty}}$ and an edge from the assignment "$b := 2$" to this transfer-in statement is in the slice of *roll-out*(*A*) with respect to "$t := 1/x$"; because in the dependence graph for any program, edges from both definitions cannot simultaneously reach the use of *b*, $G_{M^{\infty}}$ plus the slice from *roll-out*(*A*) is infeasible.

In summary, *Integrate*$^{\infty}$ provides only a single *target* for *roll-out*(*M*). The capability to provide multiple targets would reduce the number of examples on which a (different) homogeneity test must return failure due to inhomogeneity. Characterizing the conditions under which alternate targets can be obtained (by grafting slices onto the result of *Integrate*$^{\infty}$ or some other technique) is an open problem.

Base	Variant *A*	Variant *B*	*M*

procedure *Main*
  $a := 1$
  $b := 2$
  call $P(a)$
  call $P(b)$
end()
	procedure *Main*
	  $\boxed{a := 10}$
	  $b := 2$
	  call $P(a)$
	  call $P(b)$
	end()
		procedure *Main*
		  $a := 1$
		  $\boxed{b := 0}$
		  call $P(a)$
		  call $P(b)$
		end()
			procedure *Main*
			  $a := 10$
			  $b := 0$
			  call $P(a)$
			  call $P(b)$
			end()

procedure $P(x)$
  $t := 1/x$
  $x := x+1$
return
	procedure $P(x)$
	  $t := 1/x$
	  $x := x+1$
	return
		procedure $P(x)$
		  $\square$
		  $x := x+1$
		return
			procedure $P(x)$
			  $t := 1/x$
			  $x := x+1$
			return

roll-out(*Base*)	*roll-out*(*A*)	*roll-out*(*B*)	M^{∞}

procedure *Main*
  $a := 1$
  $b := 2$
  scope $P(x := a;$
  $a := x)$
  $t := 1/x$
  $x := x+1$
  epocs
  scope $P(x := b;$
  $b := x)$
  $t := 1/x$
  $x := x+1$
  epocs
end()
	procedure *Main*
	  $a := 10$
	  $\boxed{b := 2}$
	  scope $P(x := a;$
	  $a := x)$
	  $t := 1/x$
	  $x := x+1$
	  epocs
	  scope $\boxed{P}(\boxed{x := b};$
	  $b := x)$
	  $\boxed{t := 1/x}$
	  $x := x+1$
	  epocs
	end()
		procedure *Main*
		  $a := 1$
		  $b := 0$
		  scope $P(x := a;$
		  $a := x)$
		  $x := x+1$
		  epocs
		  scope $P(x := b;$
		  $b := x)$
		  $x := x+1$
		  epocs
		end()
			procedure *Main*
			  $a := 10$
			  $b := 0$
			  scope $P(x := a;$
			  $a := x)$
			  $t := 1/x$
			  $x := x+1$
			  epocs
			  scope $P(x := b;$
			  $b := x)$
			  $x := x+1$
			  epocs
			end()

procedure $P(x)$
  $t := 1/x$
  $x := x+1$
return
	procedure $P(x)$
	  $t := 1/x$
	  $x := x+1$
	return
		procedure $P(x)$
		  $x := x+1$
		return
			procedure $P(x)$
			  $x := x+1$
			return

Figure 9.2. This example, which repeats the one from Figure 5.6, illustrates why a slice of *roll-out*(*A*) (or *roll-out*(*B*)) cannot always be grafted onto M^{∞}. The boxes in *A* and *B* indicate the modifications made to Variants *A* and *B*; the boxes in *roll-out*(*A*) identify the slice that would be grafted onto M^{∞}.

Although an operation that satisfies this goal is unknown, we can capture it in the following stronger model for multi-procedure program integration:

> Integration produces an acceptable integrated program whenever any operation $Integrate^{+}$ (of which $Integrate^{\infty}$ is only one candidate) and $Integrate^{S}$ (with the homogeneity test replaced by some other test) satisfy the requirements on I^{∞} and I^{S} from Version 2 of the Revised Model for Program Integration.

9.2.2. Procedure Specialization

Even when slices cannot be grafted onto M^{∞} to make it homogeneous, it is possible for integration to succeed provided we relax one of the requirements from the integration models given in Chapter 4. Recall that these models require *M* to be constructed from only components of *Base*, *A*, and *B*. However, if we allow copies of a procedure to be created (with new names) then *procedure specialization* is possible: multiple versions of a procedure, each containing different slices (statements) may exist in the merged program. For example, consider again the integration example shown in Figure 9.2. A specialized merged

system that captures the changes from A and B is shown in Figure 9.3.

Conceptually, one way to view procedure specialization is as follows: starting with M^{∞}, replace those scopes that contain the same statements with a call on a procedure having those statements; if two scopes containing different statements have the same name then rename one of the procedures.

Practically, because M^{∞} is a potentially infinite program, we must identify other techniques for procedure specialization. One such approach would be to make use of information obtained form the (failed) homogeneity test; recall that the homogeneity test detects inhomogeneity if either the absent-vertex test or the absent call-site test discovers that an extra occurrence exists in *roll-out*(M). We outline below the actions needed when the absent-vertex test discovers an extra occurrence; the actions taken when the absent-call-site test discovers an extra occurrence are the same except that they are limited to the call-sites in the procedures callable from the absent call-site.

Suppose the absent-vertex test fails because of a vertex v in procedure P of M. In this case, M^{∞} contains a P-scope with an occurrence of v and a P-scope without an occurrence of v. Thus, the specialized merged program must have two versions of procedure P: one with the statement represented by v (*P-with-v*) and one without (*P-without-v*). It is also necessary to determine which call-sites on P should be redirected to *P-with-v* and which should be redirected to *P-without-v*. Furthermore, this process must be iterated because a call on P in procedure Q may require that two versions of procedure Q be created.

9.2.3. Algebraic Properties of Multi-Procedure Integration

Both [Reps90] and [Ramalingam90] discuss the algebraic properties of program integration. (For example, showing that integration is associative with respect to a given base program, *i.e.*, that $Integrate^S(Integrate^S(A, Base, B), Base, C) = Integrate^S(A, Base, Integrate^S(B, Base, C))$.) The approach taken in [Reps90] is to represent (single-procedure) programs as *downwards-closed sets of single-point slices* (a set of slices S is downwards closed with respect to the order "is-a-slice-of", denoted by $\leq$, iff $\forall y \in S$, if $x \leq y$ then $x \in S$; a single-point slice is any graph G such that for some $v \in V(G)$, $G = b^{hpr}(G, v)$). Integration is then recast as an operation in a Brouwerian algebra whose lattice elements are sets of downwards-closed single-point slices. In this framework it is possible to prove that a rich set of algebraic properties, including associativity, hold for single-procedure integration.

The construction of a Brouwerian algebra by forming downwards-closed sets can be applied to any partially ordered set. Consequently, one obvious place to begin the search for an algebra in which the integration operation corresponds to $Integrate^S$ is with some sort of partial order of single-point b slices. Unfortunately, the relation "is-a-slice-of" is not a partial order for b slices: consider a vertex x encountered during the b_2 pass of $b(S, v)$; the slice $b(S, x)$ is not a slice of $b(S, v)$ because only some of x's calling-

Specialized Merged Program

```
procedure Main          procedure P1(x)        procedure P(x)
  a := 10                 t := 1/x
  b := 0                  x := x+1               x := x+1
  call P1(a)              return                 return
  call P(b)
end()
```

Figure 9.3. A specialized merged system for the integration shown in Figure 9.2 that satisfies a relaxed version of the Integration Model from Chapter 4.

contexts are in $b(S, v)$, whereas all of x's calling-contexts are in $b(S, x)$.

Because "is-a-slice-of" is a partial order on b_2 slices, the construction of a Brouwerian algebra using downwards-closed sets of single-point b_2 slices is possible. However, in the resulting Brouwerian algebra, the integration operation corresponds to the HPR algorithm with b^{hpr} replaced by b_2 (and f^{hpr} replaced by $f1$). Since this operation does not correspond to $Integrate^S$, the algebra based on b_2 slices does not say anything about the algebraic properties of $Integrate^S$.

It should not be too surprising that the integration operation of the algebra based on b_2 slices does not correspond to $Integrate^S$ since it does not take calling context into account. Furthermore, even if the construction based on b slices had produced a Brouwerian algebra, it too would be unlikely to contain an integration operation that corresponds to $Integrate^S$ because $Integrate^S$ not only handles the calling-context problem when determining slices, it also handles the calling-context problem when determining other parts of the integrated program (*e.g.*, Δ^S).

A more recent algebraic approach is described in [Ramalingam90] as follows:

> In this paper, we present a new approach to studying program-integration algorithms. In particular, we introduce a new algebraic structure, *fm-algebra*. Here, the concept of program integration derives from the concept of *program modifications* and the idea of *combining* program modifications. If each variant is thought of as having been obtained by performing a certain modification to the base program, an integration algorithm creates a merged program by first combining all the modifications and then applying the resultant modification to the base program. Thus, while the work reported in [Reps90] is based on an algebra of *programs*, the work reported here is based on an algebra of *program modifications* [Ramalingam90].

In [Ramalingam90] two *fm*-algebras are defined; any integration algorithm that satisfies the axioms of either algebra is guaranteed to have certain properties including, for example, associativity. Further work is need to determine if $Integrate^S$ satisfies the axioms of either algebra.

9.2.4. Extending Integration to Realistic Languages

Our ultimate goal is to build a practical integration system to assist programmers with developing and maintaining software. To achieve this goal it is necessary to extend program integration to realistic languages. The ability to integrate programs with procedures and procedure calls represents an important step toward this goal. Thus, the multi-procedure integration discussed in this dissertation is an important step toward the semantics-based integration of programs written in a full-fledged programming language.

Full-fledged programming languages include features such as declarations, nested procedures, input and (unrestricted) output, data types (including arrays, pointers, and records), and control structures (including *do-until* loops, *break*, and *goto* statements). To handle most of these features requires at the very least the application of more powerful data-flow and control-flow analysis techniques during the construction of the procedure dependence graphs. For some features, nested procedure declarations for example, additional changes to the algorithms presented in this dissertation may be required. Although progress has been made on the integration of programs containing some of these features [Horwitz89a, Ball91, Bates91], further study is still required to extend (multi-procedure) program integration to handle the constructs found in a full-fledged programming language.

Bibliography

Aho86.
Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA (1986).

Babich78.
Babich, W.A. and Jazayeri, M., "The method of attributes for data flow analysis: Part II. Demand analysis," *Acta Informatica* **10**(3) pp. 265-272 (October 1978).

Badger88.
Badger, L. and Weiser, M., "Minimizing communication for synchronizing parallel dataflow programs," in *Proceedings of the 1988 International Conference on Parallel Processing*, (St. Charles, IL, Aug. 15-19, 1988), Pennsylvania State University Press, University Park, PA (1988).

Ball90.
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).

Ball91.
Ball, T., *personal communication*. 1991.

Banning79.
Banning, J.P., "An efficient way to find the side effects of procedure calls and the aliases of variables," pp. 29-41 in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY (1979).

Bates91.
Bates, S., *personal communication*. 1991.

Binkley89.
Binkley, D., Horwitz, S., and Reps, T., "The multi-procedure equivalence theorem," TR-890, Computer Sciences Department, University of Wisconsin, Madison, WI (November 1989).

Binkley91.
Binkley, D., Horwitz, S., and Reps, T., "Program integration for languages with procedure calls," Technical Report (in preparation), Computer Sciences Department, University of Wisconsin, Madison, WI (1991).

Callahan88.
Callahan, D., "The program summary graph and flow-sensitive interprocedural data flow analysis," *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. 47-56 (July 1988).

Cooper84.
Cooper, K.D. and Kennedy, K., "Efficient computation of flow insensitive interprocedural summary information," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, (Montreal, Can. June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 247-258 (June 1984).

Cooper88.
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).

Donzeau-Gouge84.
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).

Ferrante87.
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).

Gordon79.
Gordon, M., *The Denotational Description of Programming Languages*, Springer-Verlag, New York, NY (1979).

Horwitz87.
Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," TR-690, Computer Sciences Department, University of Wisconsin, Madison, WI (March 1987).

Horwitz88.
Horwitz, S., Prins, J., and Reps, T., "On the adequacy of program dependence graphs for representing programs," pp. 146-157 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Horwitz88a.
Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *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. 35-46 (July 1988).

Horwitz89a.
Horwitz, S., Pfeiffer, P., and Reps, T., "Dependence analysis for pointer variables," *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, (Portland, OR, June 21-23, 1989), *ACM SIGPLAN Notices* **24**(7) pp. 28-40 (July 1989).

Horwitz89.
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).

Hwang88.
Hwang, J.C., Du, M.W., and Chou, C.R., "Finding program slices for recursive procedures," in *Proceedings of IEEE COMPSAC 88*, (Chicago, IL, Oct. 3-7, 1988), IEEE Computer Society, Washington, DC (1988).

Kastens80.
Kastens, U., "Ordered attribute grammars," *Acta Informatica* **13**(3) pp. 229-256 (1980).

Knuth68.
Knuth, D.E., "Semantics of context-free languages," *Math. Syst. Theory* **2**(2) pp. 127-145 (June 1968).

Kou77.
Kou, L.T., "On live-dead analysis for global data flow problems," *Journal of the ACM* **24**(3) pp. 473-483 (July 1977).

Kuck72.
Kuck, D.J., Muraoka, Y., and Chen, S.C., "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up," *IEEE Trans. on Computers* **C-21**(12) pp. 1293-1310 (December 1972).

Kuck81.
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).

Lyle86.
Lyle, J. and Weiser, M., "Experiments on slicing-based debugging tools," in *Proceedings of the First Conference on Empirical Studies of Programming*, (June 1986), Ablex Publishing Co. (1986).

Myers81.
Myers, E., "A precise inter-procedural data flow algorithm," pp. 219-230 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

Notkin85.
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).

Ottenstein84.
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).

Pfeiffer91.
Pfeiffer, P., "Dependence-based representations for programs with reference variables," Ph.D. dissertation, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

Ramalingam90.
Ramalingam, G. and Reps, T., "A theory of program modifications," TR-974, Computer Sciences Department, University of Wisconsin, Madison, WI (October 1990).

Reps88a.
Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

Reps88.
Reps, T. and Yang, W., "The semantics of program slicing," TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

Reps89a.
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).

Reps89.
Reps, T. and Yang, W., "The semantics of program slicing and program integration," pp. 360-374 in *Proceedings of the Colloquium on Current Issues in Programming Languages*, (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Vol. 352, Springer-Verlag, New York, NY

(1989).

Reps90.
Reps, T., "Algebraic properties of program integration," pp. 326-340 in *Proceedings of the Third European Symposium on Programming*, (Copenhagen, Denmark, May 15-18, 1990), *Lecture Notes in Computer Science*, Vol. 432, ed. N. Jones,Springer-Verlag, New York, NY (1990).

Schmidt86.
Schmidt, D., *Denotational Semantics*, Allyn and Bacon, Inc., Boston, MA (1986).

Weiser83.
Weiser, M., "Reconstructing sequential behavior from parallel behavior projections," *Information Processing Letters* **17**(5) pp. 129-135 (October 1983).

Weiser84.
Weiser, M., "Program slicing," *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).

Yang90.
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).