A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis

Judith A. Stafford

Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

University of Colorado
Department of Computer Science
Technical Report CU-CS-907-00 August, 2000

Abstract

Dependence relationships among the statements of a program are important to understand for various software development and maintenance purposes. The program's dependence graph is used as a base for various types of program analyses. A dependence graph represents the potential for one statement in a program to affect another in terms of the control and data dependencies among a program's statements. A dependence graph is a directed multi-graph; the vertices of the graph represent the statements in a program and the arcs represent control and data dependencies separately. During the past two decades the value of a dependence graph as a program representation has been recognized by a wide audience and the definition has been extended in various ways in order to incorporate dependence relationships in various types of programs.

This dissertation concentrates on the control dependence aspect of program dependence and describes a new approach to identifying control dependencies in sequential, imperative, multi-procedure programs. The approach is formal, compositional, and language independent. It addresses previously identified pitfalls associated with identifying control dependencies in programs that contain procedure calls. Additionally, because it is rigorously defined, it provides a foundation for reasoning about its potential use as a base for formal extension to other types of programs.

Models of control dependencies for uni-procedure programs are typically based on composing two program representations: the control flow graph and the forward dominance tree. The key observation underlying the work described in this dissertation is that the notion of forward dominance has not been carried forward into approaches to computing control dependencies in more complex types of programs. The forward dominance relationship, as previously defined, is not effective for use in identifying control dependencies in non-inlined control flow representations of multi-procedure programs.

In this thesis we extend the definitions of control flow, forward dominance, and control dependence for application to multi-procedure programs. We describe structures that represent the interprocedural relationships in a program. We describe and define interprocedural forward dominance and a related program representation called the *forward dominance forest* and its use to identify control dependencies in multi-procedure programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dependencies of many types exist among elements of software systems. Understanding the nature and extent of these interdependencies is critical to many software development and maintenance tasks. For instance, during compilation of source code it is helpful to identify dependencies among program statements so that various optimizing code transformations can be made without affecting the functionality of the program [3], and during debugging it is helpful to isolate all the code on which an error-producing statement depends for calculating its result in order to more easily determine the source of the error [42]. One can exploit dependence information at higher levels of abstraction in order to reason about possible effects of changing a component in a system [50].

Various forms of system description are available to system developers based upon the phase of the software life cycle to which the description applies and the type of system being developed. Early work in understanding system dependencies concentrated on programs that are comprised of a single procedure written in sequential, imperative programming languages [17, 41]; in this dissertation these types of programs are referred to as *uni-procedure programs.* More recently, research has been directed at more complex types of languages used to describe programs with procedure calls [22, 27, 33, 37, 47], object-oriented programs [31, 57], concurrent programs [14, 23, 56], concurrent object-oriented programs [57], software architectures [48, 55], and hardware architectures [15].

Work in the area of identifying control dependencies in uni-procedure programs produced two, very similar formal models, one by Ottenstein and Ottenstein [39] and the other by Podgurski and Clarke [41]. These formal models are based on combining control flow and forward dominance information to produce control dependence information. There have been several attempts to directly extend these models for application to programs with procedure calls [22, 27, 37, 47], which we refer to as *multi-procedure programs.* These attempts fall into two categories. In the first, a program control flow graph is constructed by inserting individual copies of each procedure's control flow graph after each call to a procedure [22]. This is known as an *inlined control flow graph (ICFG).* In the other, one copy of a procedure's associated control flow graph is used, then control flow arcs are inserted between each call site and the appropriate procedure's control flow graph to represent the call to, and return from, the procedure. This is referred to as a *one-to-one*

CFG representation in this dissertation.

The inlining approach works because it represents the program as a super control flow graph with the same characteristics as the CFG used to define the forward dominator tree. However inlining is impractical to use, due to the growth in the size of the representation when procedures are called from multiple call sites. Approaches based on the one-to-one CFG representation have encountered several issues with respect to both the precision and the conservativeness of the identified sets of dependencies. Lack of conservativeness (i.e., not identifying all potential dependencies), results from failure to address the effects of program control mechanisms such as halts, non-terminating loops, and infinite recursion. Lack of precision results from failure to reflect the calling context of a procedure when calculating dependencies. To deal with these issues, researchers have created *ad hoc* extensions to the control flow graph and the program dependence graph in the form of dummy vertices and additional types of control dependencies, each intended to deal with a specific type of dependence created by a specific type of control control mechanism.

1.1 A New Approach

Horwitz et al. recognized that it would be useful to be able to use methods based on dependence graphs to identify dependencies in programs that are incomplete or are composed of precompiled libraries [27]. To our knowledge, no method exists that provides support for such analysis for programs in which global interactions may lead to non-termination of the program or non-return to calling procedures. This thesis describes an approach to interprocedural control dependence analysis that provides this support. The approach consists of a model and a set of algorithms for automatically constructing the structures that comprise the model. The model consists of a set of structures including a call graph, and extensions of the control flow graph, the forward dominance tree, and the control dependence graph. The model is presented in Chapter 3 and algorithms for constructing it are described in Chapter 4.

1.1.1 Guiding Principles

Three guiding principles were followed during development of our model: we wanted to create a model that is compositional, language independent, and defined with mathematical rigor. Our reasoning was as follows:

Compositional It is useful to be able to determine intraprocedural dependencies once for each procedure then use that information when composing programs.

Current trends in system development point towards increased use of prefabricated components known at COTS (commercial, off-the-shelf) components. Our vision is to extend this compositional model for use with COTS components. A component manufacturer can opt to calculate intra-component dependencies once and supply them along with the component for use in identifying program-wide dependencies at program composition time.

Language Independent Modern programs are generally comprised of procedures written in a variety of languages. Thus, any model for program-wide analysis must be based on a language independent representation. Additionally, language independence allows for a particular analysis technique to be applied to a larger number of programs than is possible for analyses that are tailored for specific languages. The language independence of the model is currently restricted to sequential, imperative languages. Extension to other types of languages is left as an area for future work.

Rigorously Defined We wish to use this model as a foundation to be used when developing dependence models of other types of programs. As such, it must be possible for one to recognize its limitations and power. A mathematically defined model provides support for reasoning about such properties.

The model is based on Podgurski and Clarke's formal model of control dependencies for uniprocedural programs [41]. The key insight of the work presented in this dissertation is the limitations of the applicability of forward dominance, and the power that would result from extending this notion in a way that supports interprocedural dependence analysis in a one-to-one graph approach. This extension is called *procedure forward dominance*. It recognizes the potential for non-return from procedure calls. That is, it is not possible to know until a program is composed whether statements within a procedure that follow a procedure call will forward dominate the procedure call. Each procedure is represented by a new representation of control flow within a procedure called a *procedure control flow graph* (PCFG). From this PCFG, a *procedure forward dominance forest* (PFDF) is constructed; these two graphs are used in combination to identify control dependencies among statements in a procedure and these are represented in a *procedure control dependence graph* (PCDG).

The primary contributions of this dissertation are the formal description of a language independent, compositional model of dependencies in multi-procedure programs and the algorithms for constructing the structures that comprise the model. A forward dominance forest (FDF) is defined to support reasoning about program-wide forward dominance relationships. At program composition time, statement-level control dependencies among procedures are identified by using the FDF, the program's *call graph* (CG), and the program's PCDGs in combination. Alternatively, the FDF may be used in conjunction with a CG to identify control dependencies among procedures. An additional contribution of this thesis is extension of Podgurski and Clarke's notion of weak control dependence to reflect that non-return from a procedure call may be due to embedded halts or it may be due to non-termination of loops in called procedures, or infinite recursion among subsequently called procedures. While other work has considered the potential for non-return due to embedded halts [22] and arbitrary control flow [47] in the form of Java exceptions, to our knowledge accounting for potential non-return due to loops and recursion has not been previously addressed.

1.2 Evaluation

We evaluate the model in several ways:

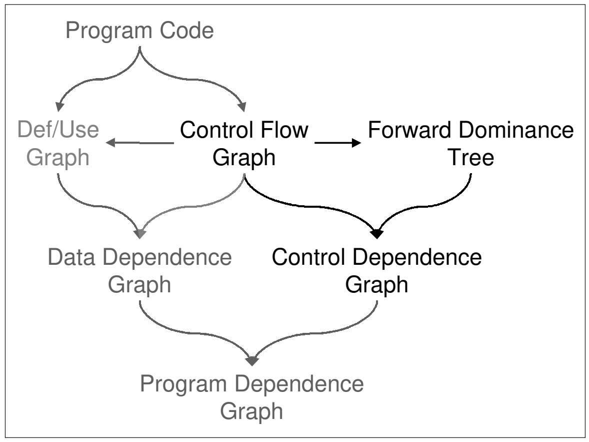

Figure 1.1: Structures of a Dependence Model and the Relationships Among Them.

Correctness We evaluate the model's ability to support modeling of all control mechanisms supported by commonly used, sequential, imperative programming languages. We show that the same set of control dependencies are identified through application of our compositional model as those obtained using traditional control dependence applied to an inlined control flow graph.

Complexity We evaluate the algorithm's complexity and compare it to the complexity of other, published, interprocedural analysis algorithms.

Size of the Representation We compare the space requirements of the structures of the model to those used in other interprocedural dependence models.

Usefulness We show that our algorithms are conservative and at least as precise as other published interprocedural control dependence algorithms. Additionally we show that the model satisfies the guiding principles for its development in that it possesses the positive characteristics of compositionality, formality, simplicity, and language independence.

1.3 Scope and Contribution

This dissertation confines itself to discussion of the control dependence aspects of dependence analysis. Figure 1.1 describes a model of program dependence analysis that is composed of both data and control dependencies. The graph shown in that figure represents the relationships among structures that comprise dependence models that have been developed for application to sequential, imperative programs with arbitrary control flow. An arc in the graph represents the fact that the source of the arc is used in the creation of the target of the arc. We extend one such model for application to multi-procedure programs; we describe and defined the following structures associated with this extension in Chapter 3: the procedure control flow graph (PCFG), the procedure forward dominance forest (PFDF), the procedure control dependence graph (PCDG), forward dominance forest (FDF), program call graph (CG), and compound control dependence graph (CCDG) and its associated resolved control dependence graph (CDG). Algorithms for the construction of all structures are provided in Chapter 4 with the exception of the procedure control flow graph. As mentioned earlier, construction of the procedure control flow graph is a language-dependent operation. Availability of a PCFG for each procedure in the program is assumed in our algorithms.

1.4 Roadmap

This dissertation is organized as follows: Chapter 2 contains a review of dependence analysis and related graph theoretic terminology and an introduction to dependence analysis including an historical overview of prior work in the area, discussion of some related topics, and ending with a description of the limitations of earlier approaches to interprocedural dependence analysis. Chapters 3 and 4 describe the contributions in terms of both the model and the algorithms used to construct the structures that make up the model. Chapter 5 contains an evaluation of the usefulness of the model. Chapter 6 describes related work in the area of interprocedural program dependence analysis and positions this work in that field. Chapter 7 summarizes the contributions of this work and describes some promising areas for follow-on research.

Chapter 2

Foundations and Background

This chapter provides background necessary to understand the research described in this dissertation, the context in which the work was carried out, and the contributions of this research. A dictionary of graph theoretic terminology, which contains an alphabetized listing of related terms and their definitions, is provided in Appendix B. The reader should consult this dictionary as needed while reading this and subsequent chapters of the dissertation. We begin this section with a description of dependence analysis and continue with an historical look at dependence analysis research. We finish with discussion of difficulties encountered in prior work in the area of graph-based interprocedural dependence analysis.

2.1 Dependence Analysis

Informally, dependence analysis is the automatic identification of the potential for one element (e.g., a program statement) of a program or system to affect or be affected by other elements of the program or system or its environment. Work in this area has focused primarily on identifying dependencies among statements in computer programs. Generally, analyses that are based on dependence analysis, such as program slicing, are concerned with identifying dependencies with respect to a specific statement by determining which other statements have the potential to affect the value of data used in the computation performed at a statement or the potential to determine whether or not the statement will be executed.

Before one begins a study of dependence analysis it is helpful to understand some related concepts: static versus dynamic analysis, direct versus indirect dependencies, conservativeness versus precision of analysis, and aliasing among program elements.

- *Static:* Static analysis is based solely on the program code. It applies to all possible runs of a program. Dependence analysis algorithms are generally based on static analysis.

- *Dynamic:* Dynamic analysis provides information that is run specific. That is, the results apply to the program given the initial condition of the program and the inputs that it received during execution. Program testing is an example of dynamic analysis.

- *Direct:* A direct dependence (u,v) exists if v depends on u and does not depend on any other element w that depends on u and for which there exists a w-v path.

- *Indirect:* An indirect dependence (u,v) exists if there exists an element $w \neq u$ and $w \neq v$ such that v directly depends on w and w directly or indirectly depends on u.

- *Conservativeness:* A dependence algorithm is said to be "conservative" if it produces a set of dependencies that is a superset of the set of dependencies that would exist given all executions of the program (i.e., the set must contain at least all true dependencies and may contain some false dependencies). This is not a hard problem for dependence analysis. In fact, it is easy to be conservative; one simply assumes that every statement in the program can affect every other statement. The problem with this approach is that the information that results from such an algorithm is useless.

- *Precision:* A dependence algorithm is said to be "more precise" if it contains fewer false dependencies than some other conservative algorithm. Any useful dependence analysis algorithm must not only be conservative but it must also be precise enough to be useful. given a specific set of criteria.

- *Aliasing:* Aliasing occurs in programs when the same memory location is referenced by variables with different names. In dependence analysis we are most concerned with the possibility of aliasing among variable or procedure names. This is an issue for static dependence analysis because it is difficult to create precise algorithms when it is difficult to determine which variable names reference a given memory location.

There are several types of analysis that are used in order to identify dependencies in programs: control flow analysis, data flow analysis, control dependence analysis, and data dependence analysis. We describe each of these briefly below.

2.1.1 Control flow analysis

Control flow analysis is the process of identifying which statements in a program have the potential to lead directly to the execution of other statements based on the syntax of the program code. Control flow analysis is the most fundamental aspect of dependence analysis. Faithful representation of potential flows among program statements is essential to the identification of which variable definitions can reach variable uses and which statements control the execution of other statements. Control flow relationships among statements in a program are transitive and can be represented in a graph, referred to as a *control flow graph (CFG)* in which the vertices represent the program statements and the arcs represent the potential for control to flow from the statement represented by the source of an arc to the statement represented by the target of the arc. If the vertices in the graph are annotated with information about which variables are defined and which are used in the statement represented by the vertex, then the graph is referred to as a *def/use graph (DUG)*.

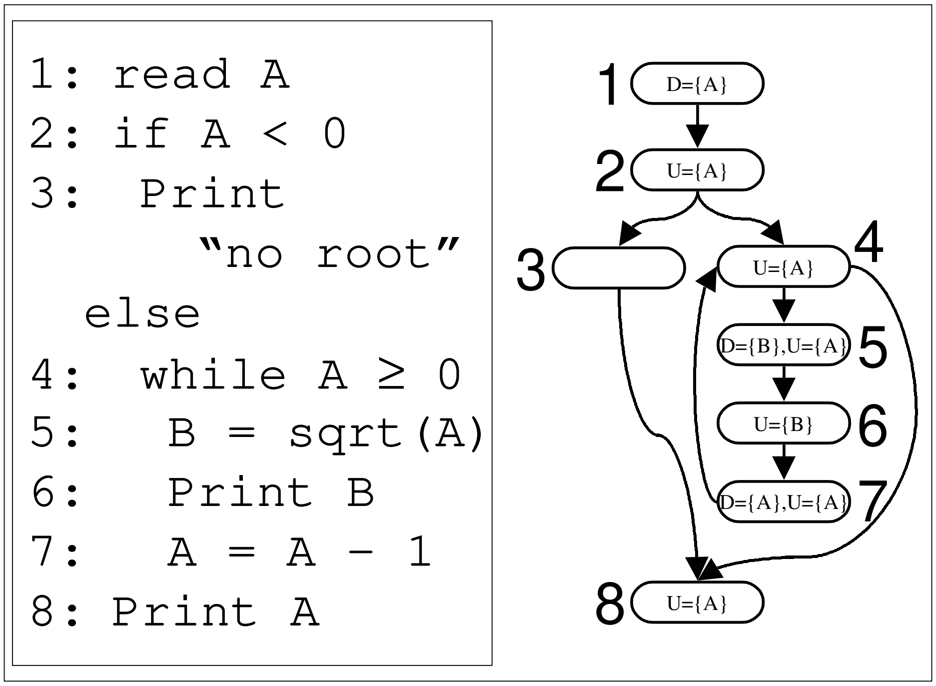

Figure 2.1: Example Code and its Def/Use Graph (DUG).

Constructing a CFG is a language dependent task and lies outside the scope of this dissertation. The exact form of CFG that is required for use in this model is defined in Chapter 3. Figure 2.1 contains pseudo code for a sequence of statements in a program along with the related DUG. As examples of control flow relationships, referring to the code in the figure, execution of statement 1 can only lead to execution of statement 2. After statement 2, either statement 3 or statement 4 will be executed depending on evaluation of the condition in statement 2. If statement 4 is executed then, depending on whether $A \geq 0$, either the loop beginning with statement 5 will be executed or the program will finish. The vertices and arcs of the graph shown in the right side of the figure represent the flow of control in the program. The def/use graph for this code is shown in the right side of the figure.

2.1.2 Data flow analysis

Data flow analysis is the process of identifying the potential for a value computed in one statement to affect the computation in another. There are a variety of ways in which data flow information can be computed. The first, and still most widely used method is known as iterative data flow analysis. Its introduction is attributed to Vyssotsky and Wegner [52] while working on analysis methods for FORTRAN at AT&T Bell Laboratories in Murray Hill, New Jersey. During data flow analysis sets of variable information known as "gen" and "kill" sets, representing the variable values that are generated (defined) and killed (undefined) respectively are computed for each statement in the program. These sets are manipulated by data flow equations in order to solve problems relating to the potential for a variable definition to affect a computation later in a program's execution.[1] This information is most widely recognized as being useful for identifying opportunities for safe code transformations in optimizing compilers [1, 38].

Two basic data flow problems are the "reaching definitions" problem and the "live variable" problem. Solutions to the reaching definitions problem are sought relative to a specific statement. The goal is to identify all the locations of definitions for all variables that may reach (i.e., are available for use at) a particular statement in a program. The solution to a reaching definitions problem for a particular statement S is a set of tuples where each tuple $(x, dloc(x))$ represents the fact that statement $dloc(x)$ defines x and there is a sequence of statements leading from $dloc(x)$ to S that does not contain any other statements that define x. Solutions to the live variables problem provide the set of similar tuples for all variable values that may be used after a given statement. Solutions to these two problems can be used to create variable def-use and use-def chains. These chains provide information that is sufficient to support many powerful code optimizations and software maintenance activities. For instance, if a def-use chain for a particular variable terminates with a definition at a particular statement, then it is clear that that definition is not used later in the program and it can be removed from the optimized code [1, 38]. On the other hand, such an anomaly could be the result of a typing error when entering the program's code. The static detection and resolution of such anomalies is an important software maintenance activity [18].

As an example of def-use chaining, referring again to the code in Figure 2.1, a set of reaching definitions for statement 2 is {(`A`,1)}. The set of live variables for statement 2 is {(`A`,4),(`A`,8)}. The set of reaching definitions for statement 8 is {(`A`,1),(`B`,5),(`A`,7)}. In order to find all the statements that contain definitions that could affect the value of `A` at statement 8 we calculate the reaching definitions for statement 7, since the value of `A` defined there reaches statement 8. Reaching definitions for statement 7 are {(`B`,5),(`A`,1),(`A`,7)}. Thus a def-use chain for `A` at statement 8 is 1-7-8 with some number of possible definitions at statement 7 due to its inclusion in a loop.

There are an infinite number of paths over which data can flow from statement 1 to statement 8 depending on the exit condition of the loop headed by statement 4: 1-2-3-8, 1-2-4-8, and infinitely many beginning with the prefix 1-2-4-5-6-7, continuing with zero or more additional traversals over 4-5-6-7, and ending at 8. This detail of infinite numbers of paths that result from the existence of loops in programs requires special handling during data flow analysis. Other issues that cause difficulties during data flow analysis are variable and procedure aliasing, procedure calls, non-local jumps, and recursion.

[1] The level of abstraction at which data flow analysis is applied can vary. We use the word "statement" in order to simplify the explanation, but it should be noted that the analysis can be applied at other levels of abstraction.

2.1.3 Graph-based Program Dependence Analysis

Graph-based program dependence analysis is the process of determining when one statement of a program has the potential to affect or be affected by the computation made at another statement in the program. Two types of dependencies are used in combination to identify statements that have potential to affect another: control and data dependencies.

- *Control dependence analysis* is the process of identifying, from the program syntax, the potential for a statement represented by a vertex in a control flow graph to determine the number of times a program statement will be executed. For instance, an if-statement branches one way or the other, and thus the statements in only one side of the branch will be executed; the condition of a "while" determines whether a loop is entered and when it is exited, and thus determines the number of times the statements in the loop are executed. Control dependence analysis is used in conjunction with information about the locations of definitions and uses of variables to determine program dependencies.

- *Data dependence analysis* is the process of identifying when there is a path in a control flow graph that connects a definition of a variable to a use of the variable. A def/use graph can be used as a base for def-use chaining, which can be computed efficiently using data flow analysis, to identify data dependencies in a program; the results of def-use chaining can be recorded in a dependence graph.

The results of data and control dependence analysis can be recorded in a directed multi-graph of the program's dependencies. If a chain of control dependencies and/or data dependencies leads from one vertex to another in a dependence graph, then there is a dependence between the statements represented by the vertices. The use of dependence graphs as a base for code optimization allows many optimizations to be performed more quickly than with other program representations, since irrelevant ordering among unrelated statements that is a feature of the control flow graph are removed in the dependence graph. For example, the order of statements 6 and 7 of Figure 2.1 is irrelevant; as shown in Figure 2.2, they are both control dependent on statement 4 and there is no data dependence between them so there will be no connection between them in the dependence graph. The dependence graph makes explicit which parts of the program are independent and thereby exposes opportunities for automatic parallelization.

Many program understanding problems can be solved by performing a transitive closure over the dependence graph beginning at specific vertices. The dependence graph can be used in this way to identify program slices. Weiser defined a program slice as being the set of statements in a program that could possibly affect the value of a variable at a specific statement in the program [54]. He proved that the problem of identifying "statement minimal" slices is unsolvable. That is, it is not possible in general to statically identify the precise set of dependencies for a particular variable at a particular statement in a program. Weiser's data flow-based algorithms for computing program slices are proved to be conservative for structured programs; precision is a secondary concern about which he makes no claims. Slicing based on dependence graphs is a less costly approach and can produce more precise slices [17, 25, 39].

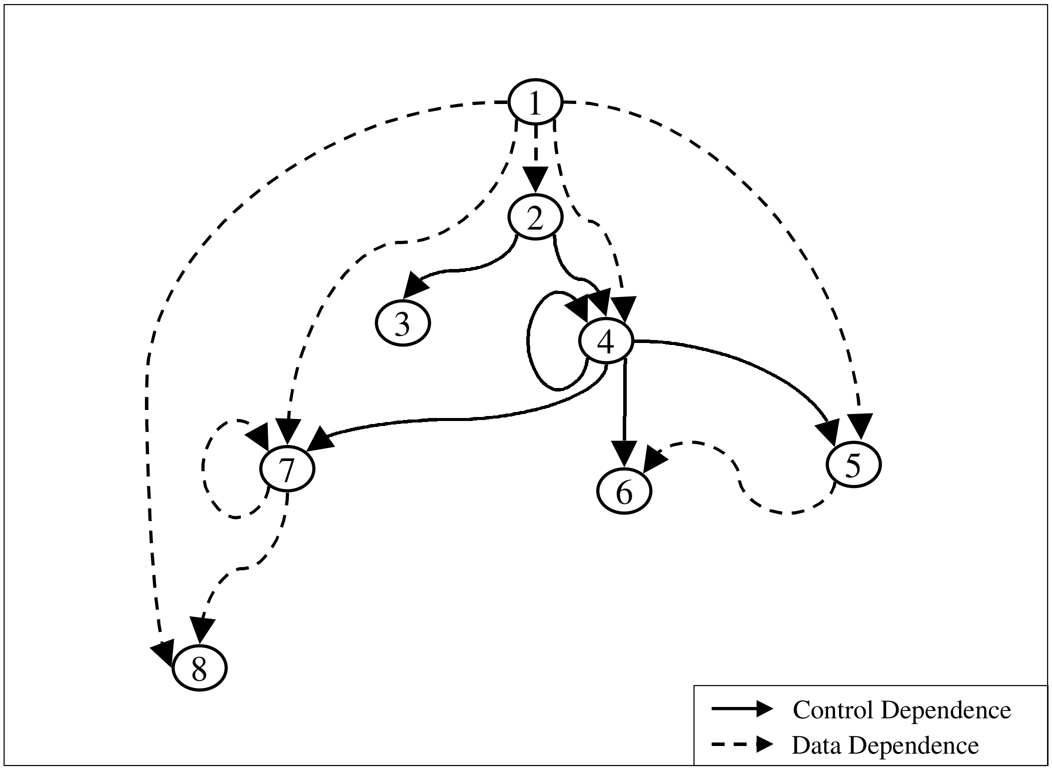

Figure 2.2: Dependence Graph for Example Code in Figure 2.1.

2.2 History of Dependence Analysis Research

Our search of the literature finds that the first discussion of automated program dependence analysis should be attributed to Prosser [43], who in 1959 made the observation that graphical representations of the control flow among program statements and the related dominance relationships among vertices of the graph could be used to automate identification of dependencies among program statements. In the late 1960s researchers observed that automated identification of dependencies among instructions in computer programs could be used as an aid to safe code restructuring during code optimization [2, 36], thereby allowing programmers to write more flexible, higher-level code that could later be optimized into the inflexible, highly-efficient code that machines prefer as input. By the late 1970s Denning and Denning [16] were using program dependencies to reason about secure information flow within programs. Fosdick and Osterweil suggested the use of data flow analysis to identify anomalies in program code and for producing program documentation about various aspects the definitions and uses of variables in the program [18]. During the last two

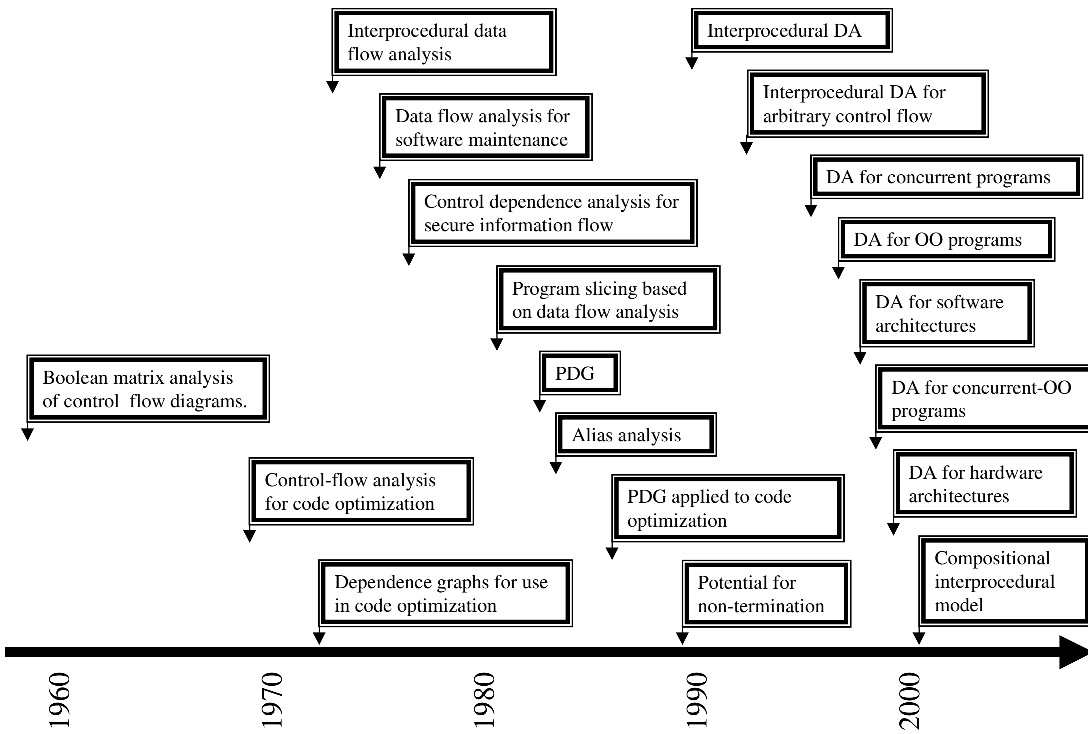

Figure 2.3: Timeline of Research in Program Dependence Analysis.

decades Weiser and others [23, 27, 40, 47, 56, 50] have applied program dependence analysis to a wide variety of software maintenance and understanding activities. Figure 2.3 shows a chronology of major steps in dependence analysis research. The remainder of this section contains a concise history of dependence analysis, including descriptions and discussion of major contributions along the way.

2.2.1 1959

In December 1959, Prosser, a scientist at MIT's Lincoln Laboratory, introduced the notion of automated reachability analysis [43] by way of applying matrix addition and multiplication to three types of matrices: one that records the potential for control to flow from one statement directly to another, another that records the def/use relationships among statements, and another that records the dominance relationships. It appears, after an extensive search of the bibliographies of research papers in this area, that Prosser was the first to describe the use of the graph-based dominance relationship to identify relationships among program statements. He talks about applications such as automated debugging and identification of opportunities for creating subroutines. Questions reported in the Q/A section of the conference version of the paper ask about issues such as identifying conditions that could cause a loop to be executed different numbers of times and also issues related to correctly accounting for the calling context of procedures when traversing interprocedural control flow graphs. Prosser confirms that a much more sophisticated analysis is required to address these issues.

2.2.2 1960 — 1969

Although there are a few earlier references to the use of various structures to represent computer program flows, most notably that of Prosser described above, the main thread of program dependence analysis research appears to have begun with Allen, Cocke, and others at IBM's Thomas J. Watson Research Center during the late 1960s. These scientists recognized that knowledge about flows in a program could be used as an aid to various code optimizations that involve code restructuring. Work in the area was also carried on in other settings such as the work of Lowry and Medlock at Sun Oil Co. [36].

2.2.3 1970 — 1979

The use of dependence information to aid code optimization became a very active research topic in the early 1970s and has remained so to this day. Early in the decade Kuck et al. introduced the notion of recording data dependence information in a digraph [30], but for the remainder of the decade this idea took a back seat to exploration in the area of data flow analysis. Aho et al. [1] and Muchnick [38] provide excellent reviews of the literature at the end of Chapters 10 and 8 of those books respectively. The decade produced on the order of hundreds of research papers describing various aspects of data flow analysis research focusing on identification of opportunities for program optimization based on data flow analysis [4], issues related to interprocedural analysis [5, 46], improved efficiency [20, 29], the definition of data flow problems and their related solutions [6], algorithms for use at the source-level [45], complications due to variable aliasing [35], more powerful approaches to data flow analysis [28], general improvements to flow analysis algorithms [8], methods to deal with the huge amounts of information that exhaustive data flow analysis generates [9], application to other areas of computer science including secure information flow [16], and most important to this dissertation, the application of data flow techniques to software maintenance problems [18, 53].

2.2.4 1980 — 1989

The early 1980s saw increasing interest in the area of applying data flow techniques to software maintenance problems. The value of statically analyzing a program for errors that may not reveal themselves until the program has been executed grew in appeal as people realized that the early identification of programming errors would produce programs that were more reliable and less likely to exhibit unexpected behaviors. Program debugging was another area of expected benefit. Weiser's Ph.D. thesis of 1979 introduced the notion of "program slicing". Informally, a program slice is the subprogram that contains the set of statements that could affect the value of a variable at a specific statement in the program. The idea of a program slice is to extract just the lines of code that need be considered when trying to determine the fault associated with a program failure, thus reducing the effort required to locate the problem statement. Weiser used data flow techniques to identify slices. He worked in the area for the better part of a decade looking at applications for program slicing technology as well as ways to improve the efficiency and the precision of his algorithms. Literally hundreds of research papers have focused on program slicing since its introduction.

Also during the first half of the decade the idea of using a dependence graph to perform code optimizations resurfaced. Using dependence graphs provides a means for performing many optimizations more quickly than can be accomplished using data flow equations. The program dependence graph was first described by the Ottensteins [39], and was later shown to be a suitable representation for performing compiler optimizations by Ferrante et al. [17].

Before long, researchers became interested in formally evaluating the appropriateness of using program dependence graphs as a base for program evaluation. Horwitz et al. [25] proved that it is possible to define a program dependence graph that is adequate for faithfully representing the behavioral aspects of sequential, imperative programs with structured control flow. Their chief concern is showing that if two programs can be represented by the same program dependence graph they are equivalent. That is, if the two programs have the same initial state they either halt or diverge with the same final state. If this is a required property of a program dependence graph, then a distinction must be made between loop-independent and loop-carried data dependencies. We describe these briefly in Section 2.2.5.

Also during the second half of the decade, Podgurski and Clarke proved that the combined set of data and control dependencies is a superset of the statements in a program that could affect the execution behavior of a particular statement. They defined a graph-based model of program dependencies for use in software maintenance activities [41, 42]; for instance, the identification of semantic errors in programs such as the use of an incorrect operator in an arithmetic statement.

The PDG and the Podgurski and Clarke models of dependencies are very similar. Each is based on combining control and data dependencies. Data dependencies are identified through the use of reachability analysis and control dependencies are identified through the combination of control flow and forward dominance information. The Podgurski and Clarke model is described in detail in Section 2.3.2.

2.2.5 1990 — 1999

With Podgurski and Clarke's formal model of intraprocedural dependencies in place, dependence analysis research in the 1990s turned to addressing challenges in a more broad class of programs as well as to developing dependence-graph-based software maintenance tools [44, 37, 10]. Table 2.1

Table 2.1: **Dependence Analysis Research in the 1990s.**

References	Multi-proc	Obj-Orient	Concurrent	Con-OO	Reactive
[11]					
[14]			X		
[15]					X
[21, 31, 22, 47]	X	X			
[23]				X	
[25, 27]	X				
[33]	X				
[37]	X				
[49]					X
[55, 56]				X	X

Table 2.2: System Types and Related Dependence Relationships.

Reference	Dependence Type	Uni-proc	Multi-proc	OO	Con	Con-OO	Reactive
	Control	X	X	X	X	X	X
	Data	X	X	X	X	X	X
[41]	Strong Control	X	X	X	X	X	X
[41]	Weak Control	X	X	X	X	X	X
[41]	Strong Syntactic	X	X	X	X	X	X
[41]	Weak Syntactic	X	X	X	X	X	X
[41]	Semantic	X	X	X	X	X	X
[23]	Divergence	X	X	X	X	X	X
[27]	Call		X	X	X	X	X
[27]	Parameter		X	X	X	X	X
[22]	S-control		X	X	X	X	X
[14]	Selection				X	X	X
[14]	Synchronization[1]				X	X	X
[14]	Communication				X	X	X
[14]	Interference				X	X	X
[23]	Synchronization[2]				X	X	X
[23]	Ready				X	X	X

provides references to works that address issues in a variety of types of programs. These approaches are generally based on either the PDG, the Podgurski and Clarke model of dependencies, or a combination of the two. The primary focus for the decade was the extension of dependence analysis definitions to include the more complex communication mechanisms available to programmers in these richer languages. These approaches generally involved defining additional types of dependencies among program statements and adding new types of arcs and various types of dummy vertices to the PDG in order to represent the additional dependencies.

The 1990s saw major expansion in the types of programs targeted by dependence analysis algorithm designers. Table 2.2 contains a list of dependence types, the research papers in which they are defined, and the indications of the types of programs in which the particular type of dependence is encountered. A listing of these definitions, including brief characterizations of their meaning, is given below.

The formal definitions of the dependence types listed in Table 2.2 depend on many variations of basic graph theoretic terminology; therefore, rather than present the definitions given in the various works, we present characterizations so that the reader may gain an intuitive idea of the meaning of each. In order for the reader to get a feel for what they mean in terms of a program, the characterizations are given in terms of program statements instead of the vertices in a CFG.

Dependence Type 1 *A statement* s_2 *is* directly control dependent *on a statement* s_1 *if* s_1 *can affect the number of times* s_2 *is executed and no sequence of statements leading from* s_1 *to* s_2 *contains a statement on which* s_2 *is control dependent. A statement* s_2 *is* control dependent *on a statement* s_1 *if* s_1 *is in the backwards transitive closure of direct control dependencies beginning at* s_2.

Dependence Type 2 *A statement* s_2 *is* directly data dependent *on a statement* s_1 *if there is a walk from* s_1 *to* s_2, s_2 *uses a value* v *that is defined at statement* s_1, *and* v *is not redefined along the walk from* s_1 *to* s_2. *A statement* s_2 *is* data dependent *on a statement* s_1 *if* s_1 *is in the backwards transitive closure of direct data dependencies beginning at* s_2.

Several types of data dependence have been distinguished in order for the dependence graph to provide adequate information to solve specific problems. These include output and anti dependence [30], and def-order, loop-independent, and loop-carried [25].

Dependence Type 3 Strong control dependence *is the same as control dependence as described above.*

Dependence Type 4 *Statements following a loop header are* weakly control dependent *on the loop header. This dependence type recognizes that the exit condition of a loop may delay execution of a statement indefinitely. A statement* s_2 *is directly weakly control dependent on a statement* s_1 *if* s_2 *is weakly control dependent on* s_1 *and is not weakly control dependent on any statements on a walk from* s_1 *to* s_2.

Dependence Type 5 *A statement is* strongly syntactically dependent *on a statement* s_1 *if there is a chain of data and/or strong control dependencies from* s_2 *to* s_1.

Dependence Type 6 *A statement* s_2 *is* weakly syntactically dependent *on a statement* s_1 *if there is a chain of data and/or weak control dependencies from* s_2 *to* s_1.

Dependence Type 7 *A statement* s_2 *is* semantically dependent *on a statement* s_1 *if the function computed by* s_1 *affects the execution behavior of* s_2*; that is, if the value computed at* s_2 *or the number of times* s_2 *is executed can be affected by the function computed at* s_1.

Dependence Type 8 Divergence dependence *is the same as weak control dependence except that it is defined less rigorously and not in terms of strong forward dominance.*

Dependence Type 9 *A procedure entry is* call dependent *on statements that contain calls to that procedure.*

Dependence Type 10 Parameter dependencies *exist between actual and formal parameters. Parameter-in dependencies are the dependence of the formal parameter on the value passed in during a procedure call. Parameter-out dependencies are the dependence of variables used after a procedure call in which the variable may be redefined and returned to the calling procedure.*

Dependence Type 11 S-control dependence *relates control dependencies of statements in multiple procedure programs to the control dependencies of sets of vertices of an inlined control flow graph representation of the program. A statement s_1 is said to be s-control dependent on another statement s_2 if any vertex representing statement s_1 in the inlined interprocedural control flow graph is control dependent on any vertex representing s_2.*

Dependence Type 12 Selection dependence *is similar to strong control dependence but differs in that rather than depending on a decision in a conditional statement in sequential, the selection depends on non-deterministic choice in a concurrent program.*

Dependence Type 13 *If s_1 and s_2 are two statements in a concurrent program and the start (termination) of s_1 determines whether s_2 starts (terminates), then s_2 is* synchronization dependent *on s_1 according to the first definition of synchronization dependence given in the table.*

Dependence Type 14 *If s_1 and s_2 are two statements in different processes, then s_2 is* communication dependent *on s_1 if the value v defined at s_1 can reach s_2 by way of interprocess communication and v is used at s_2.*

Dependence Type 15 *Given threaded program $\mathcal{P}$, there is an* interference dependence *between two statements s_1 and s_2 if s_1 and s_2 are in different threads and access the same variable.*

Dependence Type 16 *If m_1 and m_2 are entry and exit monitors of a critical region CR, and s is a statement enclosed in CR, s is* synchronization dependent *on m_1 and m_2 according to the second definition of synchronization dependence listed in the table.*

Dependence Type 17 *A statement s_1 is* ready dependent *on another statement s_2 if the failure of s_2 to execute indefinitely delays the execution of s_1. For example, if s_1 is reachable from s_2 and s_2 is a wait then s_1 is ready dependent on s_2. The authors claim this is a cost effective alternative to interference dependence for gaining precision when slicing Java programs.*

The fact that this list of dependencies contains duplicate as well as conflicting definitions is a by-product of the immaturity of the research area and points to the need for increased rigor and improved generality of these solutions to specific problems.

2.3 Related Topics

In this section we present descriptions of three areas of research that are fundamental to the work described in this dissertation: forward dominance, also known as postdominance and inverse dominance; the Podgurski and Clarke model of program dependencies that we use as a base for our definitions and theorems; and the program call graph, which is used in conjunction with information about dependencies in individual procedures to compute program-wide dependence information.

2.3.1 Forward Dominance (a.k.a. postdominance or inverse dominance)

Computing the forward dominators in a control flow graph is equivalent to computing dominators in its reversed graph, that is the graph that results from reversing the direction of all arcs in the control flow graph. The dominance relationship among vertices of a rooted digraph can be used to automatically identify loops in a program based on the program's language-independent control flow graph representation. It is therefore used in algorithms for performing many types of program analyses. Informally, one vertex u in such a graph dominates another vertex v if every path from the entry to the graph to v includes u. When a path beginning at a vertex v in a control flow graph contains a dominator u of v then v is contained in a loop. An immediate dominator of a vertex v is the first dominator encountered during a reverse traversal over the control flow graph beginning at v. If a vertex u is the immediate dominator of two different vertices, v and w, then it is a branch statement and v and w lie on different paths originating at u. A formal definition of dominator can be found in Appendix B.

In this dissertation we are concerned with a variation of the dominance relationship called forward dominance. Forward dominance applies to digraphs that have a single point of exit. A reverse graph of a digraph G is referred to as the inverse graph of G and is denoted G^{-1}. This is the source of the alternative term, "inverse dominance" that is used by some researchers [54]. Informally, a vertex v in a single exit digraph forward dominates a vertex u if every path from u to the graph's exit vertex includes v. This concept is important to dependence analysis algorithms because forward dominators identify joins in a program, and the lack of forward dominance is used to identify control dependencies.

As examples of dominance and forward dominance relationships, referring again to the control flow graph shown on the right in Figure 2.1:

- Vertex 2 is the immediate dominator of vertices 3 and 4.
- Vertex 4 is a dominator of vertices 5, 6, and 7 but not of vertex 3.
- Vertex 4 also forward dominates vertices 5, 6, and 7.
- Vertex 8 is the immediate forward dominator of vertices 2, 3, and 4.

2.3.2 Podgurski and Clarke Dependence Model

A guiding principle for the model developed in this dissertation is that the model be rigorously defined. Podgurski and Clarke provide a formal model of dependencies for programs that can be represented by traditional control flow graphs. The rigorous development of that model provides a solid foundation for the model developed here. The structure of the Podgurski and Clarke model is similar to that depicted in Figure 1.1. In addition to the structures shown in this figure, the model contains definitions of important graph theoretic terms that are useful when discussing dependence analysis. These are reproduced in Appendix B. Additionally the Podgurski and Clarke model distinguishes two types of forward dominance, forward dominance and strong forward dominance, and distinguished two types of control dependence, strong and weak, based on these. The purpose of making these distinctions is to recognize the fact that the execution of statements in a program that follow loops is dependent on the exit condition of the loop. In other words, a statement following a loop will never be executed if the program's execution ends up looping infinitely many times because of an error in specifying the exit condition of the loop. Podgurski and Clarke provide formal graph theoretic definitions for these terms.

Definition 1 *[42] Let G be a control flow graph. A vertex $u \in V_G$* strongly forward dominates *a vertex $v \in V(G)$ if and only if u forward dominates v and there is an integer $k \geq 1$ such that every walk in G beginning with v and of length $\geq k$ contains u.*

Definition 2 *[42] Let G be a control flow graph, and let u,$v \in V_G$. Vertex u is* strongly control dependent *on vertex v if and only if there exists a $v{-}u$ walk vWu in G not containing the immediate forward dominator of v; this walk is said to demonstrate that u is strongly control dependent on v. The strong control dependence relation on V_G is denoted by $\xrightarrow{scd_G}$: (u,v) in $\xrightarrow{scd_G}$ if and only if u is strongly control dependent on v.*

Let G be a control flow graph, and let $u, v \in V_G$. Vertex u is directly strongly control dependent *(or dsc-dependent) on v if and only if there is a walk vWu in G such that both of the following are true:*

- *vWu demonstrates that u is strongly control dependent on v and*
- *u is not strongly control dependent on any vertex of W.*

The walk vWu is said to demonstrate that u is directly strongly control dependent on v.

Definition 3 *[42] Let G be a control flow graph, and let $u, v \in V_G$. Then u is* directly weakly control dependent *(or dwc-dependent) on v if and only if v has successors v' and v'' such that u strongly forward dominates v' but does not strongly forward dominate v''; u is* weakly control dependent *(or wc-dependent) on v if and only if there exists a sequence, $v_1, v_2, ..., v_n$, of vertices, where $n \geq 2$, such that $u = v_1$, $v = v_n$, and v_i is directly weakly control dependent on v_{i+1} for $i = 1, 2, \ \ldots \ , n-1$.*

The combination of strong/weak control dependencies with data dependencies is defined to be strong/weak syntactic dependence, which can be represented as the dependence graph at the bottom of Figure 1.1. In addition to these definitions, Podgurski and Clarke provide a formal definition of semantic dependence that describes the conditions under which one statement of a program has the potential to affect another statement. This definition is quite complex and is provided in Appendix B along with other definitions of the model. Semantic dependencies may be classified as being "finitely demonstrated" in which case they can be conservatively identified using strong control dependence algorithms.

A semantic dependence relationship between two statements in a program is required in order for one statement to affect another. While it is not possible, in general, to know if the semantics of one statement can affect another, it is possible to compute sets of syntactic dependencies. Podgurski showed that a syntactic dependence must exist between two statements if a semantic dependence exists and, as such, provides a conservative, though imprecise, approximation of semantic dependencies.

2.3.3 Program Call Graph

It is useful for many types of interprocedural analysis to build a structure known as a program call graph. A call graph represents the calling relationships among the procedures of a program.

Definition 4 *A* call graph *for a given program $\mathcal{P}$ is a digraph $\mathcal{CG} = \{V, A\}$ where V is a set of vertices, one for each procedure in the program; and A is a set of arcs $\{a_1, \ldots, a_n\}$ where each $a_i =$ (P_1,P_2) for some P_1,$P_2 \in \mathcal{P}$ and the procedure represented by P_1 contains one or more calls to the procedure represented by P_2. The arcs in the call graph may be annotated with call site identifiers for its associated calls if that information is deemed useful to the analysis for which the call graph is to be constructed. We call this an* annotated call graph.

In the absence of procedure aliasing it is simple to construct a call graph by scanning the code of each procedure, constructing vertices and arcs as needed when calls are encountered in the scanning process.

In this dissertation we assume a distinguished procedure associated with each program that represents the main procedure of the program. The vertex in the call graph associated with this procedure has zero in-degree. We also assume the arcs of the call graph are annotated with unique call site identifiers.

Loops in a program call graph indicate the presence of recursive procedure calls. Sets of procedures that are involved in a recursive sequence of calls can be identified through the application of algorithms designed to identify strong components in digraphs. When strong components have been identified, the vertices of the call graph that belong to the strong component can be combined into a single vertex of what is referred to as the *contracted call graph.* There are many such algorithms available; we frame the discussion in Section 3.3.6 in terms of an algorithm recently developed by Gabow [19].

2.4 Pitfalls for Interprocedural Dependence Analysis

Interprocedural dependence analysis has been an active area of research throughout the 1990s. In this dissertation we consider the work of three major projects in this area:

- Loyall and Mathisen [37] focus on creating algorithms for identifying the potential for one procedure in a program to impact another.
- Horwitz et al. [27] focus on interprocedural slicing of structured programs.
- Harrold et al. [22, 47] focus on enhancing interprocedural dependence algorithms for application to a wider class of programs.

Each of these projects centers on calculating dependencies based on some form of interprocedural control flow representation in which a given procedure is represented one time in the program's control flow graph and arcs are inserted in the graph to connect the call and return of the procedure to each of its call sites. A common theme among these works is that there are problems associated with this type of representation that create difficulties in identifying conservative sets of dependencies using traditional definitions of control and data dependence. We describe these projects in detail in Chapter 6.

In this section we describe several issues that have been raised by researchers working in this area; we call these the "pitfalls of interprocedural dependence analysis." We use the program Sum to illustrate the effects of these pitfalls as a running example through this dissertation. This example is borrowed from Harrold et al. [22]. Pseudo code for Sum is shown in Figure 2.4. This example allows us to discuss several pitfalls that one encounters when developing control dependence algorithms based on non-inlined control flow representations. We briefly describe them below.

2.4.1 Valid Walks

A valid walk in the graph is a digraph walk that represents a potential execution of the program. Assuring that a walk is valid requires that an algorithm ensure that traversals over the control flow graph contain sequences of vertices that represent valid execution paths in the program. Loyall and Mathisen [37] described an interprocedural control flow graph in terms of a collection of control flow graphs where each call site is connected to the entry and return vertices of the called procedure by call and return arcs. When using this view it is possible to traverse the arcs of the graph in a way that does not represent any potential execution of the program because a procedure entry may be incident from many call arcs. Return vertices may be incident to many arcs, each of which is associated with one call site; when exiting a procedure's control flow graph while performing a forward traversal, the one chosen for return must match the current call. The choice of any other return arc would not represent any potential execution of the program; these are referred to as invalid walks. Referring to the example of Figure 2.4, when returning from procedure `B` it must be possible to determine whether the appropriate return is with respect to the call at statement 4 of procedure `M` or the call at statement 5. Loyall and Mathisen provide a definition for "interprocedural

```
    proc M
1:  read i, j
2:  sum = 0
3:  while i < 10 do
4:     call B
    endwhile
5:  call B
6:  print sum
7:  return

    proc B
8:  call C
9:  if (j>= 0) then
10:   sum = sum + j
11:   read j
    endif
12: i = i + 1
13: return

    proc C
14: if (sum > 100) then
15:   halt
    endif
16: return
```

Figure 2.4: Pseudo Code for Program Sum.

walk" that assures only valid walks are considered. Horwitz et al. deal with this issue through the use of an attribute grammar [27].

2.4.2 Calling Context

Both control and data dependencies within procedures depend on those that exist at call sites. Additionally, control and data dependencies of statements to be executed upon return from the procedure call may exhibit dependencies on statements that are executed as a result of the procedure call, or may in fact cross the procedure call boundary and return to the calling procedure. It is this potential that requires that only valid paths be traversed when identifying dependencies using a non-inlined interprocedural control flow representation. Harrold et al. [22] discuss the fact that identification of control dependencies can be masked by the existence of multiple call sites when one of the call sites forward dominates a conditional statement and another does not.

The Loyall and Mathisen definition of interprocedural control dependence correctly accounts for the calling context. However, their definition of direct interprocedural control dependence suffers from the effect described by Harrold et al. The definitions are as follows:

Definition 5 *Let $\mathcal{G}$ be an interprocedural CFG and let G_i and G_j be CFGs in $\mathcal{G}$. Let $u \in V_{G_i}$, and $v \in V_{G_j}$. Node u is* directly strongly control dependent *on v if and only if v has successors v' and v'' such that u forward dominates v' but does not forward dominate v''. Node u is* strongly control dependent *on v if and only if there exists a v-u interprocedural walk not containing the immediate forward dominator of v.*

The primary difference between this definition and Definition 2 is the use of the term "interprocedural walk". Prior to providing the definition of strong interprocedural control dependence, Loyall and Mathisen defined "interprocedural CFG" as we described above as well as well as an extended definition of digraph walk for application to interprocedural control flow graphs. The interesting thing to note about this Definition 5 is that the definitions of direct strong control dependence and strong control dependence are independent. In fact, the definition of direct strong control dependence is based on the Ferrante et al. [17] definition of control dependence, not the definition given by Podgurski in his thesis [42]. However, the definition of strong control dependence is based on the Podgurski and Clarke [41] definition. In Podgurski's thesis the definition of direct strong control dependence is given as:

Definition 6 *Let G be a control flow graph, and let u, v in V_G. Vertex u is* directly strongly control dependent *(or dsc-dependent) on v if and only if there is a walk vWu in G such that both of the following are true:*

- *vWu demonstrates that u is strongly control dependent on v and*
- *u is not strongly control dependent on any vertex of W.*

In his thesis, Podgurski proves that this definition is equivalent to the Ferrante et al. definition. This proof is based upon using an inlined control flow graph representation. The difference between an extension of this definition and the extension of the Ferrante et al. definition is that this one is given entirely in terms of the absence of an immediate forward dominator.

Harrold et al. pointed out that if one uses Definition 5 above to determine control dependencies in the code of Figure 2.4 one finds that statements in procedure `B` are not identified as being control dependent on statement 3 even though they should be due to the call at statement 4. This is because all execution paths in the program beginning at statement 3 eventually execute statement 5 and consequently the statements in procedure `B`.

They call this condition the "multiple context effect" and distinguish it from the "calling context effect". In our view both effects are related to recognizing the need to consider the dependencies of the call site when identifying dependencies of statements that may subsequently be encountered in an execution trace of a program. Taking this view allows us to identify dependencies equivalent to those identified using an inlined control flow representation.

We recognize that if, however, one applies an extension to Definition 6 that says that a vertex v is directly control dependent on another vertex u if u is control dependent on v and not on any other vertex on a valid walk from v to u in the interprocedural CFG, then the statements in procedure B would be identified as being control dependent on statement 3. In this case the valid walk in the interprocedural CFG, 3-4-8, does not include statement 5, which is the immediate forward dominator of statement 3. According to Definition 5, statement 8 is strongly control dependent on statement 3. Using the idea we propose at the beginning of this paragraph, statement 8 is directly strongly control dependent because statement 8 is not strongly control dependent on statement 4.

This was a key insight leading to the development of the model described in Chapter 3. The absence of forward dominance was useful in determining interprocedural control dependencies, but its presence, as traditionally viewed, was not. We recognized that the notion of forward dominance does not apply to interprocedural control flow representations in the same way that it does to inlined control flow representations.

2.4.3 Potential for Non-Return

There are three reasons why control may not return after a call to a procedure.

- *Embedded halt* — The existence of an embedded halt within a called procedure or its descendents creates potential for non-return to the calling procedure. This situation results in control dependencies of vertices within the calling procedure on the called procedure or its descendents.

- *Non-termination of loops* — As we discussed in Section 2.1, it is sometimes useful to consider the potential that loops may not terminate due to errors in their exit conditions. Naturally, if a loop in a called procedure or one of its descendants in the call graph does not terminate, then control will not return to the calling procedure.

- *Infinite recursion* — If a called procedure or any of its descendents is involved in a recursive sequence of calls, then there is potential for infinite recursion and the possibility of non-return from a call to that procedure.

The first of these is addressed by Harrold et al. [22]; to our knowledge the second and third have not previously been addressed in the literature.

2.4.4 Incomplete Programs

Performing dependence analysis on multi-procedure programs that perform calls to procedures that exist in the form of precompiled libraries provide a challenge to interprocedural dependence analysis because it is not possible to determine control and data dependencies that may result from calling such procedures. Normally, a conservative approach is taken to data dependence analysis. It is assumed that all global variables and all in/out parameters are modified during the procedure call. It is also assumed that control returns from calls to such procedures.

2.4.5 Function Pointers

Function pointers or indirect procedure calls create problems because it is not possible to know statically exactly which procedure will be called. This effect is similar in nature to that of variable aliasing and as is the case when considering aliases, target programs are either restricted from using function pointers or a conservative approach is taken and all possible procedures are considered targets of indirect procedure calls.

Any model of interprocedural dependencies must provide support for addressing these pitfalls when designing dependence algorithms based on the model.

We take a compositional approach to the identification of program dependencies. We have rigorously defined a language-independent model that addresses these difficulties. The model supports development of algorithms intended to identify dependencies among procedures that are created in isolation and in cases where access to the source code may be restricted such as is often the case with precompiled libraries. The following two chapters of this dissertation contain a description of our new approach to interprocedural program dependence analysis, a set of algorithms for constructing the structures of the model, and an evaluation of the complexity of the algorithms.

Chapter 3

The Model

This chapter contains a description of a model of interprocedural dependence analysis. The model supports the identification of intraprocedural dependencies in isolation; at program composition time the interprocedural implications of the intraprocedural dependencies are computed. This model avoids pitfalls described in Section 2.4 that are associated with performing dependence analysis using one-to-one control flow graph representations of sequential, imperative programs that contain procedure calls.

As described in Chapter 1 the guiding principles followed during the design of this model were that it be a rigorously defined, language independent, and compositional. To motivate our decision to follow these principles we begin with a discussion of the underlying cause of the difficulties described in Chapter 2. We then define interprocedural control dependence and present a graph-based model that supports identification of such dependencies that does not suffer from these difficulties. In Chapter 4 we present a set of algorithms for constructing the structures that comprise the model and present an analysis of their complexity.

3.1 The Meaning of Control Dependence Among Procedures

A key insight upon which the thesis of this dissertation is founded is that defining the meaning of a semantic relationship among elements of a system is the most fundamental aspect of developing a model of dependencies for a given type of system. More generally stated, a semantic dependency exists between two elements of a system when there is potential for one element to affect another in some way. Before one can begin to develop dependence analysis algorithms one must answer two questions: first, what type of system element is to be the target of the analysis; and second, what types of relationships among the elements constitute an "affects" relationship. The answer to the second question will be based on both the type of systems being analyzed and the purpose of the analysis.

Consider for example a question about the potential impact of reordering instructions when performing code optimization. Dependencies must be calculated considering not only their ordering within the code but also considering latency associated with execution of instructions that occur on paths between them [38]. However, when the objective of the analysis is to determine the affect of changing the condition associated with a decision statement, one need not be concerned with low-level details such as instruction latency. The level of abstraction at which dependence analysis is performed should be appropriate to the purpose for which dependencies are being identified.

In this discussion, we are interested in identifying the potential for the calculation made in one program statement to affect the calculation made in another statement. The transitive closure over control and data dependencies beginning at a particular statement will contain a superset of semantic dependencies [41] given finite execution of loops in the program.[1] This dissertation focuses on identifying control dependencies in sequential, imperative programs that contain procedure calls. Previous work in this area has concentrated on computing dependencies based on one of two forms of a super control flow graph: either an inlined interprocedural control flow graph that contains a copy of each procedure's control flow graph at each call site or a one-to-one interprocedural control flow graph that contains just one copy of a procedure's control flow graph that is connected to all call sites via call and return arcs. Our approach differs from these in that dependencies are identified in a compositional manner; dependence information for each procedure is calculated in isolation then combined at program composition time to identify interprocedural dependencies. This is a unique approach to interprocedural summary analysis used by Fosdick and Osterweil [18], that is appropriate for use in programs that are composed of procedures that are created in isolation and is particularly useful in the case that the procedure's author wishes to conceal the procedure's source code.

The following definition of control dependence applies to sequential, imperative multi-procedure programs and supports identifying dependencies useful in software maintenance activities such as impact analysis and regression testing.

As noted earlier, interprocedural communication among statements in a program creates several challenges for the dependence analyst. These challenges fall into two areas: consideration of the effects of calling context and failure to return from a procedure call. These result in very different types of dependence-related problems:

- Ignorance of calling context results in identification of unnecessarily imprecise sets of dependences.

- Ignorance of the potential for non-return from procedure calls introduces the possibility of creating non-conservative sets of dependences.

The definition of interprocedural control dependence presented here reflects the recognition of these two concerns.

[1] Podgurski and Clarke define *strong* and *weak* control dependence. Our use of the term control dependence refers to strong control dependence unless otherwise specified.

3.1.1 Sources of Interprocedural Control Dependence

Informally, if u and v are statements in a multi-procedure program $\mathcal{P}$, $v \in P$ where $P \in \mathcal{P}$, is control dependent on u if the number of times v is executed with respect to the same invocation of P or the number of times P is invoked can be affected by a decision made at u.

The first condition for control dependence is the same condition that applies to uni-procedure programs. However, when this condition is considered for multi-procedure programs, then the control dependence of v may rest on a statement in another procedure; for example, if v follows a procedure call from inside a while loop, then its execution depends upon the resumption of P after the procedure call. The second condition is equivalent to considering a procedure call to be an entry to a region of code in which all statements are dependent on the condition that determines whether entry to the region is executed. We call these two types of dependence *resumption* and *invocation* dependence and provide definitions for them below. Additionally, we define a third type of control dependence, *potential control dependence* that is required due to the compositional nature of our model. Potential control dependence provides a means for refining certain intraprocedural control dependencies during the interprocedural control dependence identification stage of analysis.

Before we proceed to define interprocedural control dependence, we define each of these types of control dependence. For illustrative purposes we introduce Figure 3.1 containing pseudo code for two procedures that comprise a program `Compute_tax`. In addition to the code associated with the two procedures we add lines indicating that there is a single start and a single stop for the program. These are the lines labeled `S` and `F`. These are added because we consider the entry to procedure `M` to be control dependent on the `Start` of the program. The procedure on the left of the figure is the main procedure for the program. The amount of tax depends on whether the purchase is for more or less than \$10. The procedure on the right is called from `Compute_tax` to perform the multiplication; `mult` takes one in/out parameter `num`. In the following paragraphs we describe the types of control dependence in terms of the statements of `Compute_tax`.

Potential Control Dependence

Potential control dependence indicates the potential to inherit the control dependence of a call statement. A potential control dependence is converted to a control dependence if, after program composition, it is determined that control is certain to return to a procedure after a call at that call statement, otherwise it is converted to an inherited control dependence. Referring again to Figure 3.1, if the code for `mult` were not available it would not be possible to tell for certain whether statements 7 and 8 are control dependent on statement 6. Therefore, when creating a procedure control dependence graph for `Compute_tax` in isolation we say that statement 8 is potentially control dependent on statement 6. The actual control dependence depends on the structure of the called procedure, `mult`.

```
S: Start Compute_tax
   proc M                     proc mult(num)
1: read price            11: rate = 1.06
2: if price < 10         12: if (num > 0) then
3:   print "Low Tax"     13:   num = rate*num
   else                       else
4:   call mult(price)    14:   halt
5: print price           15: return
6: call mult(price)
7: print price
8: return
F: Finish Compute_tax
```

Figure 3.1: Example Pseudo Code for Program `Compute_tax`.

Resumption Control Dependence

If it can be determined that there is potential for non-return from the called procedure, then the control dependence of statements within the calling procedure that follow the procedure call is inherited from that of the return statements of the called procedure. We call this kind of control dependence *resumption control dependence*. Statement 15, the return from `mult`, is control dependent on statement 12 because statement 14 is a halt; therefore, statements 5, 6, 7, and 8 are resumption dependent on statement 12.

Invocation Control Dependence

If there is a statement in a procedure that is not control dependent on any decision within the procedure or a called procedure, then that statement will inherit direct control dependencies from each call site from which its enclosing procedure can be called or from the start of the program in the case that its enclosing procedure is the program's main procedure. We call this *invocation control dependence* because the execution of the statement depends on whether or not its enclosing procedure is invoked. As examples, statements 1, 2, and 5 are invocation control dependent on S. Because the call to `mult` at statement 4 is control dependent on statement 2, statements 11 and 12 are invocation control dependent on statement 2. Statements 5, 6, 7, and 8 are resumption control dependent on statement 12 and statements 11, 12, and 15 are invocation control dependent on statement 12 because they inherit the control dependence of the call at statement 6 which is control dependent on statement 12 by way of the call at statement 4.

Each of these types of control dependence is formally defined below.

Definition 7 *Given program $\mathcal{P}$ and a procedure $P \in \mathcal{P}$, a statement x and a sequence of consecutive statements $Y = y_1, y_2, \ldots, y_n$ such that:*

- *$x \in P$ is a call statement,*
- *y_1 is the first statement to be executed if control returns to P from the call at x, and*
- *for every $y_i \in Y$, y_i is not a conditional statement or a call statement,*

we say that each $y_i \in Y$ is directly potential control dependent *on x. This relationship is denoted $y_i \xrightarrow{dpcd} x$. y_i is* potential control dependent *on x if and only if there is a chain of direct potential control dependencies $s_1, \ldots, s_n$ such that $s_1 = y$, $s_n = x$, and $s_{i-1} \xrightarrow{dpcd} s_i$, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{pcd} x$.*

Definition 8 *Given program $\mathcal{P}$ and two procedures $P_1, P_2 \in \mathcal{P}$, statements w, x, and y such that:*

- *$w \in P_1$ is a call statement to P_2,*
- *$x \in P_2$,*
- *$y \in P_1$ and $y \xrightarrow{dpcd} w$,*

if any return statement in P_2 is control dependent on x, y is directly resumption control dependent *on x. This relationship is denoted $y \xrightarrow{drcd} x$. y is* resumption control dependent *on x if and only if there is a chain of direct control dependencies $s_1, \ldots, s_n$ such that $s_n \xrightarrow{drcd} s_{n-1}$ where $s_1 = y$, $s_n = x$, and $s_{i-1} \xrightarrow{drcd} s_i$, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{rcd} x$.*

Definition 9 *Given program $\mathcal{P}$ and two procedures $P_1, P_2 \in \mathcal{P}$ and statements x, y, and z in $\mathcal{P}$ such that:*

- *$x \in P_1$ is a call statement to P_2,*
- *either $y \in P_i$ for some $P_i \in \mathcal{P}$ and x is directly control dependent on y or y is $start^{\mathcal{P}}$ and P_1 is the main procedure of $\mathcal{P}$,*
- *z is the entry to P_2 or a statement in P_2 that is not otherwise control dependent on any vertex in $\mathcal{P}$;*

then z is directly invocation control dependent *on y. This relationship is denoted $z \xrightarrow{dicd} y$. z is* invocation control dependent *on y if and only if there is a chain of direct control dependencies $s_1, \ldots, s_n$ such that $s_1 = y$, $s_n = z$, and $s_{i-1} \xrightarrow{dicd} s_i$, $1 < i \leq n$. This relationship is denoted $z \xrightarrow{icd} y$.*

With these definitions in place it is now possible to present an extended definition of control dependence that is sufficient for describing control dependencies among statements of multi-procedure programs. When it is necessary to make a distinction between intraprocedural control dependence and this extended definition we refer to the former as intraprocedural control dependence and the following as interprocedural control dependence.

Definition 10 *Given program $\mathcal{P}$, a statement y in $P_1 \in \mathcal{P}$ is* directly control dependent *on another statement x in $P_2 \in \mathcal{P}$ if and only if one of the following conditions holds:*

1. *$P_1 = P_2$ and y is directly intraprocedurally control dependent on x, or*
2. *$y \xrightarrow{drcd} x$, or*
3. *$y \xrightarrow{dicd} x$, or*
4. *$y \xrightarrow{pcd} z$ in $\mathcal{P}$, $y \not\xrightarrow{rcd}$ on any statement in $\mathcal{P}$, and $z \xrightarrow{dcd} x$;*

and there exists a chain of statements $C = s_1, \ldots, s_n$ such that $s_1 = x$ and $s_n = y$, there is potential for control to flow directly from s_i to s_{i+1} for all $i \in \{1, \ldots, n\}$, and y is not control dependent on any s_i in C. Direct control dependence is denoted $y \xrightarrow{dcd} x$. y is control dependent *on x if and only if there exists a chain $c_1, \ldots, c_n$ such that $c_1 = y$, $c_n = x$, and $c_{i-1} \xrightarrow{dcd} c_i$, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{cd} x$.*

To illustrate the use of this definition, referring again to the code for program `Compute_tax` shown in Figure 3.1, condition 1 states that $3 \xrightarrow{dcd} 2$, condition 2 tells us that $5 \xrightarrow{dcd} 12$, condition 3 implies $12 \xrightarrow{dcd} 2$, and condition 4 tells us that the potential control dependence, $5 \xrightarrow{dpcd} 4$, is resolved at program composition time to identify $5 \xrightarrow{dcd} 12$. The fourth condition serves a dual purpose: first, it distinguishes direct and indirect dependencies, since it does not allow intervening control dependencies and, second, it permits multiple control dependencies to be defined for statement 5, which is not only directly control dependent on statement 12 but also on Start.

The goal of this dissertation is to provide automated support for identifying control dependence relationships among statements in a program. Traditionally, automated identification of control dependencies among program statements is accomplished by way of algorithms based on control dependence definitions stated in terms of the forward dominance relationships that exist among vertices in a control flow graph representation of the program's potential execution behavior. In a like manner we provide a graph-based definition of interprocedural control dependence called *compositional control dependence* in Section 3.3.5 that is based on an extension to the notion of forward

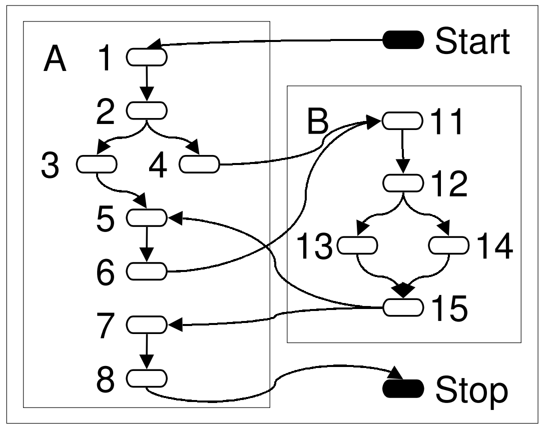

Figure 3.2: Example Interprocedural Control Flow Graph.

dominance. This extension allows us to compute control dependence relationships compositionally that are equivalent to those identified based on inlined control flow graph representations. As motivation for this extension, in the following section, we explore problems associated with identifying forward dominance relationships among statements of a program based on non-inlined control flow graph representations such as the one-to-one control flow graph representation introduced in Chapter 2. We follow this explanation with a description of our model and continue in Chapter 4 with a description of the algorithms for creating the structures associated with this model.

3.2 Forward Dominance and Interprocedural Control Flow

The difficulties described in Section 2.4 are due in part to the fact that forward dominance of one graph vertex over another, as applied to a non-inlined interprocedural CFG, does not represent the forward dominance relationship between the associated program statements. This is because a given statement in a procedure represents many statements in a program when the procedure's calling contexts are considered. Thus, the graph theoretic definition of forward dominance is not directly useful for determining the forward dominance relationships among statements of a program when these statements are represented in non-inlined control flow representations. We can see this most readily by studying the relationships among the vertices of Figure 3.2, in which we represent a program composed of two procedures, A and B. Procedure A contains calls to procedure B at statements 4 and 6.

Clearly, any execution of the program associated with this control flow graph would involve executing the statement associated with vertex 5 and thus, vertex 5 should be identified as a forward dominator of vertices in procedure B and vertices 1, 2, 3, and 4 of procedure A. However, even though the walk Start-1-2-4-11-12-13-15-7-8-Stop does not represent any potential execution trace for the program represented by the graph, it is a walk from 2 to the final vertex of the graph that does not include vertex 5, and therefore vertex 5 is not recognized by traditional forward dominance definitions as being a forward dominator of vertex 2. However, vertex 11 is recognized as being a forward dominator of vertex 2 since it is encountered on either path originating from vertex 2, and is in fact identified as the immediate forward dominator of 2. This fact then incorrectly causes traditional dependence analysis definitions to identify a dependence of vertex 5 on vertex 2 since there is a path from vertex 2 to vertex 5 (2,3,5) that does not include vertex 11, vertex 2's immediate forward dominator.

Although the notion of forward dominance is critical to most traditional definitions of control dependence, its limitations have not been addressed in previous work on interprocedural dependence analysis. We recognize this importance and extend the notion of forward dominance to represent the expected relationships among program statements. We call this extension the *forward dominance forest* and use it as part of our compositional, graph-based model of interprocedural control dependencies.

Our model is based on earlier work by Ferrante et al. [17] and by Podgurski and Clarke [41]. We have chosen to use the Podgurski-Clarke model as a foundation for our model for two reasons:

1. A detailed description of the model is available in Podgurski's doctoral dissertation [42].

2. It is a simpler model in that it does not include the notion of regions, but can be extended to include this optimizing feature of the Ferrante et al. model if desired.

The scope of applicability of the Podgurski-Clarke model is sequential, imperative, uni-procedure programs in which control flow may be unstructured. The definitions and theorems of the Podgurski-Clarke model that are used as the foundation for our model are included in the body of this dissertation when appropriate and are duplicated in Appendix B, which contains a dictionary of dependence-related terminology. Our model also leverages the experiences of researchers who have developed algorithms for identifying dependencies in multi-procedure programs; we described four of these projects in Chapter 2.

Podgurski and Clarke [41] state that one statement in a program is semantically dependent on another if a value that it uses or the number of times it is executed during a run of the program can be affected by the other statement.

Definition 11 *(Informal)*[2] *A statement s_2 is* semantically dependent *on a statement s_1 if the function computed by s_1 affects the execution behavior of s_2 — that is, if the value computed at s_2 or the number of times s_2 is executed can be affected by the function computed at s_1.*

It is the second condition of Definition 11 that is of concern in this dissertation. Control dependencies are used to identify when one statement has the potential to directly affect the number of times another statement is executed. The purpose for this concern with numbers of times that a statement is executed is to identify the potential for a variable assignment to be affected by an

[2]The formal definition is given by Podgurski and Clarke [41].

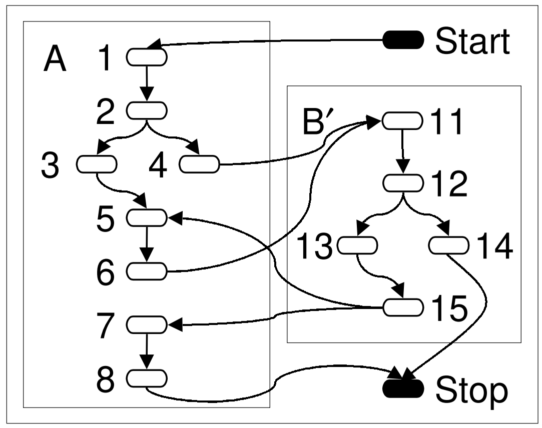

Figure 3.3: Example Interprocedural Control Flow Graph With a Halt.

evaluation of a conditional statement. This can happen either because the statement is avoided, such as is the case for assignments made along the paths in the body of if-statements or within loops that have conditional entry, or it can happen because execution of the statement is indefinitely delayed due to non-termination of a loop.

These concerns must be reflected in a definition of interprocedural control dependence. This is a point of confusion and was the subject of a related discussion by Harrold et al. [22] in which they note that the recognition or non-recognition of control dependencies between statements in a procedure and its calling procedures depends upon the definition of control dependence used. We say a statement is control dependent on another if a control dependence would be recognized when applying traditional control dependence definitions to an inlined control flow representation. If the statements are in different procedures, then we qualify the control dependence with respect to a given call site.

Consider the control flow graph shown in Figure 3.3. This graph is identical to that shown in Figure 3.2 except that the arc incident from vertex 14 now is incident to vertex Stop rather than vertex 15. This change represents the fact that the statement associated with vertex 14 is a halt. The existence of this halt statement produces control dependencies of vertices 5, 6, 7, and 8 of procedure A on vertex 12 within procedure B′ since the decision made at vertex 12 determines whether these vertices in procedure A will be executed. More interesting is the fact that this definition identifies vertex 11 as being control dependent on vertex 12. This is because the invocation of procedure B′ associated with vertex 6 is control dependent on the decision made at vertex 12 during B′'s invocation by the call at vertex 4. Thus, vertices within the same procedure can exhibit context-related interprocedural control dependencies among themselves.

3.3 The Structures

In review, the Podgurski-Clarke model of control dependencies that we are using as a foundation for our model describes the control dependence relationships among statements in a program in terms of control flow among the program's statements and the forward dominance relationships among vertices of a control flow graph. These relationships can be represented as a graph, thereby providing a base for automated identification of control dependencies. The control flow graph and the forward dominance tree are described in detail in Section 2.2.4. In this section we describe extensions to these structures for the purpose of representing the control-related relationships among statements of individual procedures independent of the context in which the procedure is called. We then describe how these structures can be combined at program composition time in order to determine dependencies in terms of calling contexts.

Figure 3.4 represents the model. A program is comprised of a set of procedures. Each procedure has an associated procedure control dependence graph (PCDG). The procedure control dependence graphs are computed using the procedure control flow graph (PCFG) in combination with the procedure forward dominance forest (PFDF). Once these structures are available they are used in combination with the call graph of the program to create the composed control dependence graph (CCDG). And then the control dependence graph (CDG) is created by tracing back from sources of inherited control dependencies to the first direct control dependence relationship encountered in a path in the CCDG.

Before we can provide a graph-based definition of interprocedural control dependence we must understand the effect of procedure calls on the control flow graph, we must understand what it means for one vertex of such a graph to forward dominate another, and we must understand how to identify paths in the graph that represent valid execution paths for the represented program. We describe and define each of these below.

3.3.1 Intraprocedural Control Flow

The control flow graph is the most basic program representation used in our model. It is in the building of the control flow graph that the language-dependent aspects of a program's code are interpreted and expressed in terms of vertices that represent executable statements and arcs that represent the potential for program execution to progress from one statement to another. Our model is based upon the ability to faithfully represent all language constructs within the confines of procedure control flow graphs, described below. Sequence, decision/junction, and loop are sufficient for representing all control mechanisms supported by commonly used sequential, imperative programming languages. We add procedure call/return to the list because we focus on non-inlined control flow representation. There may be language-specific differences in the order in which the control primitives are assembled to compose a specific mechanism, but the primitives listed have been proven sufficient for capturing the semantics of the associated mechanism [12, 34].

In this section we define the procedure control flow graph as an extension to a traditional control flow graph. Traditional definitions of control flow graphs are not sufficient to capture the semantics

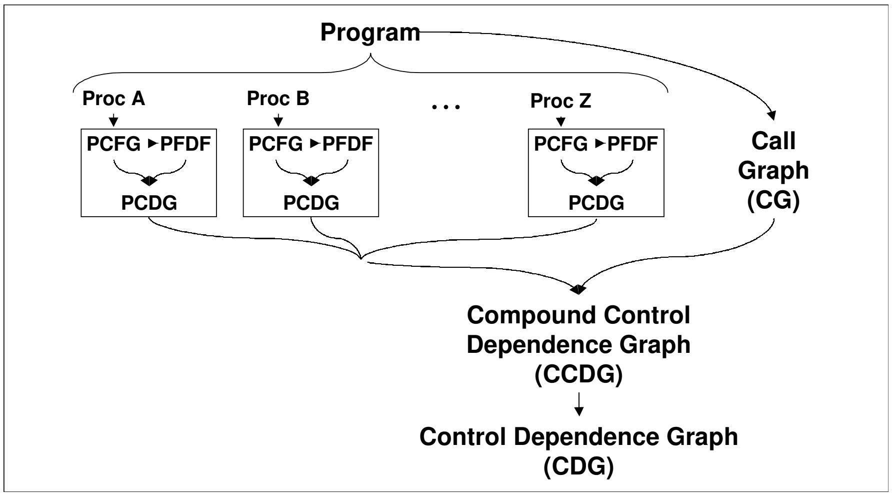

Figure 3.4: Compositional Model of Program Dependencies.

of control flow within procedures that contain procedure calls. This is because flow of control from call statements to their intraprocedural successors is interrupted by flow to the called procedure, which may result in non-return to the calling procedure due to the existence of embedded halts, infinite recursion, or non-terminating loops within the called procedure or its descendents in the call graph.

We extend the control flow graph with the addition of an *interrupted-flow arc*, and two distinguished vertices: an *initial vertex* v^I and an *annotated return vertex* v^R. The initial vertex represents the procedure declaration. The return vertex succeeds each return or explicit exception in the procedure. Embedded halts are vertices with zero out-degree that are not return vertices; their control flow successor is the program's global Stop vertex. Any other type of non-local jump is not representable in the procedure control flow graph. We do not consider this restriction to be a serious limitation of our model since the model is intended for application to programs composed of procedures that have no knowledge of the internal structure of the procedures with which they interact. Therefore, knowledge of jump targets in other procedures is not available and it is not possible to write procedures for use in this type of environment.

It is important to distinguish control flow among statements within a procedure that naturally leads from one vertex to the next from that which crosses procedure boundaries and, thus, may be affected by manipulations within the called procedure or, transitively, the procedures that it

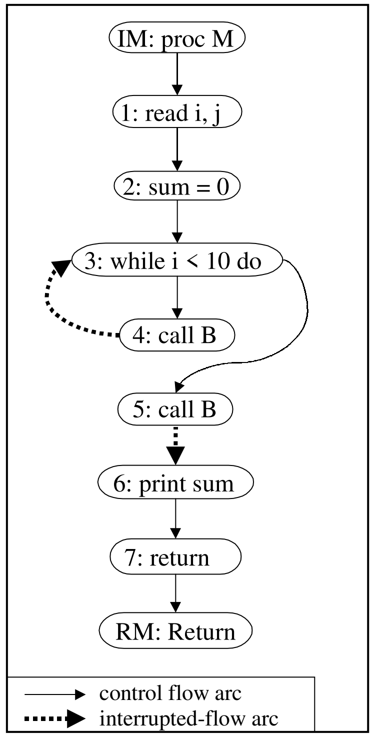

Figure 3.5: Example Procedure Control Flow Graph for Procedure M of Program Sum Shown in Figure 2.4.

calls. The former are called *direct control flow arcs* and are denoted (x,y) where x is the source of control flow and y is target. The latter are called *interrupted-flow arcs* (x,y) where x is a call to a procedure and y is the next statement to be executed should control return to the calling procedure. We make this distinction in recognition that resumption of the called procedure is dependent upon the behavior of the called procedure and the procedures it calls. Figure 3.5 shows the procedure control flow graph for the main procedure of the program Sum example introduced in Section 2.4. The code for Sum is shown in Figure 2.4. For sake of convenience, we display the code associated within each vertex. In Figure 3.5 the arcs from vertex 5 to vertex 4 and from vertex 6 to vertex 7 are interrupted-flow arcs.

Definition 12 *Given a procedure P, and statements $s, t \in P$, where s is a call statement, an* interrupted-flow arc *(u,v) is an arc incident from a graph vertex u representing s and incident to vertex v representing t. An interrupted-flow arc represents the expectation that, in the absence of abnormal termination, program execution will eventually resume at t after a call at s. u is called an* ifa-source *or* call vertex *and v is called an* ifa-target *or* resumption vertex.

Definition 13 *An* annotated vertex *is a vertex that contains zero or more predefined attribute/value pairs.*

Definition 14 *Given a procedure $P = \{s_1, \ldots, s_n\}$, a* procedure control flow graph *(PCFG) $G = (V, A)$ is a digraph where $V = \{v_1, \ldots, v_n, v^I, v^R\}$, $A = (F_G, I_G)$. V contains two distinguished vertices: an* initial vertex *v^I associated with invocation of the procedure, and an annotated* return vertex *v^R. F_G is a set of control flow arcs and I_G is a set of interrupted-flow arcs. For every vertex v with zero in-degree there exists a control flow arc (v^I,v) and for every vertex v_i that represents a procedure return, there exists a control flow arc $(v_i, v^R) \in F_G$. G satisfies each of the following conditions:*

- *in G there exists exactly one $v \in V$ to represent each $s \in P$.*
- *For every call statement in P there exists exactly one arc in I_G.*

The procedure control flow graph for the procedure M of program Sum shown in Figure 3.5 is composed of sequence, loop, and procedure call/return control primitives. In this PCFG $v^I = 1$, $v^R = Return$, and $I_G = \{(4,3), (5,6)\}$.

With these definitions in place we can proceed to describe interprocedural control flow and the composed control flow graph we use to represent the potential flows within and among procedures of a program.

3.3.2 Interprocedural Control Flow

During program execution, flow of control proceeds directly from a call site to the first executable statement in the called procedure. At the end of a procedure invocation, control can be transfered back to the calling procedure by way of normal return or exceptional return, or execution can jump directly to the program exit. Each of these situations results in identification of a different set of dependencies within the calling procedure. We now define an interprocedural control flow graph as a set of procedure control flow graphs connected by call/return arcs and jump arcs that connect non-returning exits from the procedure to the program exit.

Definition 15 *Given a program $\mathcal{P} = \{P_1, \ldots, P_n\}$, an* interprocedural control flow graph *is a digraph $\mathcal{G} = \{G_1, \ldots, G_n, v^S, v^F, C, R, J\}$ where*

- *there exists exactly one procedure control flow graph $G_i \in \mathcal{G}$ for each procedure $P_i \in \mathcal{P}$;*
- *v^S is a distinguished* start vertex *that has zero in-degree and represents the start of $\mathcal{P}$;*
- *v^F is a distinguished* finish vertex *that has zero out-degree and represents the stop of $\mathcal{P}$;*
- *C is a set of* call arcs *(u,v) where u is a call vertex in V_{G_i} and $v = v^I_{G_j}$ is the initial vertex for some procedure control flow graph G_j; $G_i, G_j \in \mathcal{G}$;*

- *R is a set of* return arcs *(u,v) where $u = v^R_{G_j}$ is the return vertex in procedure control flow graph G_j and $v \in V_{G_i}$ is also incident from an interrupted-flow arc in I_{G_i}; $G_i, G_j \in \mathcal{G}$*

that satisfies each of the following conditions:

- *There is a one-to-one correspondence between C and R. For each $c_i = (u, v^I_{G_j}) \in C$ there exists exactly one $r_i = (v^R_{G_j}, v) \in R$, and there exists (u,v) $\in I_{G_i}$.*

- *Every vertex of $V_{\mathcal{G}}$ occurs on some $v^I - v^F$ path.*

The last two conditions specify first, that each procedure must have at least one return statement and second, that any statement in the program that is syntactically recognizable as being unreachable is not represented in the control flow graph.

The interprocedural control flow graph $\mathcal{G}$ for Sum, the program introduced in Section 2.4 as our running example, is shown in Figure 3.6. In this example there are three control flow graphs: $G_1 = M$, $G_2 = B$, and $G_3 = C$. $v^I_{\mathcal{G}}$ is the vertex labeled "Start Sum" and $v^F_{\mathcal{G}}$ is the vertex labeled "Finish Sum", $C = \{(4, IB), (5, IB), (11, IC)\}$, $R = \{(RC, 12), (RB, 3), (RB, 6)\}$, and $J = \{(22, F)\}$.

Modeling Exceptions in an ICFG

Support for the definition and handling of exceptions in programming languages provides a challenge for dependence analysts due to the extensive implications for control dependence that result from the unpredictable nature of control flow paths associated with exception handling. Representation of control flow related to handling exceptions is a language specific activity associated with building a program's control flow graph. The definition for the procedure control flow graph provides support for modeling exceptional control in the form of the annotated return vertex. The return vertex may optionally contain the names of exceptions that can be triggered within the procedure. If exceptions are to be considered, then a dummy decision vertex is inserted as the ifa-target that tests the return to see if it is a normal or exceptional return. Arcs incident from the dummy vertex lead either to the vertex that represents the first statement within the procedure that should be executed upon return from the related call or, in the case of exceptional return, leads to a vertex that begins a further set of tests on the exception name. This can be represented similarly to a case statement defaulting with a jump to the procedure's return vertex and annotating it with exception information.

The semantics of exceptions vary considerably among languages that support them. For instance, in ML control is returned to the point of error after an exception is handled whereas in Ada the subprogram containing the exception handler is terminated immediately after the exception is handled. As mentioned earlier in this section, such language-dependent differences must be faithfully represented in the procedure control flow graph. We do not support, nor do we suggest attempting to model dependencies that result from the potential for implicit raising of exceptions such as "divide by zero" as this would result in creating control dependencies on virtually every statement in a program, immediately rendering the dependence analysis useless.

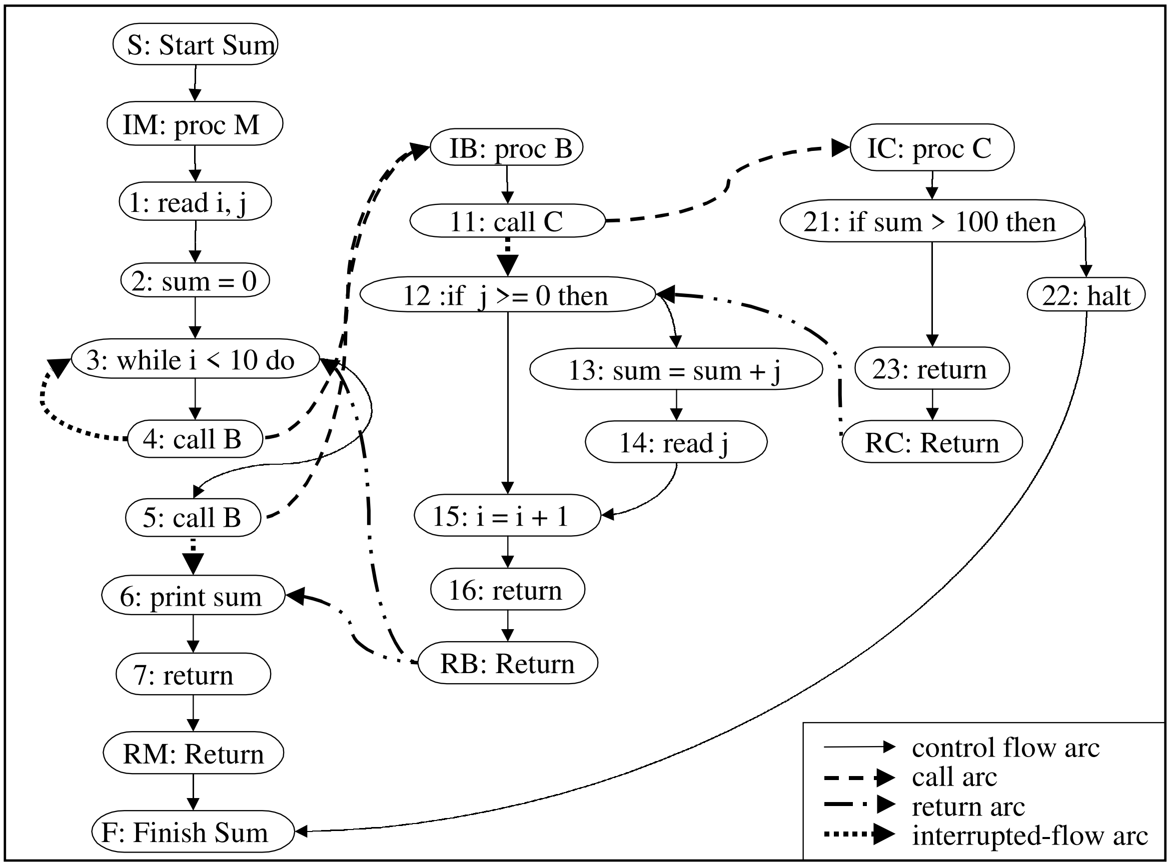

Figure 3.6: Interprocedural Control Flow Graph for Program Sum.

3.3.3 Interprocedural Walk

The calling context effect described in Chapter 2 is a result of the fact that there are paths in an interprocedural control flow graph that do not represent any executable sequence of program statements. While it is not possible to determine syntactically exactly which walks in a control flow graph represent valid execution sequences, it is possible to improve precision by illiminating walks that are syntactically recognizable not representing any potential execution. For example, referring to Figure 3.6, there is a path beginning at vertex 5, (5-IB-11-12-15-16-RB-3) that includes vertex 3. Clearly execution of statement 3 cannot follow that of statement 5 in any execution of Sum. This is an invalid walk of Sum's ICFG that results from the existence of more than one return arc from v_B^R. Each of these arcs is associated with exactly one call arc. In any execution of a program, the relationship between procedure calls and returns is like that of matching parentheses, with the call being the left parenthesis and the return being the right parenthesis.

Definition 16 *An* interprocedural walk $\mathcal{W}$ *in an interprocedural control flow graph* $\mathcal{G}$ *is a digraph walk that satisfies the following conditions:*

- *For any two vertices* v_i *and* $v_{i+1} \in \mathcal{W}$ *there does not exist* (v_i, v_{i+1}) *in* $I_\mathcal{G}$
- *Given any (c,r)* $\in I_{G_i}$, *where for some* $G_i \in \mathcal{G}$ *where* $c\mathcal{Y}r$ *is a subsequence of vertices in* $\mathcal{W}$, *then if for any vertex v in* $\mathcal{Y}$ *there exists an interrupted-flow arc (u,v) in* $\mathcal{G}$ *then* $u\mathcal{S}v$ *is a subsequence in* $\mathcal{Y}$ *for some sequence of vertices* $\mathcal{S}$.

Informally, the first condition of this definition prevents traversal over interrupted-flow arcs and the second condition says that if a walk from a call vertex of a procedure flow graph to its associated resumption vertex contains any other return arcs, these arcs must be preceded in the walk by associated call arcs. In our running example, this condition assures us that a walk beginning at vertex 4 will return to vertex 3 because traversal of return arc (v_B^R,6) is prohibited by the lack of existence of vertex 5 in the 4-v_B^R walk. Additionally, the walk from vertex 5 to vertex 6 cannot include vertex 3 because the walk 5-v_B^R does not include vertex 4.

Definition 17 *An* execution trace *of a program* $\mathcal{P}$ *is a sequence of strings* $T_{\mathcal{P}(I)} = t_1, t_2, \ldots, t_n$ *where each* $t_i \in T_{\mathcal{P}(I)}$ *represents the execution of a statement in* $\mathcal{P}$ *based on a run of* $\mathcal{P}$ *given input* I *plus infeasible traces that are undetectable based on program syntax alone.*

Theorem 1 *Given a program* $\mathcal{P}$ *represented by interprocedural control flow graph* $\mathcal{G}$, *every interprocedural walk in* $\mathcal{G}$ *represents some execution trace for* $\mathcal{G}$.

Proof: An interprocedural control flow graph may contain five types of arcs: direct control flow arcs ($F_\mathcal{G}$), interrupted-flow arcs ($I_\mathcal{G}$), call arcs ($C_\mathcal{G}$), return arcs ($R_\mathcal{G}$) and return arcs ($J_\mathcal{G}$). To prove that the definition of interprocedural walk is sufficiently restricted so as to assure that any graph traversal that conforms to this definition represents an execution trace, we first explore the possible combinations of types of arcs that may exist within an ICFG.

By the definition of interprocedural control flow graph we know that $I_\mathcal{G}$ is empty if and only if $C_\mathcal{G}$ and, consequently, $R_\mathcal{G}$ are empty. Thus, there are only three possible combinations of these five types of arcs that can comprise a traversal of an ICFG:

1. traversals that contain only direct control flow arcs,
2. traversals that contain all types of arcs except jump arcs, and
3. traversals that may contain an unmatched call arc incident to v_P^I and that are terminated with (h, v^F) where h is a halt in P.

We now explore the potential for traversal of these types to introduce walks in the digraph that do not represent execution traces of the program and show that none is possible.

- case 1: If $C_{\mathcal{G}}$ is empty then all arcs in $\mathcal{G}$ represent potential flow of control. Additionally, there is no predefined relationship between arcs incident to and incident from any vertex or region in the graph (i.e., after entering a vertex by way of a particular arc, any arc incident from the vertex provides a suitable exit route). Thus, for any execution trace $T = t_1, t_2, \ldots, t_n$, for each $t_i \in T$, $1 \leq i < n$, there exists a $v_i \in V_{\mathcal{G}}$ and $(v_i, v_{i+1}) \in A_{\mathcal{G}}$ and, therefore, by definition of digraph walk, $\mathcal{W}$ represents an execution trace of $\mathcal{P}$.

- case 2: If on the other hand $C_{\mathcal{G}}$ is not empty, and the called procedure does not contain a halt, then the call and return arcs are related by the condition of the definition of an ICFG and are related with respect to a program's execution in the same relationship as matching parenthesis — that is, for any call-return pair, if there is a return between a call-return pair in the subtrace, it must have be been preceded in that subtrace by a related call. If this were not the case, then the return would be to a procedure invocation other than the one from which it was called. The definition of interprocedural walk requires such ordering of call and return vertices; thus, any interprocedural walk that contains a procedure call will represent an execution trace of $\mathcal{P}$.

- case 3: Finally, if h is a halt statement in P_j and c is a call to P_j from P_i then there is an interprocedural walk $\mathcal{W}$ of the form $\mathcal{X}$-c-$v^I_{P_j}$-$\mathcal{Y}$-h-v^F. $\mathcal{X}$ and $\mathcal{Y}$ are walks of form of case 3 and therefore represent possible execution traces for $\mathcal{P}$. $\mathcal{X}$,c is a valid subtrace since a call may be executed after any other statement in the program. $(c, v^I_{P_j})$ is a call arc and thus $c, v^I_{P_j}$ represents a valid subtrace. $v^I_{P_j}$-$\mathcal{Y}$-h is valid since any type of statement may be executed after a procedure declaration, and a halt may occur after any other statement. Also, (h, v^F) is a jump arc that must terminate any digraph path to which it belongs and therefore represents a valid subtrace associated with program termination. Therefore $\mathcal{W}$ represents a potential execution trace for $\mathcal{P}$.

■

3.3.4 Forward Dominance in the Face of Procedure Calls

We now define and review the notion of forward dominance and examine its limitations for application to definitions of interprocedural control dependence. We introduce the forward dominance forest, which provides the foundation for our definition of control dependence among statements of multi-procedure programs.

The purpose of using forward dominance in dependence analysis is to capture the fact that the execution of a statement in a program necessarily implies the execution of other statements before program termination. Below we adapt the notion of forward dominance to apply to procedure control flow graphs that may contain call vertices. The definition of intraprocedural forward dominance requires prior definition of an intraprocedural walk. The definition of intraprocedural walk extends the Podgurski and Clarke definition of a digraph walk [41] by allowing traversal of ifa-arcs.

Definition 18 *An* intraprocedural walk *W in a procedure control flow graph G is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $n \geq 0$ and $(v_i, v_{i+1}) \in A_G$ or $(v_i, v_{i+1}) \in I_G$ for $1 \leq i < n$ and is denoted p-walk. The* length *of a p-walk $W = v_1, v_2, \ldots, v_n$, denoted $|W|$, is the number n of vertex occurrences in W. Note that a p-walk of length zero has no vertex occurrences; such a p-walk is called* empty.

Next, in preparation for our discussion of intraprocedural forward dominance, we define the forward dominance tree that was described in Section 2.2.4. This definition is adapted for forward dominance from Theorem 1 of Lengauer and Tarjan [32], which proves that the dominance relation among vertices of a single entry control flow graph forms a tree rooted at the entry to the CFG. If a CFG G is a single exit digraph the dominance relation on G^{-1} is the forward dominance relation on G. It is for this reason that some researchers refer to this relationship as "inverse dominance".

Definition 19 *Given a control flow graph G, a* forward dominance tree *$T = \{r, V, A\}$ where $V = \{v_1, \ldots, v_n, v^F\}$ represents the vertices in G, $r = v^F$ is the root of the tree, and $A = \{(\mathit{ifdom}(u), u)$ where $u \in V - v^F\}$.*

Intraprocedural Forward Dominance

The forward dominance relationship depends on the existence of a single point of exit from the graph. Since we are interested in representing procedures that may contain halts, our procedure control flow graph may have multiple exits, prohibiting its representation as a forward dominance tree. We define the forward dominance forest as a collection of related forward dominance trees, each of which represents the forward dominance relationships among a subset of the vertices of a control flow graph that contains more than one vertex with zero out-degree. The forward dominance relation of a procedure control flow graph is represented by a forward dominance forest. In the case that a procedure control flow graph has exactly one vertex with zero out-degree (i.e., v^R), then the procedure contains no halts and the forward dominance forest contains a single tree.

Definition 20 *Given a digraph G and $V_A = \{v_{a_1}, \ldots, v_{a_n}\}$ is a set containing exactly one element for each childless vertex in G, a* forward dominance forest *$F = \{t_1, \ldots, t_n\}$ is a set of forward dominance trees where, for each $t_i = \{r_i, V_i, A_i\} \in F$, $V_i = \{v_{i_1}, \ldots, v_{i_n}\}$ represents vertices in G, $r_i = v$ for some $v \in V_A$, and $A_i = \{(\mathit{ifdom}(u), u) \mid u \in V_i - v_A\}$.*

Forward dominance relationships among vertices in a procedure control flow graph depend upon the forward dominance relationships that exist in called procedures. Consider the situation where a procedure P_2 is called from procedure P_1 at statement x. If $v^R_{P_2}$ does not forward dominate $v^I_{P_2}$ then there is a path from $v^I_{P_2}$ that will not lead to a return to P_1, and therefore statements that follow x in P_1 do not forward dominate x. In recognition of this fact we define intraprocedural forward dominance, as well as procedure-level forward dominance, and prove a theorem that characterizes forward dominance in the face of procedure calls.

Definition 21 *Let G be a procedure control flow graph. A vertex $v \in V_G$* intraprocedurally forward dominates *a vertex $u \in V_G$ if and only if every u-v_G^R intraprocedural walk contains v, v* intraprocedurally properly forward dominates *u if and only if $u \neq v$ and v intraprocedurally forward dominates u. The* intraprocedural immediate forward dominator *of a vertex $u \in V_G$ is the vertex that is the first proper intraprocedural forward dominator of u to occur on every $u - v_G^R$ walk; we denote the immediate intraprocedural forward dominator of u by ip-$ifdom(u)$.*

Intraprocedural knowledge of the forward dominance relationships among vertices of the procedure does not cross interrupted-flow arcs since it is not possible to know in isolation whether control will return to an ifa-target. However, in the absence of procedure calls, intraprocedural immediate forward dominators are immediate forward dominators.

Theorem 2 *Given interprocedural control flow graph $\mathcal{G}$ and a procedure control flow graph $G \in \mathcal{G}$, if $(u, v) \in A_G$, ip-$ifdom(u) = v$, and no $u - v$ intraprocedural walk contains an ifa-target, then $ifdom(u) = v$.*

Proof: Assume that $v \neq ifdom(u)$, then some $u - v_{\mathcal{G}}^F$ path does not contain v. But every $u - v_G^R$ path in G contains v since ip-$ifdom(u) = v$ and, since no $u - v$ path contains a call vertex, every $u - v$ path is a prefix of some $u - v_{\mathcal{G}}^F$ path. ■

Next we define potential forward dominance in recognition of the fact that the forward dominance relationships among vertices in procedures can depend on the forward dominance relationships of called procedures and their descendents in the program's call graph. We follow this with a definition of an arc type that is used to replace potential forward dominance arcs when it can be determined that the target of pfdom arc does, in fact, forward dominate its source. We call these arcs indirect forward dominance arcs.

Definition 22 *Given a procedure control flow graph G and $(u, v) \in I_G$, then v* potentially forward dominates *u. This relationship is denoted $pfdom(u) = v$.*

Definition 23 *Given a procedure control flow graph G and $(u, v) \in I_G$, if $v \in \{x \mid x = fdom(u)\}$ then v* indirectly forward dominates *u. This relationship is denoted $indfdom(u) = v$.*

We define a procedure's forward dominance forest to be a collection of trees, one rooted at the vertex representing the procedure's return statement and other rooted at each vertex representing a halt statement. The arcs of a procedure forward dominance forest can be of two types.

Definition 24 *Given a procedure control flow graph G, a* procedure forward dominance forest *(PFDF) $F = \{R, T, V, A, D\}$ where $V = \{v_1, \ldots, vn\}$ is the set of vertices in G that have successors, $R = \{r_1, \ldots, r_n\}$ is the set of childless vertices in G, and for every $v \in V$ if v is an ifa-source then there exists a $d_i = (pfdom(v), v) \in D$ else there exists an $a_i = (ifdom(v), v) \in A$.*

During program composition time, we determine whether the target of a pfdom arc does in fact forward dominate its source. If it does then we convert the pfdom arc to an indfdom arc. If it does not we delete the pfdom arc from D. We define a resolved procedure foward dominance forest to be the set of vertices and arcs that comprise the PFDF with the pfdom arcs removed and indfdom arcs added where appropriate.

Definition 25 *Given a PFDF F, a* resolved procedure forward dominance forest *(RPFDF)* $R = F - D + I$ *where* $I = \{a_1, \ldots, a_n\}$ *and* $a_i = (\mathit{indfdom}(v), v) \in A$.

Definition 26 *Given a program* $\mathcal{P} = \{P_1, \ldots, P_n\}$*, its associated* compound forward dominance forest *(CFDF)* $\mathcal{F}$ *is a set containing one PFDF for each procedure in* $\mathcal{P}$.

Definition 27 *Given a program* $\mathcal{P} = \{P_1, \ldots, P_n\}$*, its associated* forward dominance forest *(FDF)* $\mathcal{R}$ *is a set containing one RPFDF for each procedure in* $\mathcal{P}$.

Given these definitions we can explore the relationship between intraprocedural forward dominance and forward dominance relationships among vertices of sets of procedure control flow graphs. The following theorem states that potential forward dominance relationships between vertices in a PFDF can be converted to forward dominance relationships if a path between the vertices in the procedure forward dominance forest does not include vertices that are connected by pfdom arcs.

Theorem 3 The Intraprocedural Forward Dominance Theorem — *Given procedure control flow graph G, vertex v forward dominates vertex u if and only if there exists a sequence of vertices in* $G = v_1, v_2, \ldots, v_n$ *where* $u = v_1$ *and* $v = v_n$*, such that* $v_{i+1} = \mathit{ifdom}(v_i)$ *or* $v_{i+1} = \mathit{indfdom}(v_i)$ *for* $i = 1, \ldots, n-1$.

Proof: This follows directly from transitivity of the forward dominance relationship. ■

As noted earlier, we recognize that the forward dominance of a vertex incident from an interrupted-flow arc, and that of any of its descendents in the PCFG, may be destroyed by the potential for non-return from the called procedure. This can happen in two ways: First, there may be potential for non-termination in the forms of loops or recursive procedure calls embedded within the called procedure or its descendents. Second, there may be program exits from within the called procedure or its descendents. To account for these possibilities, we view the forward dominance of such a vertex in terms of the forward dominance relationship of the return vertex of the called procedure over that procedure's initial vertex. Thus, the forward dominance relationships within a procedure are dependent upon those of the procedures that it calls.

Referring to the CFDF for our running example, which is shown in Figure 3.7, V_C^R does not forward dominate v_C^I due to the embedded halt at statement 22. Therefore, vertex 12 does not forward dominate any vertices in procedure B. If vertex 22 did not represent a halt statement, then, v_C^R would forward dominate v_C^I and intuitively we recognize that vertex 12 forward dominates vertex 11 as well as all vertices that vertex 11 forward dominates. We state this formally in the

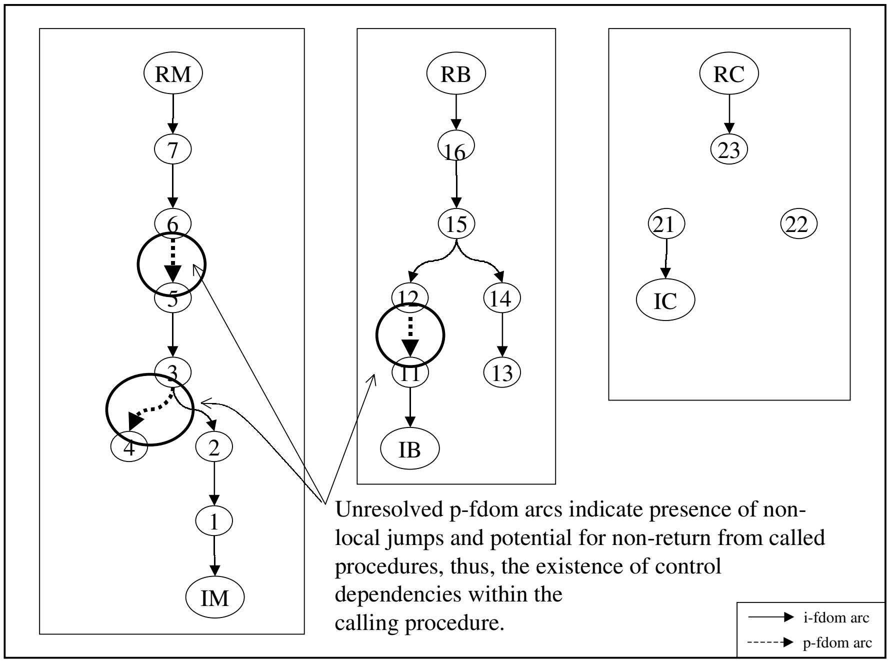

Figure 3.7: CFDF for Program Sum with Embedded Halt at Statement 18.

following theorem and prove the correctness of our intuition. We use this theorem to resolve pfdom arcs into indirect fdom arcs.

Theorem 4 *Given a procedure control flow graph G, v_G^R forward dominates v_G^I if and only if the forward dominance forest for G is comprised of a single tree rooted at v_G^R.*

Proof: This is two-part proof:

1. We first show that if v_G^R forward dominates v_G^I if the forward dominance forest for G is comprised of a single tree rooted at v_G^R. Assume that v_G^R forward dominates v_G^I and the forward dominance forest for G contains more than one tree. If there is more than one tree in the forward dominance forest then each root r of a tree has no forward dominator in G and neither does v_G^R since it has no descendants in the control flow graph and therefore must be a root in the PFDF; thus, v_G^R does not forward dominate r and there is a path beginning at r that does not encounter v_G^R. Since r is reachable from v_G^I, by path concatenation we know that there exists a path beginning at v_G^I that does not include v_G^R and v_G^R does not forward dominate v_G^I, a contradiction.

2. We now show that if the forward dominance forest for G is comprised of a single tree rooted at v_G^R then v_G^R forward dominates v_G^I. Trivially, if the forward dominance forest is comprised of a single tree rooted at v_G^R then v_G^R forward dominates all vertices in G including v_G^I.

■

Theorem 5 The Interprocedural Forward Dominance Theorem — *Given program $\mathcal{P}$ represented by composed control flow graph $\mathcal{G}$ and two procedures P_i, $P_j \in \mathcal{P}$ represented by their respective procedure control flow graphs G_i, $G_j \in \mathcal{G}$, if vertex u with related interrupted-flow arc (u,v) $\in I_{G_i}$ represents a call to P_j from within P_i, then v forward dominates u if and only if $v_{G_j}^R$ forward dominates $v_{G_j}^I$.*

Proof: This is a two-part proof:

1. We first show that $v_{G_j}^R$ forward dominates $v_{G_j}^I$ implies v forward dominates u. By the definition of forward dominance, every $v_{G_j}^I - V_{\mathcal{G}}^F$ walk $\mathcal{W} \in \mathcal{G}$ contains $v_{G_j}^R$. By the definition of compound control flow graph we have unique call and return arcs $(u, v_{G_j}^I) \in C_{\mathcal{G}}$, $(v_{G_i,v}^R)$ $\in R_{\mathcal{G}}$; thus, by walk concatenation we have an interprocedural walk $\mathcal{Y} = u\mathcal{W}$ and by the definition of interprocedural walk we know that every walk beginning with vertex u and containing $v_{G_j}^R$ also contains v. Thus, v appears in every $u - V_{\mathcal{G}}^F$ walk in $\mathcal{G}$ and v forward dominates u in $\mathcal{G}$.

2. We now show that v forward dominates u implies $v_{G_j}^R$ forward dominates $v_{G_j}^I$ By definition of forward dominance every walk $u\mathcal{W}_i$ in $\mathcal{G}$ contains v and by definition of interprocedural walk, if $uv_{G_j}^I$ is a prefix for a $u - V_{\mathcal{G}}^F$ walk $\mathcal{X}$ and that walk contains v, then $\mathcal{X}$ contains $uv_{G_j}^F v$.

■

Theorem 6 *Given program $\mathcal{P}$ represented by composed control flow graph $\mathcal{G}$ and two procedures P_i, $P_j \in \mathcal{P}$ represented by their respective procedure control flow graphs G_i, $G_j \in \mathcal{G}$, if vertex u with related interrupted-flow arc (u,v) $\in I_{G_i}$ represents a call to P_j from within P_i then v forward dominates u if and only if the forward dominance forest for G_j is comprised of a single tree.*

Proof: Follows directly from Theorems 4 and 5. ■

3.3.5 Composed Interprocedural Control Dependence

Control dependence is a relation on the vertices of a graph intended to capture relationships among statements in execution traces of a program. An inlined control flow graph is a digraph in which each procedure call is replaced by the body of the procedure. In this representation each statement in the program may be associated with several vertices of the flow graph, thereby capturing the context for each call to the procedure. Each arc in an inlined CFG represents the potential for flow of control, and the graph has a single point of entry, and a single point of exit; thus, an inlined CFG has the same characteristics as a CFG and all algorithms that apply to the CFG can be directly applied to an inlined CFG without fear of producing inaccurate results.

A composed control flow graph is a multi-graph containing several types of arcs. Each statement is associated with exactly one vertex, which can be associated with many call-site specific control dependencies. This situation is the cause of the calling context effects described in Chapter 2. Additionally, an non-inlined CFG contains unrealizable paths as discussed in Section 3.3.3.

We use a modular approach for calculating control dependencies in programs composed of multiple procedures. The dependence relationships are calculated for each procedure in isolation, then compositional dependencies are based on inheriting dependencies from call and return vertices of procedures. For example, if P_1 calls P_2 from statement x, then entry to P_2 inherits its control dependence from x. If an individual procedure contains a procedure call, then a temporary assignment of dependencies is made based on the assumption of return from the procedure. If during composition it is determined that there is potential for non-return, then these temporary control dependence arcs are replaced by interprocedural control dependence arcs.

In Section 3.1 we described and defined three types of control dependencies that are of concern when crossing procedure boundaries: potential, invocation, and resumption dependence. The initial vertex and each ifa-target are boundaries for control dependencies. Each vertex that is not otherwise control dependent that lies on a path between a vertex u of this type and the subsequent vertex v of this type is either invocation or resumption control dependent on the same vertex as u. The control dependencies of the initial vertex are inherited from call vertices in calling procedures' control flow graphs; the control dependence of an ifa-target is the same as that of the return vertex of the called procedure. These cannot be determined before program composition, since the structure of the program and called procedures are unavailable. However, it is possible to determine which vertices in the procedure control flow graph have the same control dependence as the initial vertex and the ifa-targets; the procedure control dependence graph contains "inherited" control dependence arcs to record this knowledge. Additionally, it is not possible to tell whether the ifa-target will be control dependent on a vertex in the called procedure or inherit the dependence of the ifa-source in isolation, but a potential control dependence can be assigned based on the assumption that control will return; potential control dependencies are resolved during program composition based on the structure of the called procedure.

Definition 28 *Given a procedure control flow graph G and its associated procedure forward dominance forest F, a vertex v in G is* intraprocedurally control dependent *on another vertex u in G if and only if there is a $u - v$ intraprocedural walk $\mathcal{W}$ that does not include the immediate forward dominator of u. v is* directly intraprocedurally control dependent *on u if v is the first vertex in $\mathcal{W}$ that is intraprocedurally control dependent on u.*

The control dependence of the ifa-target in a procedure control flow graph is determined by resolution or non-resolution of pfdom arcs. Recall that pfdom arcs are inserted in the forward dominance forest to record the potential for an ifa-target to forward dominate its source. During system composition, using a post-order traversal over the call graph, each called procedure is checked in order to determine whether the ifa-target does, in fact, forward dominate its source. If it does not, then the pfdom arc is removed; if it does then it is replaced with an indirect fdom arc. Since forward dominance is a transitive relationship, forward dominance among vertices of a CFG can be determined from the transitive closure over ifdom arcs; fdom arcs are inserted only when a forward dominance relationship is identified between the source and target of an ifa-arc.

If, after all call sites in the procedure have been evaluated, any pfdom arcs have been removed, then there are two or more trees in the forward dominance forest associated with the called procedure. In this case the return from the procedure is control dependent on a decision in a called procedure or one of its descendents in the call graph. The ifa-target inherits its control dependencies from the called procedure's return vertex.

In Section 3.1 we defined interprocedural control dependence in terms of program statements. Based on that definition we provide a graph-based definition of interprocedural control dependence. Informally, a vertex v in a procedure control flow graph, G_i is said to be *interprocedurally control dependent* on a vertex u in a procedure control flow graph G_j if u has potential to affect the number of times v is encountered on an interprocedural walk that begins with the first occurrence of an ifa-source and ending with the last occurrence of the related ifa-target.

A composed control dependence graph is a graph that contains one procedure control dependence graph for each procedure in the program. Inherited and potential control dependence arcs provide a means to locate the source of interprocedural control dependencies. During system composition, inherited control dependence arcs are inserted from each call site to v^I of the called procedure. Next, dependencies related to call sites within the procedure are determined. If it can be determined that v^R of the called procedure exhibits a control dependence other than an invocation control dependence then potential control dependence arcs incident to the ifa-target are removed and inherited control dependence arcs are inserted between the called procedure's return vertex and the ifa-target. Otherwise the potential control dependence arc is converted to an inherited dependence arc.

Inherited dependence arcs are traversed during system control dependence analysis in order to identify direct control dependencies that are inherited transitively from other vertices. All vertices encountered on a path in a composed control dependence graph that originate at the source s of a direct control dependence arc and that do not include any other direct control dependence arcs are directly control dependent on s. After all inherited control dependencies have been removed we call the graph a control dependence graph.

Based on this intuitive notion of composed control dependence we now define a procedure control dependence graph, composed control dependence graph, and control dependence graph.

Definition 29 *A* procedure control dependence graph *(PCDG) $G = (V, A)$ for a procedure P is a graph that has one vertex for each vertex in the procedure control flow graph that represents P. $A = CD \cup PC \cup IC$ is a set of arcs, where an arc in CD represents the direct control dependence of its target on its source, each arc in PC represents the potential for its target to inherit the dependence of its source, and an arc in IC represents the fact that its target inherits the control dependence of its source.*

Definition 30 *A* composed control dependence graph *(CCDG)* $\mathcal{G} = (V, A, v^S, v^F)$ *for a program* $\mathcal{P} = \{P_1, \ldots, P_n\}$ *is a graph that has one vertex for each vertex in the procedure control flow graph associated with each* $P_i \in \mathcal{P}$. $A = CD \cup IC$ *is a set of arcs where an arc in* CD *represents the direct control dependence of its target on its source and arcs in* IC *represent inherited control dependencies.* v^S *and* v^F *represent the start and the finish of* $\mathcal{P}$ *respectively.*

Definition 31 *A composed control dependence graph in which paths including inherited dependencies have been resolved into direct control dependencies is called a* control dependence graph *(CDG).*

The composed control dependence graph for our running example is shown in Figure 3.8. The solid arcs in the figure represent direct control dependence and the dashed arcs represent inherited control dependence. The slashed arcs: (4,3), (5,6), and (11,12), represent the replacement of potential control dependence arcs with the inherited arcs: (RB,3), (RB,6), and (RC,12). Figure 3.9 contains the control dependence graph for Sum in which all inherited arcs have been removed and replaced by the appropriate direct control dependence arcs.

3.3.6 The Call Graph

Throughout this chapter we have discussed the composition of program-level structures from their associated procedure-level structures at program composition time. In this section we describe the "call graph" that can be used to determine the relationships among the procedures of a program.

The *call graph* associated with a program is a digraph that represents the calling relationships among procedures in the program. It is composed of vertices that represent the procedures in the program and annotated arcs. Each arc represents the existence of a procedure call at a specific location within the procedure represented by the source of the arc to the procedure represented by the target of the arc. The arc is annotated with unique identifiers associated with the each call site. A call graph will be acyclic (i.e., not contain back arcs) if the program does not contain any direct or indirect recursion among its procedures. Sets of vertices that form cycles in a digraph comprise what are known as "strong components" in a digraph [13]. For dependence analysis, the important relationship among such vertices is that the existence of an embedded halt in any one of the represented procedures results in the removal of at least one pfdom arc from each associated procedure forward dominance forest. This condition creates resumption dependencies of all procedures that call any procedure the is represented by a vertex that belong to the strong component.

The graph in Figure 3.11 (a) is based on the graph used by Gabow [19] in the presentation of his algorithm for computing strong components. We have changed the names of the vertices to represent procedure names and refer to the graph as the call graph for program Recurse. Pseudo code for program Recurse is shown in Figure 3.10. The graph in Figure 3.11 (b) is called the contracted graph of the call graph for program Recurse. The vertices in the contracted graph represent the strong components of Recurse's call graph. These strong components represent sets of procedures that are callable from one and another, thus identifying procedures that are related by either direct or indirect recursion in the program.

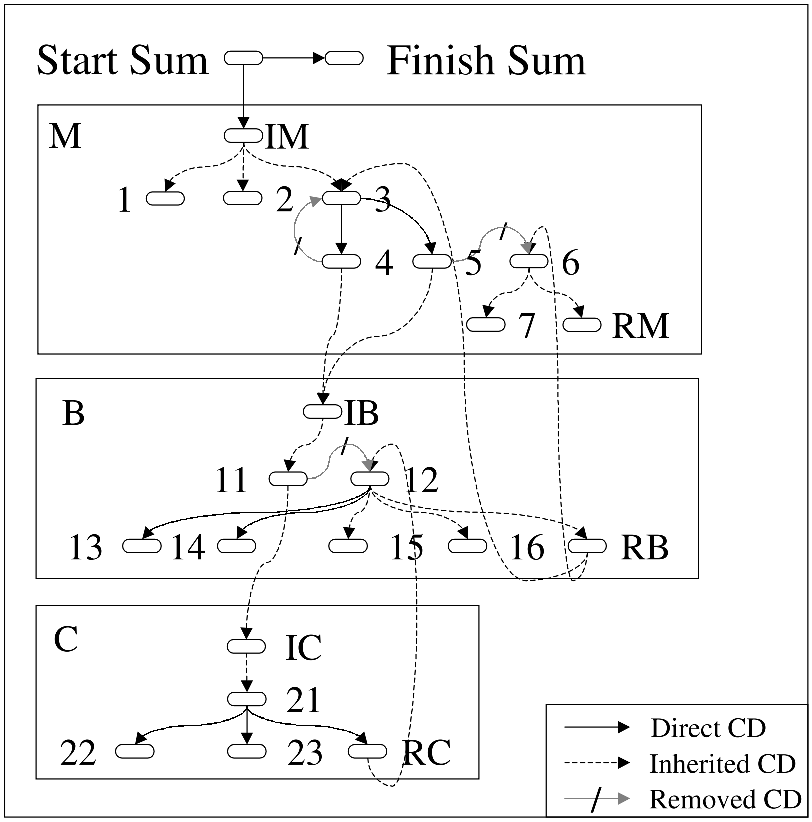

Figure 3.8: Composed Control Dependence Graph for Program Sum.

3.3.7 Summary

In this chapter we have defined a model of interprocedural control dependencies that is appropriate for identifying dependencies in programs that are composed of procedures that may have been created in isolation and for which the source code may not be available. Individual procedures are analyzed in isolation and the results of these analyses can be combined at program composition time using compositional control dependence analysis to identify control dependencies among the procedures of the program.

Each procedure is assumed to be represented by a language-independent procedure control flow graph. From this graph we identify forward dominance and potential forward dominance relationships among the vertices of the control flow graph. Potential forward dominance relationships exist between call vertices and the vertex that is the first one within the calling PCFG to be encountered if control returns from the call. These two relationships are recorded in a procedure forward dom-

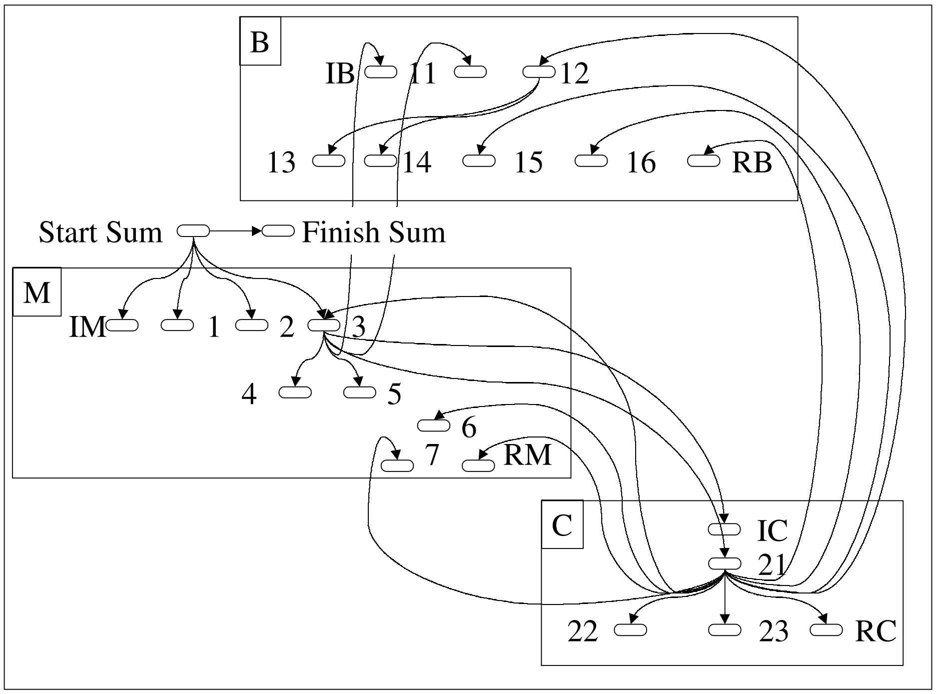

Figure 3.9: Control Dependence Graph for Program Sum.

```
                proc Main
            1:  call A
            2:  call B
            3:  return

    proc A                          proc B
1:  call B                      1:  return
2:  call D
3:  return          proc D
                1:  call B
                2:  call C
                3:  return

        proc C                      proc E
    1:  if ()                   1:  call B
    2:      call A              2:  call D
        else                    3:  return
    3:      halt
    4:  call E
    3:  return
```

Figure 3.10: Program Recurse.

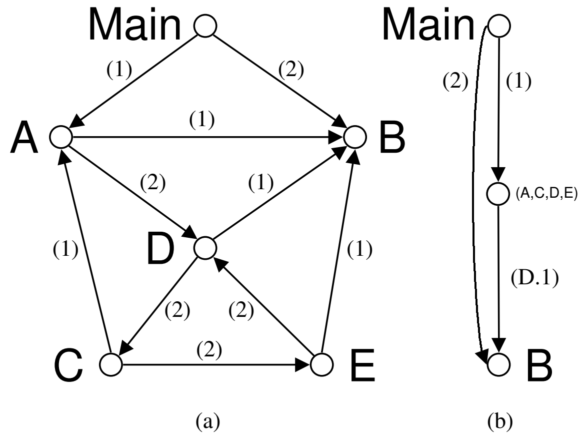

Figure 3.11: Call Graph for Program Recurse.

inance forest. Using the PCFG and the PFDF in combination, the definition of intraprocedural control dependence can be used to identify inherited and potential control dependencies among vertices of the PCFG as well as strong and weak control dependencies among them, which are then recorded in the form of a procedure control dependence graph.

At program composition time the following information associated with each procedure is sufficient for computing control dependencies for the entire program.

1. the PCDG,
2. a list of call sites and the procedures called from them,
3. a list of interrupted-flow arcs, and
4. knowledge of whether the procedure forward dominance forest is composed of exactly one tree or more than one tree.

Given this information a call graph for the program is constructed, then contracted in order to identify recursion among the program's procedures. The call graph is used in conjunction with the items enumerated above to identify program-wide control dependencies.

Taking this approach allows us to reuse procedure control dependence information in multiple applications and allows us to analyze composed programs without requiring access to its procedures' source code.

Chapter 4

The Algorithms

In this chapter we describe the algorithms for constructing the various structures associated with our model. The construction algorithms are followed by a discussion of their complexity, including a discussion of the potential for overall reduction in cost due to the compositional nature of the model.

4.1 Construction

We construct the program's control dependence graph hierarchically. Recall the model depicted in Figure 3.4 on Page 36. We begin construction of the compound control dependence graph by collecting procedure control flow graphs for each procedure in the program. From this we create procedure forward dominance forests and use these in combination with the PCFGs to create procedure control dependence graphs. We then use the program's call graph in combination with PFDFs to identify potential control dependencies among procedures. A procedure can have associated attributes including pointers to its previously computed PCFG, PFDF, and PCDG as well as a character value called PFDF_type that is set to "F" if the PFDF is a forest, "T" if it is a tree, and null if the PFDF has not yet been computed. The collection of procedure control dependence graphs is augmented with interprocedural control dependence arcs and is called the compound control dependence graph.

Construction of procedure control flow graphs is a language-dependent operation and the existence of procedure control flow graphs is assumed. For the algorithm to apply, the PCFGs must satisfy our definition of procedure control flow graph; the correctness of the resultant control dependence graph depends on the PCFGs' faithful representation of all potential control flows among statements in the procedure.

The steps for construction of the CCDG are listed below. Steps 1 and 2 may be undertaken at the time of procedure creation and the resultant structures provided along with the procedure at program composition time. Steps 3, 4, and 5 are undertaken at program composition time when all procedures are known.

1. For each procedure P in $\mathcal{P}$ compute PFDF_P
2. For each procedure P in $\mathcal{P}$ compute PCDG_P
3. Compute $\mathcal{CG}_{\mathcal{P}}$
4. Compute the CCDG $\mathcal{D}_{\mathcal{P}}$
   (a) For each PCDG determine and insert control dependence arcs for initial vertices
   (b) Resolve pfdom arcs where possible
   (c) For each PCDG resolve potential control dependence arcs
5. Resolve $\mathcal{D}_{\mathcal{P}}$ to produce $\mathcal{C}_{\mathcal{P}}$

In the following sections we provide algorithms for performing each of the steps enumerated above. The structure we have chosen for presentation of the algorithms is as follows:

- Commentary embedded within an algorithm begins with /* and ends with */.
- the keywords `Requires`, `Provides`, `Declare`, `begin`, `end`, `create`, `if` ... `then`, `if` ... `then` ... `else`, and `foreach` ... `do` have their usual meanings.
- Each algorithm is preceded by a list of required inputs, promised outputs, and declarations for variables, which may optionally be assigned initial values.
- $*v$ means "reference to v".
- $\leftarrow$ means "is assigned the value".
- $+$ when applied to a set stands for adding an element to the set.
- $-$ when applied to a set stands for removal of an element from the set .
- Indentation indicates nesting.
- Each set of nested statements is terminated by an appropriate "end" statement.
- We continue the convention used throughout this dissertation of using calligraphy font to represent program-level structures, capital letters to represent procedure-level structures, and lower-case letters to represent statement-level structures.

As we present the algorithms, we illustrate their use to create the CDG for an example program. Figures 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6 show the results at various stages of its creation. The example is contrived to exercise the algorithms and is not intended to reflect a realistic program. Figure 4.1 shows structures that result from applying Algorithms 1 and 2 to the PCFGs for procedures A, B, and C that make up the example program.

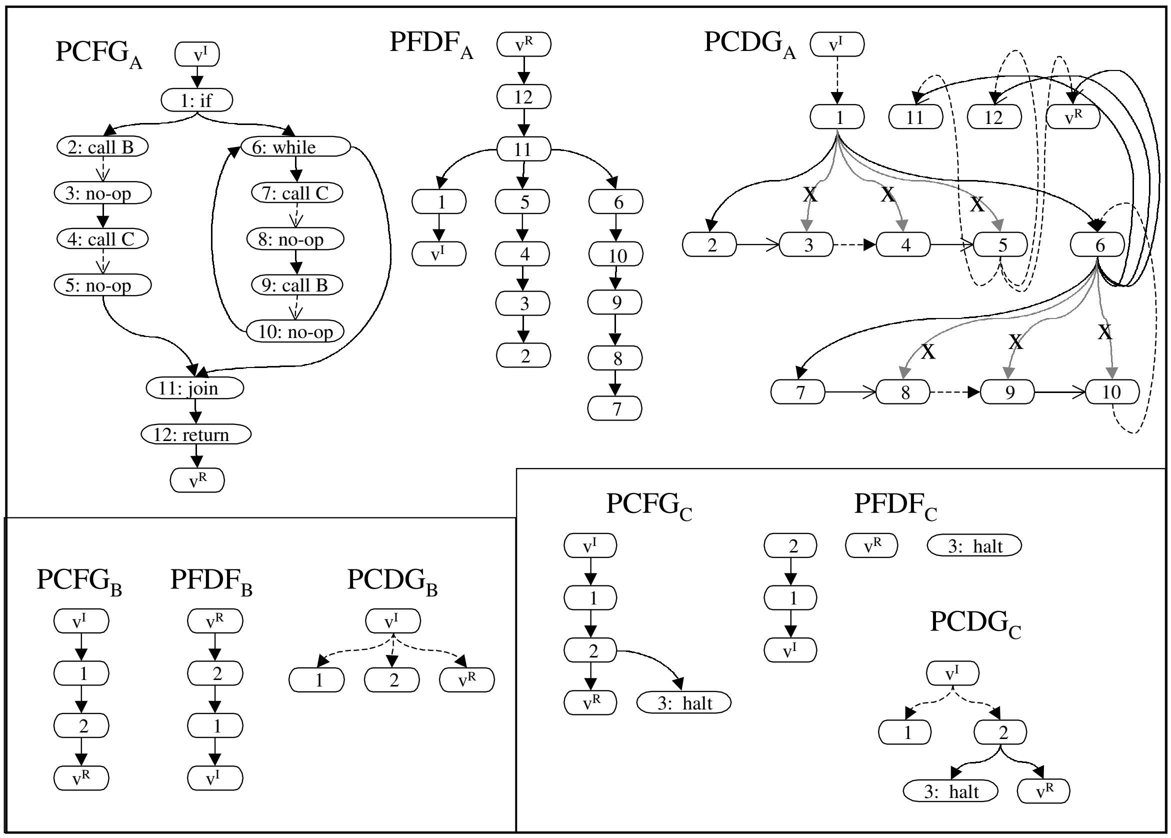

Figure 4.1: Example Structures of the Model.

4.1.1 Procedure Forward Dominance Forest

We showed in Chapter 3 that the forward dominance relationships among vertices of a procedure control flow graph can be computed based on the potential control flow from a procedure call to its related ifa-target. At program composition time, the forward dominance relationship between the ifa-target and source is resolved.

A program's forward dominance forest is a collection of procedure forward dominance forests (PFDF). There are two types of arcs in a PFDF: (1) immediate forward dominance, and (2) potential forward dominance or, after pfdom resolution, indirect forward dominance arcs. ifdom arcs represent the immediate forward dominance of their source over the target, which pfdom arcs represent the potential for forward dominance of the source over the target in the case where the vertices are ifa-source and target. Conceptually, during the pfdom resolution stage of CCDG construction, indirect forward dominance arcs may replace pfdom arcs. indfdom arcs represent non-immediate forward dominance of the ifa-target over its source. The pfdom arcs are conceptually deleted in the case where it is determined that control may be diverted around the procedure return. These two actions are conceptual in that the PFDF of the procedure is never actually altered. But the effect of such changes is required for the next stage of CCDG construction, so these changes must be recorded. In the second case the structure becomes a forest, where the trees represent the forward dominance relationships among vertices of subgraphs of the procedure control flow graph. When a PFDF contains one or more pfdom arcs it is said to be unresolved.

Construction of unresolved PFDF may be accomplished using a known algorithm such as the Lengauer/Tarjan [32] algorithm. The Lengauer/Tarjan algorithm describes an efficient procedure for computing a dominance tree for a single entry control flow graph. Computation of the forward dominance tree of a single exit control flow graph is equivalent to computation of the dominance tree for the inverse control flow graph. We begin construction of the PFDF by creating a temporary digraph T, by adding a temporary vertex v^T to G, and creating a set of arcs A^T that are incident from the return vertex of G and any other vertices in G that have zero out-degree (i.e., from any vertices representing halt statements). This is done to create a graph structure to which the Lengauer/Tarjan algorithm can be applied. After the forward dominance tree has been computed, the temporary vertex and arcs incident from it in the forward dominance tree are removed. If the resultant graph contains more than one tree, the return vertex of the procedure does not forward dominate its initial vertex and is therefore control dependent on some vertex within the procedure; we assign the value "F" to the procedure's PFDF_type, otherwise we assign it the value "T" for tree.

Algorithm for Construction of Procedure Forward Dominance Forest

Algorithm 1

`Requires`:	*G for procedure P*		/* *G is the PCFG for P* */
`Provides`:	*PFDF_type* $\leftarrow \emptyset$		/* *initialize PFDF_type for P* */
`Declare`:	*vertex:*	v^T	/* *a super exit vertex that succeeds vertices representing the return and any halt statements of P* */
	arc:	a	/* *temporary arcs connecting childless vertices of G with* v^T */
	set of arcs:	$A^T \leftarrow \emptyset$	/* *set of temporary arcs* */
	set of refs:	*Forest_list*	/* *list of references to roots of trees* */
	digraph:	G'	/* *extension of G that includes the super exit vertex* */
	digraph:	F	/* *forward dominance tree for* G' */

/* *construct PFDF* */
`begin` *Construct_PFDF($\mathcal{P}$)*

```
/* create temporary digraph G' = {G, v^T, A^T} */
foreach v ∈ G where out-degree(v) = 0 do
    create a ← (v, v^T)
    A^T ← A^T + a
end foreach
/* compute procedure forward dominance forest F for G'.
   Apply Lengauer/Tarjan to G'^{-1}, treating
   interrupted-flow arcs are as direct control flow arcs. */
F ← compute_FDT(G')
   /* in the example F_B = {(v^T, v^R), (v^R, 2), (2, 1), (1, v^I)} */
```

/ If the root of the resulting tree has out-degree* $= 1$ *then its only successor is the return vertex of G, and we know the return vertex is not control dependent on any vertex within the procedure; we mark PFDF_type = "T". If at program composition time, one or more pfdom arcs are removed this will be changed to "F" to reflect that, in the setting of that program* v_G^R*, does not forward dominate* v_G^I **/*

```
if out-degree(r_F) = 1 then
    PFDF_type ← "T" /* for tree */
else
    PFDF_type ← "F" /* for forest */
end if
/* create Forest_list – pointers to roots of trees in the forest */
foreach a = (u, v) ∈ A^T do
    Forest_list ← Forest_list + *v
end foreach
foreach (u,v) ∈ I_G do
    convert (v,u) ∈ F to pfdom arc
        /* each arc in F that is associated with an interrupted-flow arc is converted
        to a pfdom arc */
end foreach
/* G' is no longer needed. */
end Construct_PFDF(P)
```

Figure 4.1 shows the PFDFs for procedures A, B, and C that make up the example program.

4.1.2 Procedure Control Dependence Graph

Definition 10 states that direct control dependence can be of four different types. Therefore, the control dependence arc set for the PCDG may contain up to four subsets of arcs, one for each type of control dependence that is identified for the procedure. We review these types of control dependencies below:

1. Strong control dependence — assumes finite execution of all loops.

2. Weak control dependence — recognizes dependence of a statement's execution on the exit condition of a loop.

3. Potential control dependence — ifa-target assumes the same control dependence as its related ifa-source. These may be redirected at program composition time when it can be determined if there is potential for non-return from the related procedure call.

4. Inherited control dependence — if there is a path from $u = (v_G^I$ or an ifa-target) to some vertex v in the PCFG that is not otherwise control dependent on a vertex other than itself and that does not contain an ifa-target, then v inherits the control dependence of u.

If vertex v is directly strong, weak, potential, or inherited control dependent on a vertex u, we denote these relationships as $v \stackrel{dscd}{\longrightarrow} u$, $v \stackrel{dwcd}{\longrightarrow} u$, $v \stackrel{dpcd}{\longrightarrow} u$, or $v \stackrel{dicd}{\longrightarrow} u$, respectively.

Weak control dependencies are a superset of strong control dependencies and this level of precision may not be required for the particular application of the dependence analysis. Therefore, the first step in creating a PCDG is to decide whether strong or weak control dependence is to be computed. After this decision is made, the appropriate control dependence is computed; it is then possible to identify inherited control dependencies.

Algorithm for Construction of Procedure Control Dependence Graph

Algorithm 2

`Requires`: *digraph:* G /* G *is the PCFG for* P */
digraph: F /* F *is the PFDF for* P */
string: CD_type /* *user input either "strong" or "weak"* */
`Provides`: *digraph:* D /* D *is the PCDG for* P */
`Declare`: *set ref:* $V = *V_G$ /* *a reference to the vertices of* G */
set of arcs: $A^{dscd} \leftarrow \emptyset$ /* *direct strong control dependence arcs* */
set of arcs: $A^{dwcd} \leftarrow \emptyset$ /* *direct weak control dependence arcs* */
set of arcs: $A^{dpcd} \leftarrow \emptyset$ /* *direct potential control dependence arcs* */
set of arcs: $A^{dicd} \leftarrow \emptyset$ /* *direct inherited control dependence arcs* */
set of refs to sets of arcs: $A = \{*A^{dscd}, *A^{dwcd}, *A^{dpcd}, *A^{dicd}\}$
/* A *is the set of pointers to the individual sets of arcs* */
set of vertex refs: $T = \{t_0, \ldots, t_k\}$
/* T *is a set of ifa-targets associated with the procedure,* $k = |I_G|$. */
/* *create procedure control dependence graph* D *for* P */
`begin` *Construct_PCDG*
1: `if` $CD\_type = strong$ `then`

/* Compute using Ferrante et al. direct control dependence algorithm [17]. In the example $A_B^{dscd} = \{(v^I, 1), (v^I, 2), (v^I, 3)\}$ */

compute(A^{dscd})

end if

if *CD_type* = *weak* **then**

/* Compute using Podgurski's direct weak control dependence algorithm [42]. In the example $A_B^{dwcd} = \{(v^I, 1), (v^I, 2), (v^I, 3)\}$ (note that this is the same set as A_B^{dscd} because there are no loops in B). */

compute(A^{dwcd})

end if

/* Now insert $\stackrel{dpcd}{\longrightarrow}$ arcs that connect ifa-source and ifa-targets in the PCDG. If it is determined at program composition time that the return from a procedure call is not control dependent on some vertex within the called procedure or one of its descendants in the call graph, then control dependence of the related ifa-target will be inherited from its source. Otherwise the ifa-target will be resumption dependent on the return vertex of the called procedure. */

/* insert dpcd arcs */

foreach $(u, v) \in I_{G_c}$ **do**

$A^{dpcd} \leftarrow A^{dpcd} +$ (u,v)

/* Direct strong or weak control dependence arcs incident to the ifa-target are deleted since the direct control dependencies of v are a superset of those of u, since u will either be the target of a control flow and may also be inherited from program, composition time or redirected to the return vertex of the called procedure. This step is depicted by the crossed-out arcs (1,3), (1,5), (6,8), and (6,10) in Figure 4.1. */

$A^{d\{s,w\}cd} \leftarrow A^{d\{s,w\}cd} - \{(pred(u), v) \mid \forall pred(u) \in PRED(u)\}$

end foreach

/* Given vertices $u, v \in G$, if $u = v^I$ or u is an ifa-target and the following two conditions hold:

- v forward dominates u
- there is a $u - v$ path in the control flow graph that does not contain any ifa-target

then unless *CD_type* = *weak* and $u \stackrel{dwcd}{\longrightarrow} w$ for some w in $u - v$ path in G, $v \stackrel{dicd}{\longrightarrow} u$. */

/* insert inherited dependence arcs */

$t_0 \leftarrow v^I$

/* T = set of ifa-targets plus the initial vertex; these are potential sources of inherited dependencies */

foreach $a = (s, t) \in I_G$ **do**

$T \leftarrow T + t$ for $\{t \mid (s, t) \in I_G\}$

end foreach

```
foreach t ∈ T do
   foreach ancestor v_j of t in PFDF up to any v ∈ T do
      foreach a = (v, v_j) ∈ A_D
         if (v, s) ∈ A_D then
            A_D ← A_D − a
         /* this step is depicted by crossed-out arcs (1,4) and (6,9) in Figure 4.1 */
         end if
      end foreach
      A^dicd ← A^dicd + (t,v_j)
      /* v_j inherits its dependence from the ifa_target t */
```

/ A loop header becomes an if-statement when it is determined that there is potential for non-return from a call within the loop. If t is directly control dependent on v_j and v_j also forward dominates t we know that v_j is a loop header. Removal of the potential control dependence tells us that there is a path from the loop header to the program exit from inside the loop, thus the loop header is no longer forward dominated by any successors. We must create direct control dependencies of all previously identified forward dominators of v_j. When potential control dependencies are resolved, if the ifa-target does not inherit from the ifa-source, then these potential dependencies will be kept, otherwise we know that control will return and these arcs are removed. */*

```
      if (v_j,t) ∈ A_D then
         foreach v_k ancestors of v_j in the PFDF up to any v ∈ T
            A^dicd ← A^dicd + (v_j,v_k)
         end foreach
      end if
   end foreach
end foreach
end Construct_PCDG
```

Figure 4.1 shows the PCDGs for procedures A, B, and C that make up the example program.

4.1.3 Composed Control Dependence Graph

Construction of the composed control depenendence graph requires three major steps: addition of invocation control dependence arcs, the resolution of pfdom arcs, and the identification of resumption control dependence and insertion of the related arcs. Each of these steps is presented below.

Procedure entry inherits control dependencies of all call sites. If a pfdom arc is removed during pfdom resolution, then the related ifa-target inherits the control dependence of the procedure called by the ifa-source. These two steps of the algorithm are simple to perform.

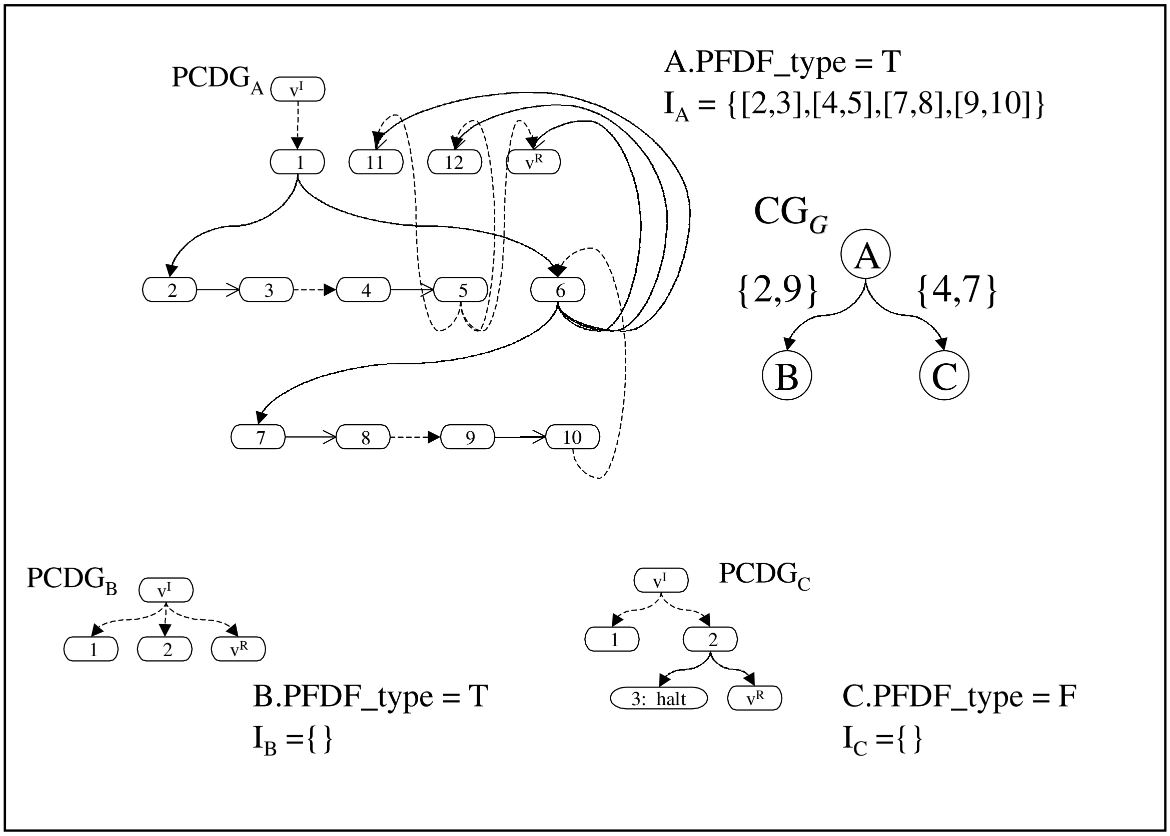

Figure 4.2: Items Required for Computing CCDG.

Figure 4.2 shows the structures for computing compositional control dependence for procedures A, B, and C that make up the example program. These are referred to in the following algorithm.

Algorithm for Construction of Composed Control Dependence Graph

Algorithm 3

`Requires`: *digraph:* $\mathcal{CG}_{\mathcal{P}}$ /* *call graph for* $\mathcal{P}$ */
digraph set: $\mathcal{P} = \{P_0, \ldots, P_{n-1}\}$ /* P_0 *is main procedure of* $\mathcal{P}$ */
for each $P_i \in \mathcal{P}$:
  digraph: D_i /* *PCDG* */
  arc sets: I_i /* *interrupted-flow arc set* */

character: $PFDF\_type_i$ /* *the PFDF a forest or a tree?* */

`Provides`*:* *digraph:* $\mathcal{D}$ /* *CCDG for* $\mathcal{P}$ */

`Declares`*:* *arc set:* $\{IF_i \mid P_i \in \mathcal{P}\}$ /* *set of indfdom arcs* */

arc set: $\{R_i \mid P_i \in \mathcal{P}\}$ /* *set of arcs removed from PFDF* */

digraph: $\mathcal{CG}'$ /* *contracted call graph for* $\mathcal{P}$ */

`begin` *Construct_CCDG*

/* *begin by inserting invocation control dependence arcs for each call site.* */

`foreach` $P_i \in \mathcal{P}$ `do`

`foreach` *call site annotation c on arcs* $(v_{P_i}, v_{P_j}) \in A_{\mathcal{CG}_G}$ `do`

/* *insert inherited control dependence arcs from each call site to* G_j. */

$A^{dicd} \leftarrow A^{dicd} + (c, v^I_{G_j})$

`end foreach`

`end foreach`

/* *Figure 4.3 shows the changes to the PCDG after computing invocation control dependencies for procedures A, B, and C that make up the example program.* */

/* *There are two primary cases to consider when resolving pfdom arcs: the call graph may be a directed, acyclic graph (DAG), indicating the absence of recursive calls among procedures, or the call graph may have back arcs forming call cycles among procedures, indicating direct or indirect recursion. In the absence of recursive calls, resolution of pfdom arcs is straightforward and is accomplished by performing a post-order traversal over the call graph. The second case is more complex, requiring the identification of strong components in the graph and resolving related pfdom arcs based on the potential for extensive interactions among the procedures included in the strong components.* */

/* *resolve pfdom arcs* */

$\mathcal{CG}' \leftarrow Contract(\mathcal{CG})$

`foreach` *vertex* $V_i \in \mathcal{V}_{\mathcal{CG}'}$, *processing in a post-order traversal* `do`

/* *If there is only one vertex in a vertex of* $\mathcal{V}_{\mathcal{CG}'}$, *then the called procedure is not involved in a recursive call, so just check control dependence of* v^R. */

`if` $|V_i| = 1$ `then`

/* V_i *represents a single vertex in* $\mathcal{V}_{\mathcal{CG}}$, V_i *is associated with procedure* P_j */

`if` *PFDF_type for* P_j = *"F"* `then`

/* *If* v^R *is control dependent other than on* v^I, *then "remove" pfdom arcs at related call sites and change PFDF_type to F for each calling procedure.* */

`foreach` V_s *such that there exists* $(V_s, V_j) \in A_{\mathcal{CG}}$ `do`

`foreach` $u = (V_s, V_j).cs\_ID$ *and* $(u, v) \in I_s$ `do`

$R_s \leftarrow R_s + (v, u)$

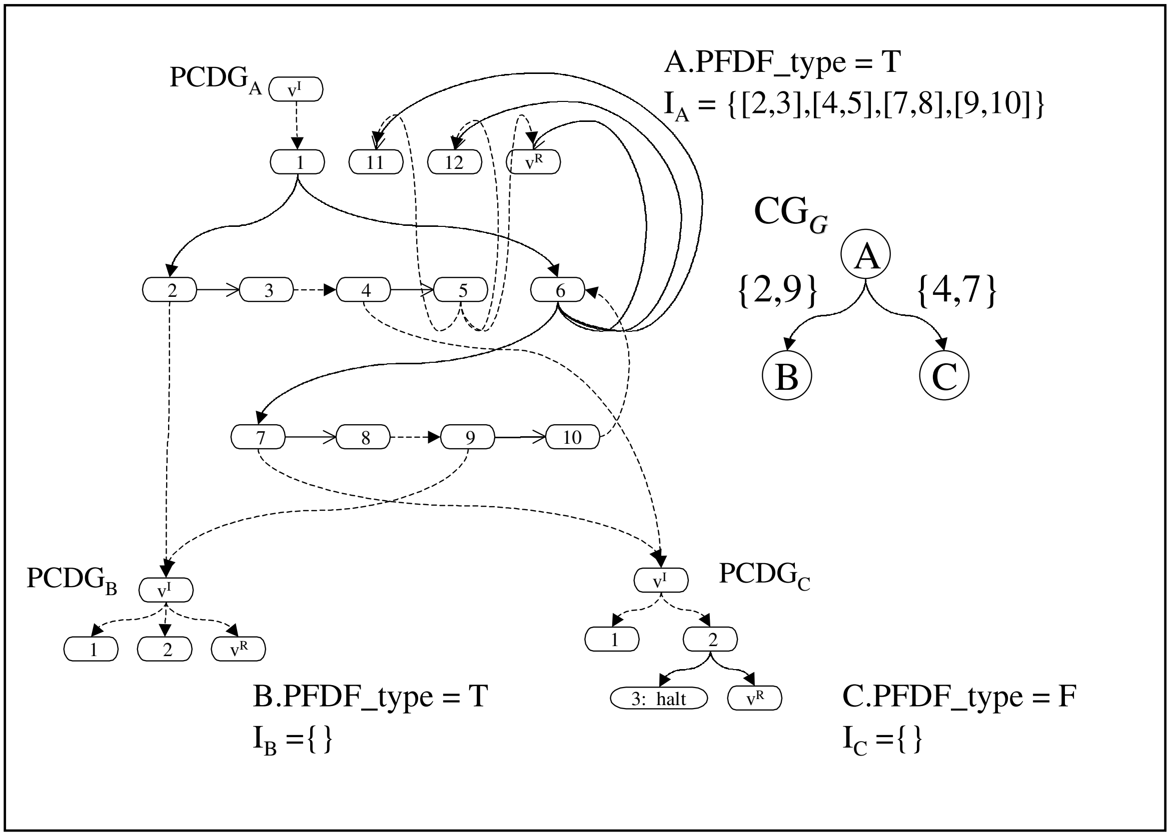

Figure 4.3: Example CDG After Computing Invocation Dependencies.

$PFDF\_type_s \leftarrow$ *"F"*
`end foreach`
`end foreach`
`end if`
`if` *PFDF_type for* P_j = *"T"* `then`
/* *If* v^R *is not control dependent other than on* v^I *then "replace" pfdom arcs at related call sites add to list of indirect forward dominance arcs.* */
`foreach` V_s *such that there exists* $(V_s, V_j) \in A_{\mathcal{CG}}$ `do`
`foreach` $u = (V_s, V_j).cs\_ID$ *and* $(u,v) \in I_s$ `do`
$IF_s \leftarrow IF_s + (v,u)$
`end foreach`
`end foreach`

```
      end if
    end if
    /* If there is more than one vertex in a vertex of a contracted call graph,
       then the called procedure is involved in a recursive call and, if any procedure
       exhibits an internal control dependence, then PFDF_type for each procedure
       is set to "F". */
    if |V_i| > 1 then
      for 1 ≤ k ≤ |V_i| do    /* V_k ∈ V_i = some V_j ∈ V_CG */
        if PFDF_type of P_j = "F" then
          foreach V_s such that there exists (V_s,V_j) ∈ A_CG do
            foreach u = (V_s,V_j).cs_ID and (u,v) ∈ I_s do
              R_s ← R_s + (v,u)
              PFDF_type_s ← "F"
            end foreach
          end foreach
        end if
        /* If after checking all internal procedures no control dependence is found,
           then inherited control dependencies exist between pfdom source and
           target for each call site to any procedure represented by the vertex
           of the contracted call graph. */
        if PFDF_type for P_j = "T" then
          foreach V_s such that there exists (V_s,V_j) ∈ A_CG' do
            foreach u = (V_s,V_j).cs_ID and (u,v) ∈ I_s do
              IF_s ← IF_s + (v,u)
            end foreach
          end foreach
        end if
      end for
    end if
  end foreach
```

/ Figure 4.4 shows the changes to the PCDG after resolving pfdom arcs as well as the addition of the sets listing the indirect forward dominance arcs that are conceptually added and the list of pfdom arcs that are conceptually removed from to the PFDF of procedures A, B, and C that make up the example program. Note that the PFDF_type of procedure A has been changed to "F". */*

/ Next, for each procedure and for each ifa arc in I, check to see if the related pfdom arc has been removed from F_i during pfdom resolution. If it has been removed, then a resumption dependence exists between the return of the called procedure and the ifa-target. If it turns out that this ifa arc is in a loop and the potential for non-return is discovered, then we need to convert the potential*

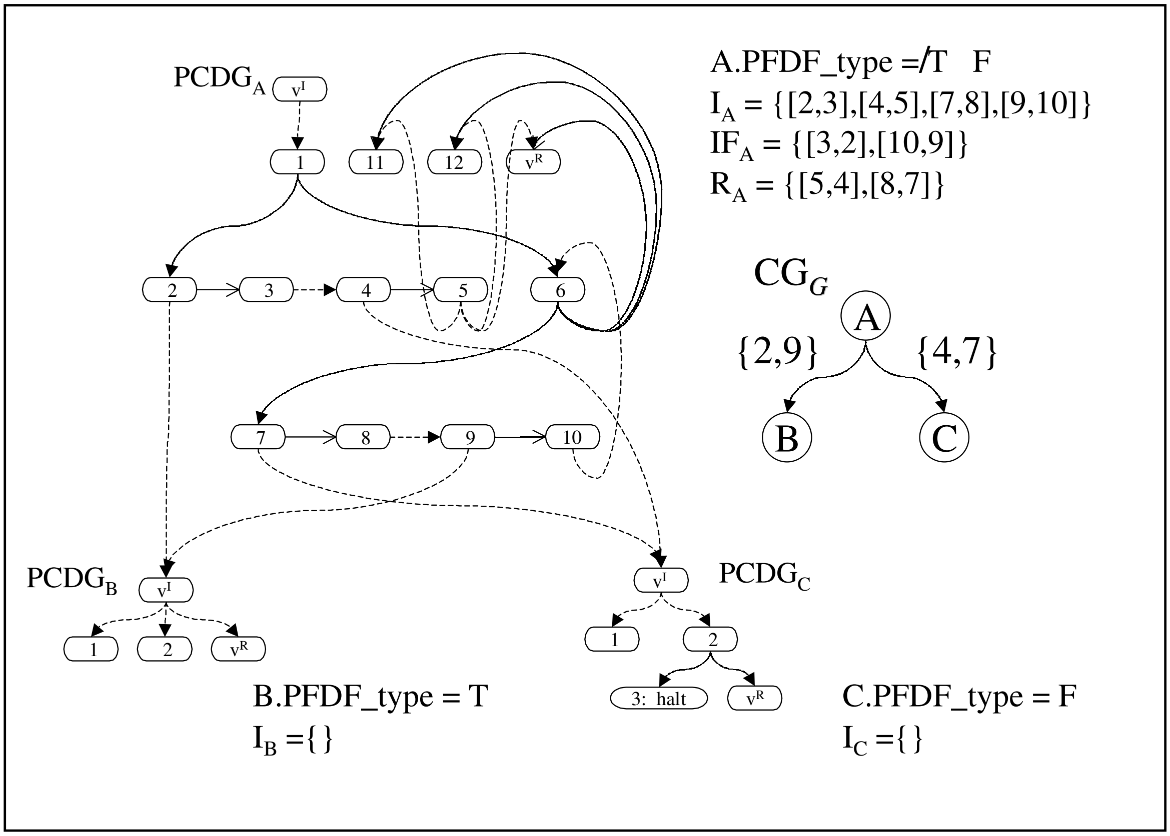

Figure 4.4: Example CCDG After Resolving Potential Forward Dominance Arcs.

*dependencies adjacent from the loop header to direct control dependencies, otherwise delete the potential dependence arc. If it is determined that all calls return, then all potential dependence arcs will have been removed, otherwise at least one will remain. */*

/ compute resumption control dependence */*

```
foreach P_i ∈ 𝒫 do
   foreach a = (u,v) ∈ I_i do
      if (v,u) ∈ R_i then
         A^dicd ← A^dicd + (v^R_{called_proc_{G_i}}, v)
         A_{D_i} ← A_{D_i} − a, for each a ∈ D_i that is incident to v
         if (w,u) ∈ A_{D_i} then
```

/* *arcs incident to* u *in* D_i *are converted from potential to directly control dependent* */

foreach $(w,v_j) \in A^{dpcd}$ do
  $A^{dcd} \leftarrow A^{dcd} + (w,v_j)$
  $A^{dpcd} \leftarrow A^{dpcd} - (w,v_j)$
end foreach
end if
end if
if $(v,u) \in IF_i$ then
  $A^{dicd} \leftarrow A^{dicd} + (u,v)$
  /* *arc incident to* u *in* D_i *are added to inherited control dependent* */
  if $(z,u) \in A_D$ then
    foreach $(z,v_j) \in A^{dpcd}$ do
      $A^{dpcd} \leftarrow A^{dpcd} - (z,v_j)$
      /* *remove potential control dependence of any siblings of* u */
    end foreach
  end if
end if
$A^{dpcd} \leftarrow A^{dpcd} - (u,v)$
end foreach
end foreach
end *Construct_CCDG*

Figure 4.5 shows the state of all structures after computation of composed control dependencies. The final step is to traverse in reverse order over each chain of inherited dependence arcs to determine the source of each direct control dependence. The results of identifying direct control dependencies is shown in Figure 4.6. This figure contains the final results of calculating the control dependence graph for our example program.

4.2 Complexity

We now compute the complexity of the suggested algorithms. We begin by presenting an overview of the detailed analysis that follows in Section 4.2.2.

4.2.1 Overview

The overall complexity of computing the control dependence graph is driven by the complexity of constructing the procedure control dependence graphs associated with procedures of a program. Therefore, we begin our analysis by determining the complexity of constructing the structures associated with a procedure. Next we compute the sum for all procedures in a program, and add this to the complexity of composing the program's control dependence graph. The construction of

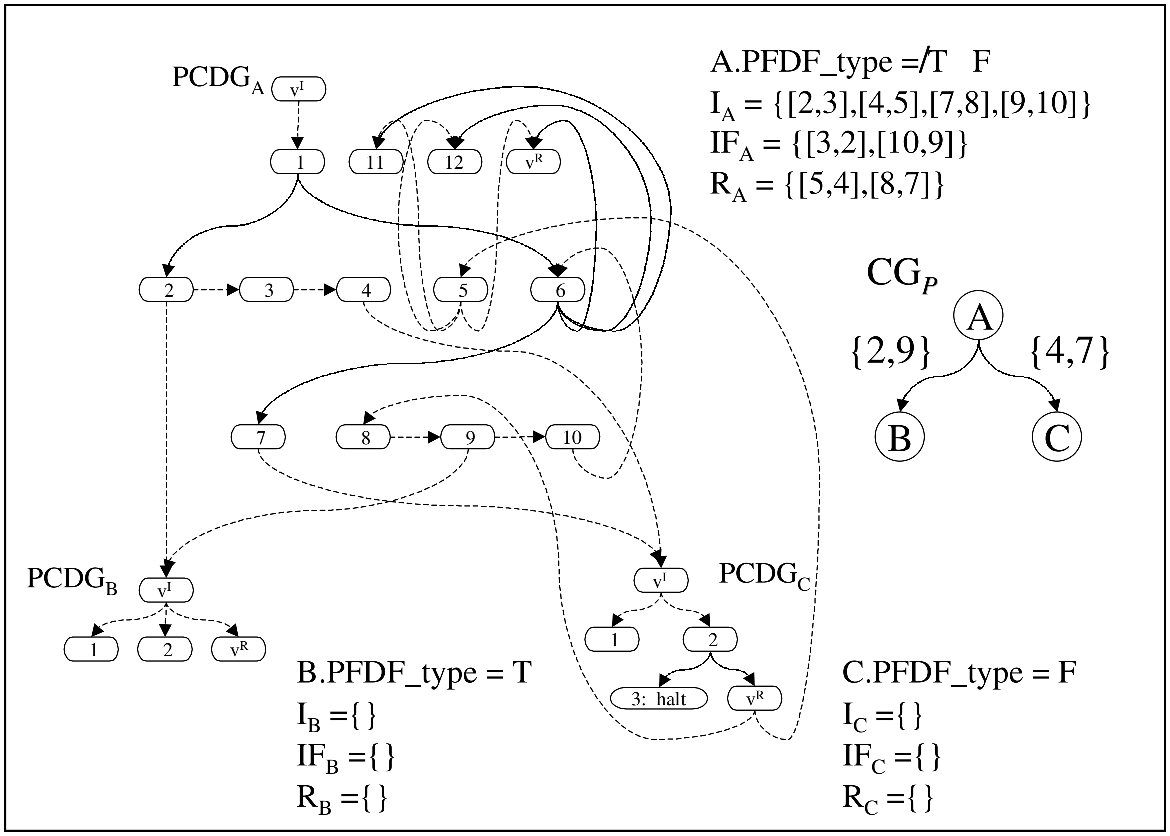

Figure 4.5: Example CCDG After Computing Resumption Dependencies.

the control dependence graph assumes the existence of procedure control flow graphs, as defined in Chapter 3, for each procedure in the program.

Throughout our analysis we use the following variables. Others may be defined and redefined as needed through the chapter.

- n = number of procedures in $\mathcal{P}$
- Program $\mathcal{P} = \{P_0, P_1, P_2, \ldots, P_{n-1}\}$ where P_0 is the main procedure in the program.
- A set of procedure control flow graphs $\mathcal{G} = \{G_0, G_1, \ldots, G_{n-1}\}$ containing exactly one PCFG for each procedure in $\mathcal{P}$.
- l_i = number of vertices in V_{G_i}

- m_i = number of arcs in A_{G_i}
- $L = \sum_{i=0}^{n-1} l_i$
- $M = \sum_{i=0}^{n-1} m_i$

The algorithmic complexity of constructing the composed control dependence graph is driven by the complexity of computing the direct control dependencies for the individual procedures. We base our complexity calculations on the use of Ferrante et al.'s algorithm for computing direct strong control dependencies and Podgurski's algorithm for computing direct weak control dependencies. The worst case analysis for these are $O(m^2)$ *and* $O(m^3)$ respectively as calculated by the authors of those algorithms. The major steps of the computations required for constructing the composed control dependence graph and their algorithmic complexity is given below; the complexity for each step is listed along the right margin. Then the overall complexity is computed based on these figures . A detailed analysis of the complexity is presented in the following section.

1. Assume: $\mathcal{G}$ a set of PCFGs, one for each procedure in the program $\mathcal{P}$, and $\mathcal{CG}$ a call graph for $\mathcal{P}$
2. Construct the contracted call graph for $\mathcal{P}$ — $O(L)$
3. For each $G_i \in \mathcal{G}$, we construct the following:
   (a) Procedure forward dominance forest — $O(M \log L)$
   (b) Procedure control dependence graph — $O(M^3)$
4. Construct the composed control dependence graph for $\mathcal{P}$ — $O(M + L^2)$

Since $L \leq M$ we can safely substitute M for L, adding together the complexity of the individual steps we get the equation:

- $O(M \log M) + O(M^2) + O(M^3)$

Because the highest degree polynomial value grows much more quickly than any lower one, we calculate the asymptotic complexity to be $O(M^3)$. There are two O^3 computations: (1) the Podgurski and Clarke weak control dependence algorithm and (2) step 19 of Construct_PCDG that looks for calls within loops and, if it is determined that the call may not return, reassigns dependencies of forward dominators of the loop header.

4.2.2 Detailed Analysis

In this section we present details to support the overview given above. We refer to the line numbers associated with the algorithms given in Section 4.1.

Program Forward Dominance Forest

The analysis is done on a per procedure basis and then combined, assuming M and L to be the worst case for the largest procedure, since that would be the case if $\mathcal{P}$ were a uni-procedure program.

Analysis of algorithm 1 goes as follows:

$O(l)$	step 1: At worst case one arc is created incident to each vertex in the graph. There are two operations for each arc so, total operations is at worst $2 \times l$.
$O(m \log l)$	step 5: As calculated by Lengauer and Tarjan.
$O(1)$	step 6: Results in one assignment
$O(l)$	step 11: In the worst case all trees are comprised of just a root, resulting in l additions to the list, which can be done in constant time.
$O(l)$	step 14: In the worst case all vertices are ifa-targets, resulting in l conversions involving a deletion from one set of arcs and an addition to another, each of which can be done in constant time. is a constant time operation.

Overall complexity of creating a PFDF is

$= O(1) + 3 \times O(l) + O(m \log l)$

$= O(l) + O(m \log l)$

$= O(m \log l)$ since $m_i > l_i$ for all $P_i \in \mathcal{P}$.

The worst-case complexity for computing PFDFs for all procedures in $\mathcal{P}$, $\sum_{i=0}^{n-1} O(m_i)$.

Let $m_k = max(m_i)$. Then worst-case complexity for computing forward dominance forests for all $P_i \in \mathcal{P}\ is\ O(m_k) = O(M)$ since $m_k \leq M$.

Procedure Control Dependence Graph

Again, the analysis is done on a per procedure basis and then combined assuming M and L to be the worst case for the largest procedure, since that would be the case if $\mathcal{P}$ were a uni-procedure program.

Analysis of algorithm 2 goes as follows:

$O(l^2)$ If step 2 is executed.

or

$O(l^3)$ If step 5 is executed.

$O(1)$ step 7: $2 \times O(1) = O(1)$

- $O(1)$ step 8: One list element insertion.
- $O(1)$ step 9: One list element removal.

$O(m)$ step 12: Requires at most m insertions to the set

$O(m^3)$ step 15: There are at most m members in T and at most a cost of $O(m^2)$ for each at step 16.

- $O(m^2)$ step 16: There are at most m vertices each requiring at most one check (step 17) and one insertion (step 24), each at constant cost, and $O(m)$ for step 23 producing an overall cost of $O(m^2)$.
  - $O(l)$ step 17: Requires at most l checks and one removal for each vertex found to be in both sets.
    - $O(l)$ step 19: Requires at most l list element deletions.
  - $O(1)$ step 22: Requires at most one insertion.
  - $O(m)$ step 23: Requires at most l operations to determine membership and at most cost of $O(m)$ at step 24 if true. Since $m \geq l$ we get $O(m)$.
    - $O(m)$ step 24: There are at most m ancestors in a PFDF and at most one insertion for each.

Summing the complexity for the steps above gives us the formula:

$= O(1) + O(m) + [O(l^2) \text{ or } O(l^3)] + O(m^3)$

$= O(m^3)$ since, as above, we take $m \geq l$

Composed Control Dependence Graph

The cost of adding the inherited control dependence arcs for invocation dependence involves a maximum number of operations equivalent to the maximum number of call sites, which is less than L, the number of statements in the program. Thus, at worst the complexity will be $O(L)$, the constant cost of adding an arc for each statement.

While resolving pfdom arcs there will be a constant number of operations for each arc in $A_{CG_{\mathcal{P}}}$ and $|A_{CG_{\mathcal{P}}}| \leq M$. Pfdom-arc resolution requires contracting the call graph as well as resolving the pfdom arcs thus, the total cost for pfdom-arc resolution is $O(M) + O(M) = O(M)$.

While computing resumption control dependence there are at most two arc set insertions and one removal for each interrupted-flow arc. Thus, the worst case complexity is $O(M)$, since M is greater than or equal to the number of arcs in the program.

4.2.3 Summary

In summary the costs incurred to obtain the benefits of compositionality are minimal if any. We have shown that in the worst case our compositional approach supports building algorithms that are reasonably efficient. The worst case time is no worse than that for computing weak control dependence as reported by Podgurski and Clarke and, in practice, should be nearly as efficient as Ferrante et al.'s control dependence algorithm when computing strong control dependencies. This is because the cubic cost at step 15 is likely to be small in practice, since the size of the sets being searched will, in general, have few members. Thus we believe the complexity for strong control dependence would be closer to $O(m^2)$.

Given the correctness of this assumption, complexity of our algorithms is heavily reliant on the algorithms used to compute strong and weak control dependence. If more efficient algorithms than those described here are used, then our algorithms would likewise be improved.

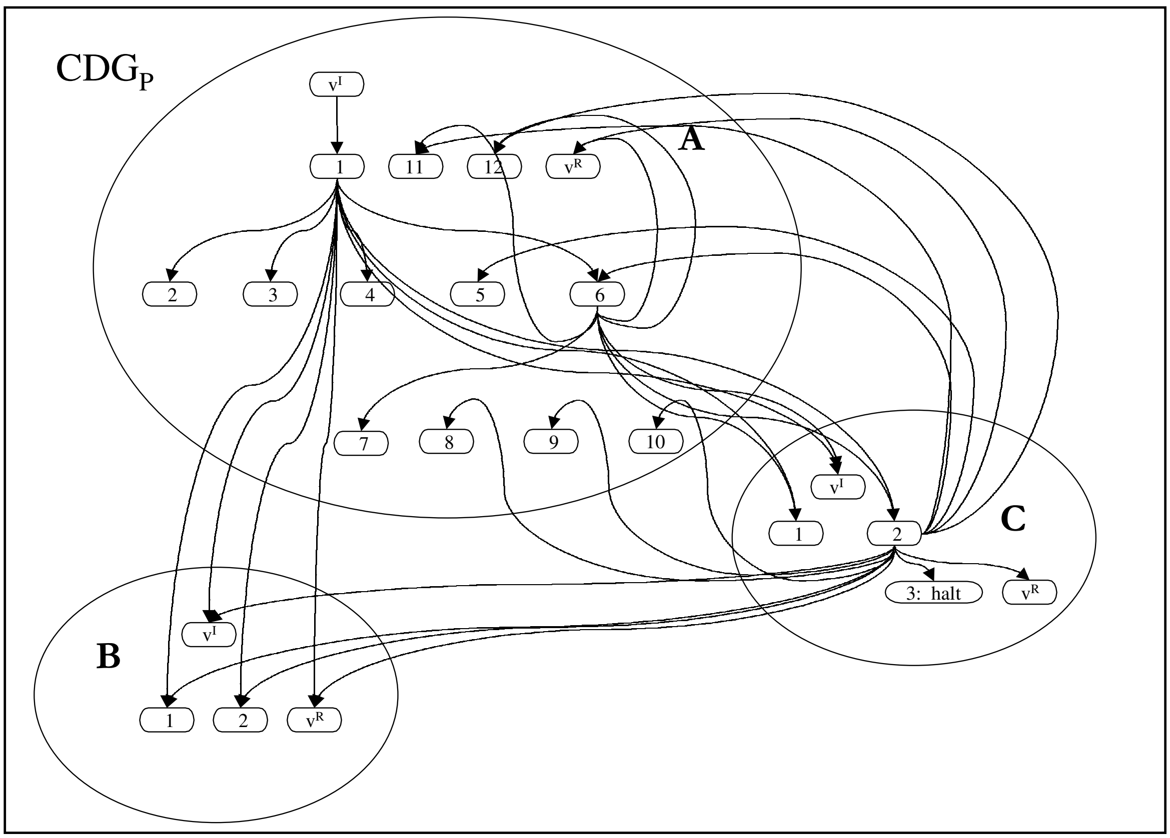

Figure 4.6: Example CCDG After Resolving Indirect Dependencies.

Chapter 5

Evaluation

We evaluate the model in several ways. First, we evaluate its correctness with respect to its ability to support modeling of all control mechanisms supported by commonly used, sequential, imperative programming languages. We show that the same set of control dependencies are identified through application of this compositional model as those obtained using traditional control dependence applied to an inlined control flow graph. Second, we evaluate its ability to address the pitfalls associated with performing control dependence analysis on non-inlined control flow graphs. Third, we evaluate its value relative to other, published, interprocedural analysis algorithms in terms of its time and space complexity. Fourth and finally, we describe its usefulness in terms of its ability to identify conservative and reasonably precise sets of dependencies, its simplicity, and its satisfaction of our guiding principles of compositionality, formality, and language independence.

5.1 Correctness

The model presented in this dissertation is sufficient to support creating algorithms to identify dependencies in commonly used sequential, imperative, programming languages that include procedure calls. The model does not support identification of dependencies that result from non-local jumps other than halts. We do not consider this to be an unreasonable restriction because the model is intended for use primarily with programs that are composed of procedures that have been created in isolation, and therefore, are not expected to have information about possible jump targets in other procedures.

It has been shown that sequence, decision/junction, and loop are sufficient for representing all control mechanisms supported by commonly used sequential, imperative programming languages [12, 34]. We add procedure call/return to this list because we focus on non-inlined control flow representation. To show that our model is language independent within this class of languages we must show that it provides support for addressing the constructs of these types of languages. Since intraprocedural dependencies are identified using algorithms previously proven to be language independent, it is sufficient to show that control constructs associated with procedure calls are addressed. Theorem 5 assures us that our model identifies the same set of dependencies as an

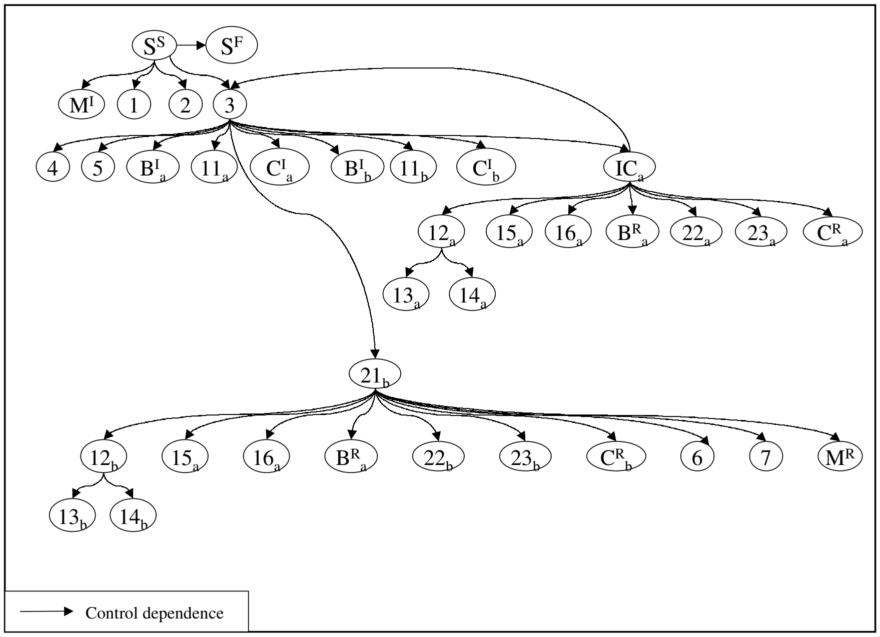

Figure 5.1: Control Dependence Graph for Sum Calculated from Sum's Inlined Control Flow Graph.

inlined interprocedural control flow representation would. We illustrate this characteristic of our model by creating control dependence graphs for two example programs using both inlined control flow graph representations and our model and show the results to be the same.

Consider Figure 5.1 containing the control dependence graph for program Sum constructed by computing strong control dependencies of the inlined control flow graph representation of Sum. When viewing multi-program statement dependencies in terms of S-control dependence as suggested by Harrold et al., described in Section 2.2.5, this control dependence graph is identical to the one shown earlier in Figure 3.9 on Page 52 except that S^S and S^F are not shown in Figure 3.9. We include a revised version of that graph shown in Figure 3.9 as Figure 5.2. In Figure 5.2 the graph is laid out in the same fashion as the graph in Figure 5.1 for ease of comparison. The revised version combines the vertices created by inserting copies of the procedure control flow graphs at various call sites into one vertex to reflect the full range of dependencies for any given vertex. The inlined control flow graph and the forward dominance tree for the inlined control flow graph are shown in Figures 5.3 and 5.4 respectively.

As another example, Figure 5.5 shows the control dependence graph for the example shown in Figure 4.1 computed based on an inlined control flow graph. Figure 5.6 shows the control dependencies as computed using our model. They reflect the same set of dependencies when

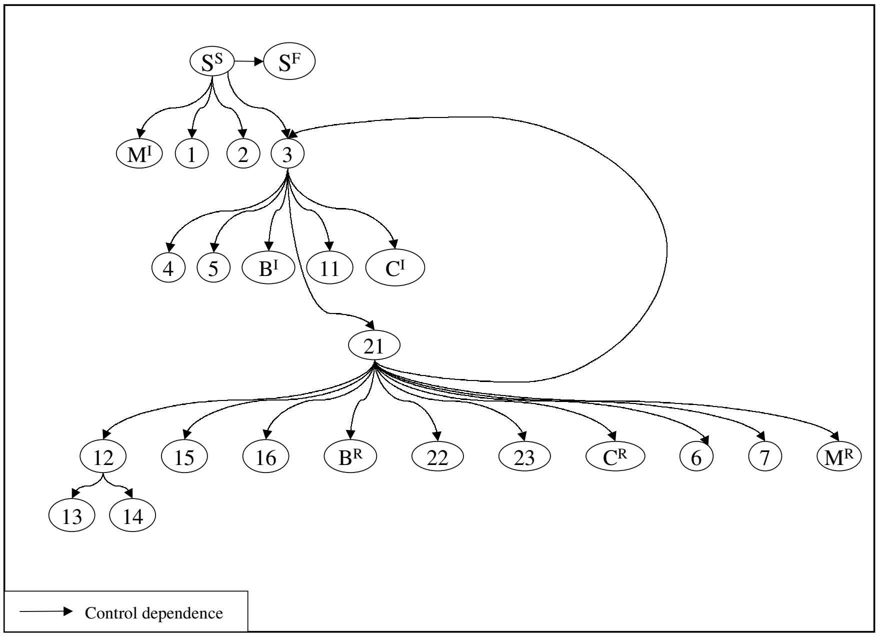

Figure 5.2: Revised Version of Graph Shown in Figure 3.8 Laid Out to Simplify Comparison with Graph Shown in Figure 5.1.

applying Harrold et al.'s definition of S-control dependence. The inlined control flow graph and the forward dominance tree for the inlined control flow graph are shown in Figures 5.7 and 5.8 respectively.

5.2 Addressing the Pitfalls

In Section 2.4 we described several issues that create difficulty when developing algorithms to identify control dependencies in multi-procedure programs. In this section we describe how these issues are addressed in our model.

Incomplete programs There are two reasons that the source code of a required procedure may not be available when one wishes to perform dependence analysis: first, the procedure may not have been written yet; and second, the procedure may only be available in the form of precompiled components. In the case of unwritten procedures the analyst must choose either to assume eventual return from the procedure call, or to take the conservative, but excessively imprecise, approach of assuming non-return. In the case of precompiled components, we propose that the required structures be supplied along with the component in support of dependence analysis. If this is

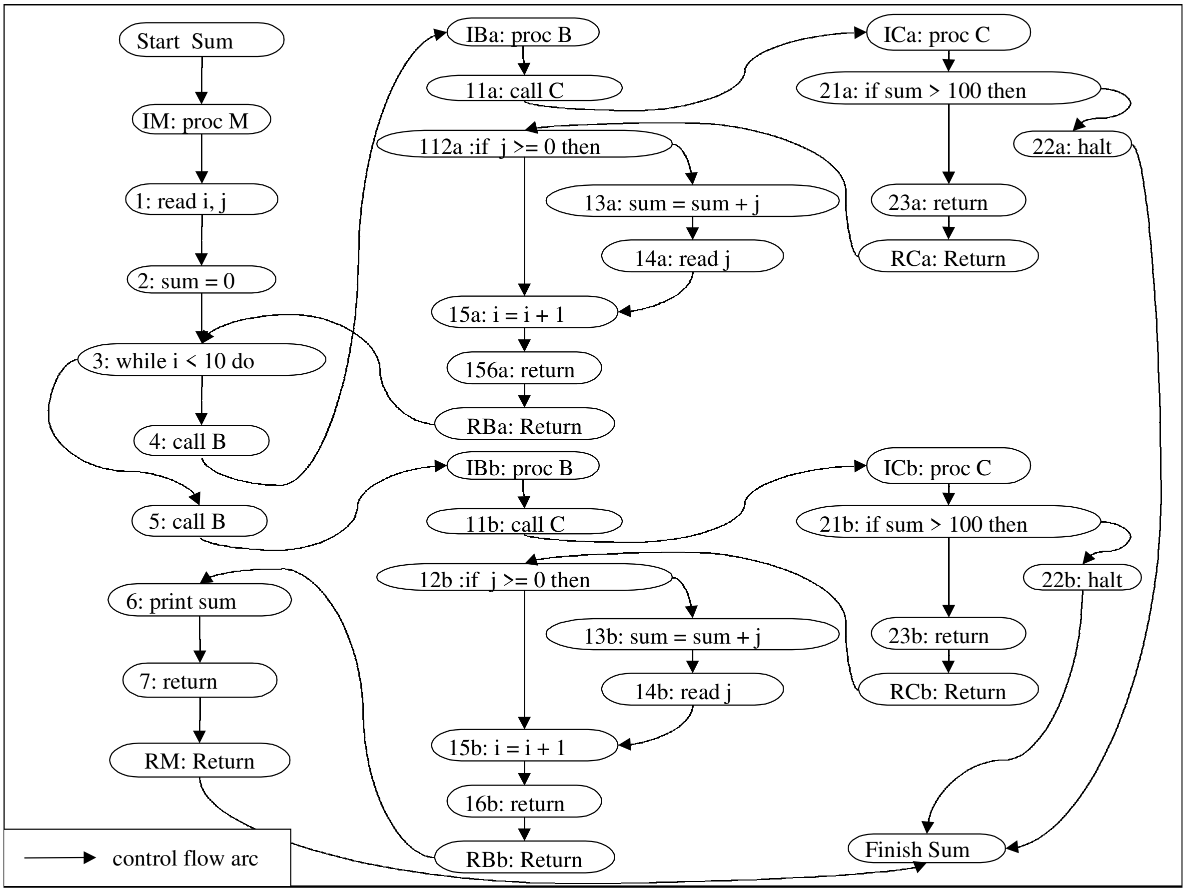

Figure 5.3: Inlined Control Flow Graph for Program Sum.

not the done then precompiled components present the same difficulty as we just described for unwritten procedures.

Function pointers Improving precision of control dependence analysis with respect to function pointers involves performing alias analysis and is, therefore, outside the scope of this thesis. Our model allows one to choose either to restrict the use of function pointers or to accept the imprecision associated with taking a conservative approach that does not involve attempting to identify aliases. This approach is not generally acceptable because it involves considering all possible procedures to be potential targets of indirect procedure calls. This issue will be addressed when a data dependence model is developed and used in combination with this control dependence model.

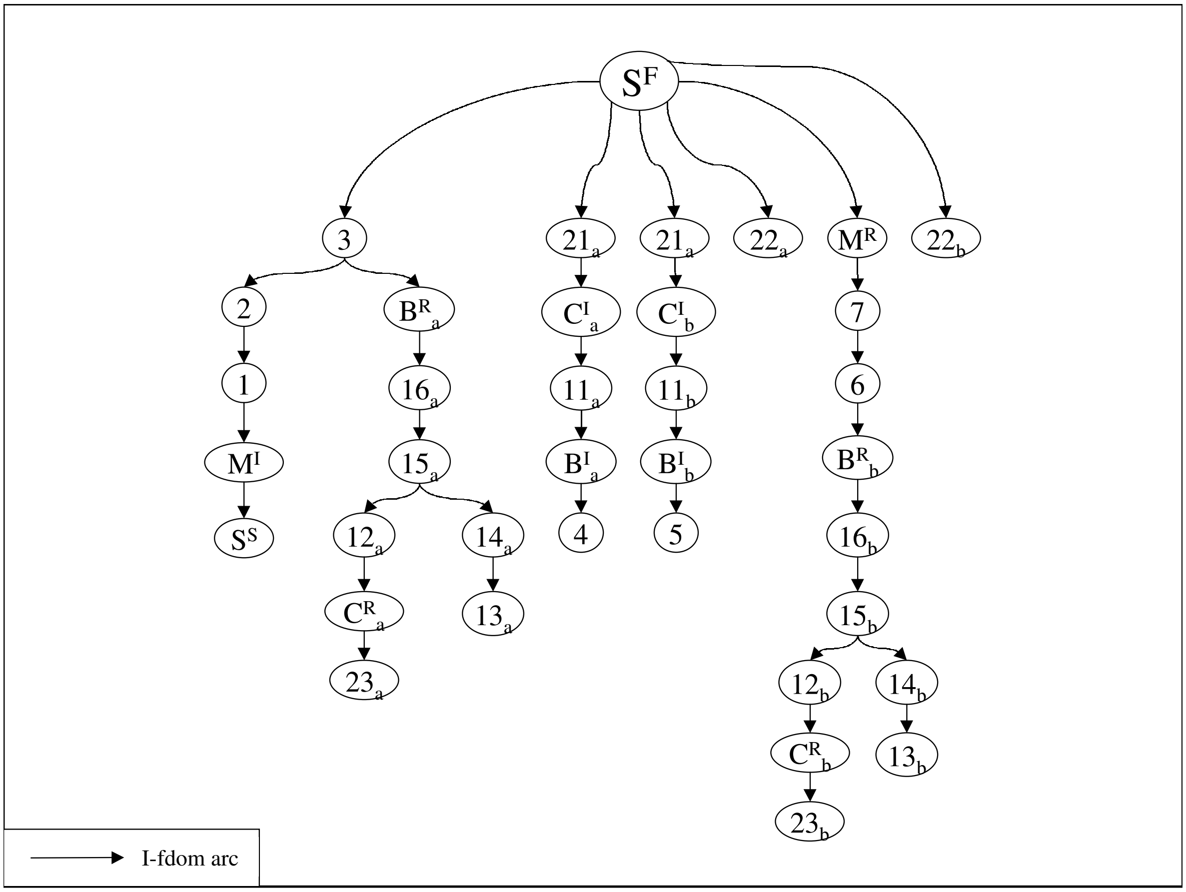

Figure 5.4: Forward Dominance Tree for Sum's Inlined Control Flow Graph.

Valid walks Identification of valid walks requires that an algorithm ensure that traversal over the control flow graph contain sequences of vertices that represent valid execution paths in the program. Return vertices may be incident to many arcs, each of which is associated with one call site; the one chosen for return must match the current call. In our model, the definition of interprocedural walk provides guidance for creating algorithms that only traverse the graph by way of walks that represent valid execution paths.

Calling context Both control and data dependencies within procedures depend on those that exist at call sites. Additionally, control and data dependencies of statements to be executed upon return from the procedure call may exhibit dependencies on statements that are executed as a result of the procedure call, or may in fact cross the procedure call boundary and return to the calling procedure. It is this potential that requires that only valid paths be traversed when identifying

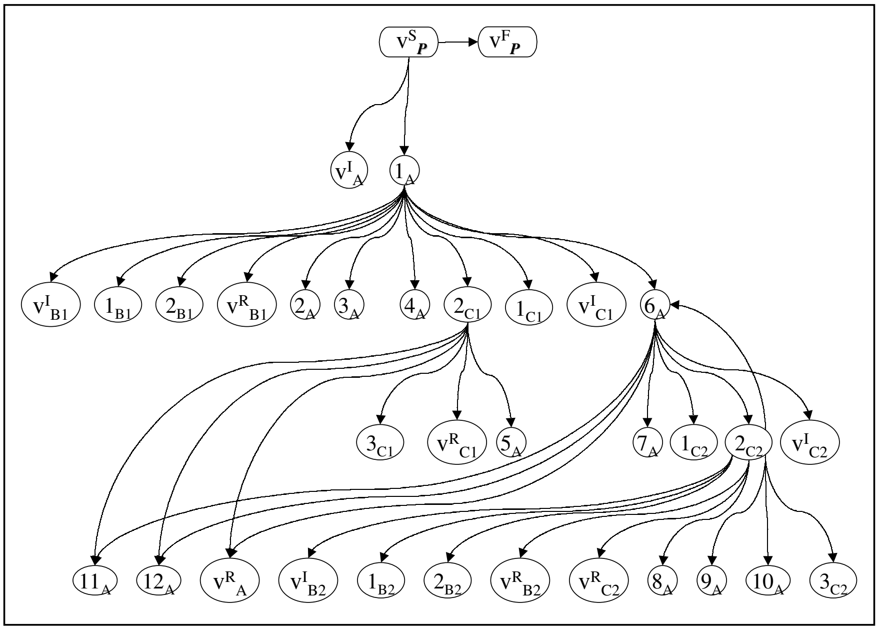

Figure 5.5: Control Dependence Graph for Example Program Shown in Figure 4.1 Calculated from Its Inlined Control Flow Graph.

dependencies using a non-inlined interprocedural control flow representation. The model includes interrupted-flow arcs in procedure control flow graphs, which provide for identification of potential control dependencies that can be resolved during program composition if it is determined that a control dependence exists on a statement inside a descendant in the call graph. Harrold et al. [22] define an additional type of context effect that they refer to as the "multiple call site effect". Rather than view these as separate effects, we recognize that they are both a result of the need for dependencies in a called procedure to reflect dependencies of the procedure's call sites and for subsequent dependencies within the calling procedure to reflect those of the return from the called procedure. By taking this view we automatically mimic the use of an inlined control flow representation in which there is an immediate forward dominance between call site and procedure entry for each vertex representing the entry to the procedure and from each vertex representing the return of the called procedure to the vertex following the call site in the calling procedure.

Potential for Non-Return There are three reasons why control may not return after a call to a procedure: embedded halts, non-termination of loops, and infinite recursion. In our model the control dependence of the return vertex of a procedure control flow graph is inherited by the ifa-target and other vertices that inherit their control dependence from the ifa-target. If the return

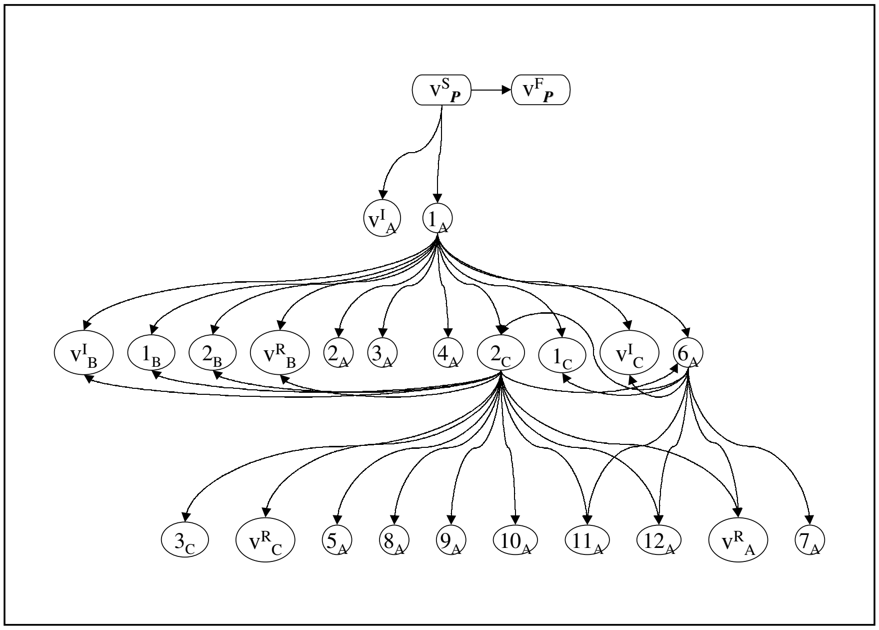

Figure 5.6: Reduced Version of Graph Shown in Figure 5.5 Representing Each Program Statement Once.

vertex of the called procedure does not exhibit an internal dependence, then the ifa-target inherits the dependence of its related ifa-source thus assuring that control dependencies only cross procedure call boundaries when necessary.

Our definition of interprocedural weak control dependence is based not only on the identification of weak dependencies within called procedures but also on detection of a multi-procedure strong component as a descendent of a procedure in the call graph. This provides a simpler means for detecting interprocedural weak control dependencies than the definition provided by Loyall and Mathisen [37]. It is not possible to compare the conceptual simplicity of the two algorithmically because Loyall and Mathisen did not present an algorithm for computing weak control dependence in their paper.

5.3 Relative Value of Approach

Four factors that contribute to the value of our approach are its conceptual simplicity, reusability, reasonable computational complexity, and reduced space complexity over other approaches.

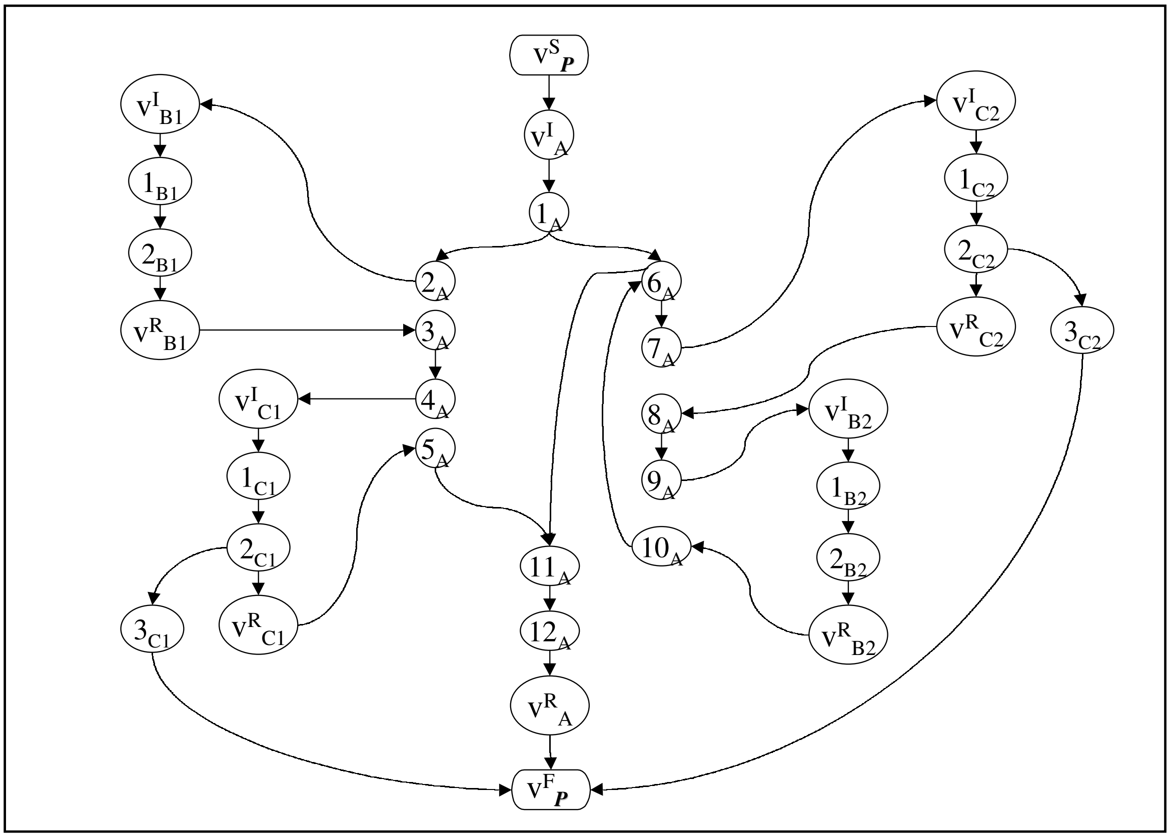

Figure 5.7: Inlined Control Flow Graph for Example Program Shown in Figure 4.1.

Simplicity This model is a simple extension to the already well-understood control dependence model used to compute control dependencies in uni-procedural programs. The conceptual simplicity of our extension makes it accessible to algorithm designers who have previous experience with either the Podgurski and Clarke model or the PDG.

Reusability The model was designed with reusability in mind. Computing control dependence information once and supplying it with components not only saves on recomputation of dependence information but also increases the reusability of procedures in that they can be supplied as interfaces, object code, and analysis results without exposing source code or control flow information.

Complexity of Algorithm We have shown that the complexity of algorithms that might be used to build an implementation of this model are directly related to the complexity of the algorithms

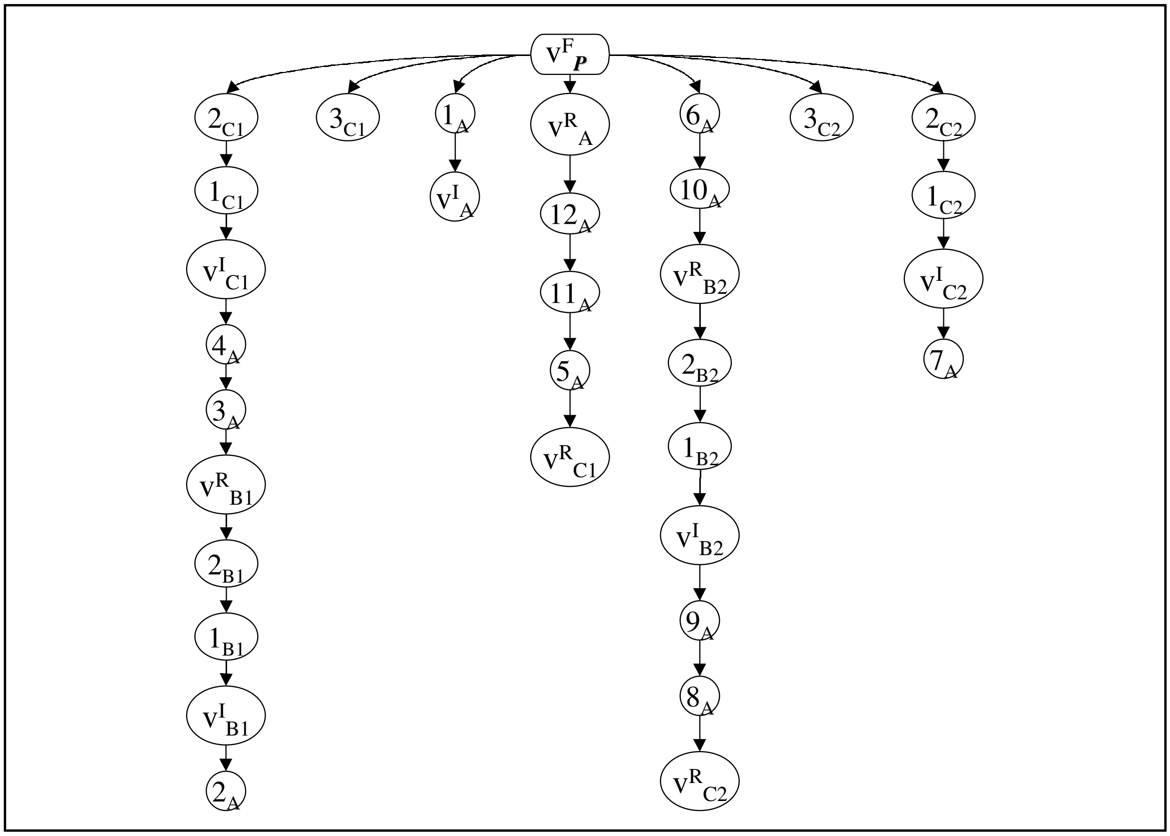

Figure 5.8: Forward Dominance Tree for Inlined Control Flow Graph of Example Program Shown in Figure 4.1.

used to create the forward dominance forest and the procedure control dependence graphs. Efficient algorithms exist to construct these structures and, therefore, the structures of this model can be computed efficiently.

Size of Representation The vertices and arcs required to construct the structures of this model are fewer than those required by in-lined models in that the relationships among the statements in a procedure need be recorded just once. In comparison to other non-inlined representations, the composed structure is minimal since it represent each statement in a program by exactly one vertex and adds only two vertices to each procedure, v^I and v^F, which are also used in other suggested representations. In fact, our composed control flow graph is at most the same size as other interprocedural control flow representations because we do not build an interprocedural control flow graph. Thus, we do not require the inclusion of procedure call and return arcs, nor does our model require the addition of various types of vertices and arcs that are typically added to deal with control constructs that were not handled by the base algorithm. For example, we do not add an equivalent vertex to the "PNR" vertex and the associated arcs incident to and from it that are used in the augmented control flow graph suggested by Harrold et al. [22] to represent the potential for non-return from a procedure due to embedded halts. Instead we allow a procedure control flow graph to have more than one vertex with zero out-degree.

5.4 Usefulness

Program dependence analysis is generally accepted as a useful form of analysis in the software engineering and programming languages communities. Beyond this general view, we recognize that a particular dependence analysis algorithm is useful if it is conservative and if it provides a reasonably precise set of dependencies. Forward dominance has previously been shown to support the identification of conservative sets of control dependencies when applied to inlined interprocedural dependence graphs [41]. We have shown that composed forward dominance preserves this property. As is the case with intraprocedural dependence analysis, the precision with which dependencies can be identified is related to the control structure of the program being analyzed; this is determined by the programming style of its author and, to some extent, by the programming language in which it is written. For example, support for exceptions provides opportunities for creating programs with extremely complex control structures over which it is very difficult to determine reasonably sized sets of dependencies based on static analysis.

The model described in this thesis provides improved usefulness in that it provides support for reasoning about procedure dependencies in isolation and supplying the required information as annotations of the procedure. This property supports the identification of more precise sets of dependencies in systems composed of precompiled libraries than is possible when using previously available dependence analysis algorithms that are based on more conservative assumptions about the relationships between inputs and outputs of procedures. The model is also more useful than previously defined for multi-procedural programs in that it provides algorithms for extending the notion of weak control dependence to address the potential for non-return due to non-terminating loops in, or infinite recursion among, subsequently called procedures.

The model was designed with simplicity and ease of understanding in mind. The three basic structures that have been extended and are associated with a procedure (the control flow graph, forward dominance tree, and control dependence graph) are already well understood by dependence analysis researchers. The additional features associated with the compositional model are quite intuitive and the rigorous development of the model assures us that they, in fact, are sufficiently rich to identify the desired set of dependencies. The formal definition of our model provides the necessary foundation for reasoning about both its power and its limitations.

The model is language-independent beyond the definition of a control flow graph. We have extended the control flow graph to include features required for representing control flow among procedures in a non-inlined manner including support for representing control flow due to exceptions. We have added the notion of annotated return vertex to support conveying information relevant to exception handling control constructs when developing interprocedural control flow graphs. Return vertices contain information indicating the potential for exceptional return and the type of exceptions that may be raised. In this type of language, ifa-targets are dummy predicate vertices that test for the potential and type of exceptional return. The control features for handling exceptions in a particular language must be represented similarly to a case statement. The correctness of the identified set of dependencies depends upon the faithful representation of control constructs available to programmers using a given language.

Chapter 6

Related Work

Static analysis is widely recognized as being useful for reasoning about correctness properties of programs and has therefore provided a wealth of opportunities computer science research over the past several decades. In this chapter we provide descriptions of projects that are closely related to the work described in this dissertation, including comparisons to our work.

6.1 Foundations

In this dissertation we have described a model for control dependencies in sequential, imperative programs that contain procedure calls. In Chapter 2 we described many dependence analysis research projects that are intended for application to other types of programs. We described the works of Ferrante et al. [17] and of Podgurski and Clarke [41] that have been used as a foundation for the development of our model; our model is a direct extension of the control dependence models described in each of these works. In this section we compare our work to other projects that have focused on the non-inlined approaches to dependence analysis of sequential, imperative, multi-procedure programs.

6.2 Interprocedural Program Dependence Analysis

In this section we describe other research projects that focus on the identification of dependencies in multi-procedure programs. Models of program dependencies for these types of programs have been developed by Horwitz et al. [26, 27], Loyall and Mathisen [37], and Harrold/Sinha et al. [22, 47]. The primary goal for developing each of these models has been to provide a means for identifying program dependencies in programs composed of multiple procedures for which the control flow among statements in the program is represented in a non-inlined interprocedural control flow graph. Each model is based on identifying direct dependencies among program statements and creating a graph to represent these in order to support the identification of sets of related dependencies by performing a transitive closure over the vertices of the dependence graph. The Loyall and Mathisen model is slightly different from the others because they are interested in identifying dependencies among procedures rather than among statements in a program. We begin by comparing these works to each other then compare them to our work.

These works are very similar to each other but differ in subtle ways. The models vary some in terms of the precise definition of an interprocedural control flow graph and in terms of the form of the dependence graph itself; in fact Horwitz et al. do not define an interprocedural control flow representation. One difference is the way in which call and return arcs are to be represented in the control flow graph. Harrold et al. split vertices that represent call sites into two vertices, a call vertex and a return vertex. Loyall and Mathisen use the call vertex for both purposes and use the definition of interprocedural walk to restrict the order of traversal of arcs incident to and incident from the call vertex. Horwitz et al. also retain the notion of "region node" of the PDG introduced by Ottenstein and Ottenstein [39] in the system dependence graph. In part these differences can be traced to the fact that the works were carried on by researchers who focus in different areas of computer science. The work of Ferrante et al. and Horwitz et al. was carried on primarily in the context of compiler optimization, thus their view that region nodes should be included in the model as they support opportunities for fine-tuning algorithms. The work of Podgurski and Clarke, and Loyall and Mathisen was carried out in the context of software engineering problems and is based on the desire to automatically identify which procedures of a system can impact other procedures in some way, thus their focus on the more abstract view of identifying dependencies among procedures instead of among statements of a program.

Each of these models exhibits approximately the same asymptotic time and space efficiency. Each control dependence graph can be computed in at worst $O(N^2)$ where N is the sum of the number of control flow arcs in each procedure of the program. We provide the only algorithm for computing weak control dependence. At worst case it can be computed in $O(N^3)$ time; it is more expensive than computing strong control dependence. Representations of the control dependence graphs are approximately the same size, but the Horwitz et al. model and our model do not require a control flow representation in order to traverse the control dependence graph. The Loyall and Mathisen and our models are somewhat smaller than the Horwitz et al. and Harrold/Sinha et al. models in that we do not use the notion of "region node". This feature of the PDG is desirable for certain applications of dependence analysis. It can be added to our model, and therefore we do not consider this to be a savings in our representation. There are differences in the size of representation used for data dependencies. These differences are discussed by Sinha et al. [47]. We do not discuss these here because data dependence is outside the scope of this dissertation.

6.2.1 The System Dependence Graph

Horwitz et al. suggest using a non-inlined control flow representation as a base for interprocedural dependence analysis. They were interested in creating program slicing algorithms for multi-procedure programs. The design goals related to parameter passing included providing a foundation for identifying data dependencies with minimal knowledge of other system components (i.e. procedures). This work identifies and provides one solution for keeping track of the calling context when traversing a non-inlined control flow representation; they use an attribute grammar approach to determine transitive relationships and add appropriate arcs to the system dependence graph to represent them. This data dependence model has been extended and refined by others such as Liao et al. [33] in conjunction with the design of an efficient implementation of dependence analysis algorithms in the SUIF compiler. The control dependence model developed by Harrold/Sinha et al. is also based heavily on that suggested by Ferrante et al.

A major distinction between the Horwitz et al. model and the other models including ours is that it does not address the possibility of non-return from procedure calls. Control dependencies are computed for each procedure in isolation and return of control is assumed. A control dependence is assigned from each call site to the entry of the called procedure, and no return control dependence is assigned. Our work extends their idea of composing the system's control dependence without requiring the building of a control flow graph representation for the whole program by addressing the difficulties associated with the potential for control dependencies to cross procedure call boundaries.

6.2.2 The Interprocedural Control Dependence Graph

Loyall and Mathisen developed a model for the purpose of performing static impact analysis on large software systems. They focus on identifying dependencies among procedures, but their work is similar to the others because their definitions and algorithms are based on statement-to-statement relationships and then generalized to procedure-to-procedure relationships. Of the three models being described in this section, our definitions most closely resemble their definitions because both models are based on the Podgurski and Clarke model. They provide a definition of interprocedural forward dominance based on the use of non-inlined interprocedural control flow graph, which is defined to be a collection of procedure control flow graphs, an initial vertex, and a final vertex. The procedures are connected by call and return arcs, the initial and final vertices are the only vertices with in/out degree zero, and any exit vertex has an arc incident from it to the final vertex. Their definition of forward dominance is identical to the definition of forward dominance given by Podgurski and Clarke except that they qualify walks as being "interprocedural walks", which were previously defined to be traversals over the graph that represent valid execution paths. They define a walk in terms of allowable sequences of vertices that is sufficiently restricted to identify only valid execution paths over the form of interprocedural control flow graph that they use. Our definition of interprocedural walk is similar to theirs but is simpler due to differences in our control flow graph representations.

In their interprocedural control flow graph, a call vertex is not only the source of call arcs but also the target of return arcs. Thus, a call vertex has two arcs incident from it: one to the initial vertex of the called procedure's graph and another to the vertex that is to be encountered first upon return to the procedure. This representation requires them to add two conditions to the definition of interprocedural walk: first, they must assure that upon entry to a call vertex from within the calling procedure, the call arc is selected for traversal; and second, if entry to the call vertex is from the called procedure then the non-call arc is selected for traversal. Our use of interrupted-flow arcs provides an alternate solution that we feel more accurately represents the control flow of the program and provides the required information for keeping track of the calling context. Our view also provides the foundation for calculating control dependencies in a compositional manner. We use the definition of interprocedural walk to prove that our definition of interprocedural forward dominance accurately represents the forward dominance relationships within the program and do not require further use of the notion of interprocedural walk or interprocedural control flow graph.

6.2.3 Extensions to Previous Approaches

Harrold/Sinha et al. define a new version of an interprocedural control flow graph to be a collection of "augmented control flow graphs". In their ISSTA paper [22] they recognized the fact that the use of an interprocedural control flow graph such as that used by Loyall and Mathisen, and applying traditional notions of control flow, do not support identification of sets of dependencies that preserve the semantic-syntactic relationship as described by Podgurski and Clarke [41]. They identify the multiple context effect associated with calling a procedure from a call site that exhibits a control dependence and then again from a vertex that is a forward dominator of the source of that control dependence. This is the situation that exists in the program Sum that we introduced as a running example in Chapter 2. They recognized that certain definitions of direct control dependence do not correctly identify the vertices in the called procedure as being control dependent because all paths in the control flow graph beginning from the vertex 4 to the final vertex of the graph pass through the vertices of the called procedure eventually by way of the call site at vertex 6. But the number of times that the vertices in the called procedure are executed depends on the decision at vertex 4; so they must be identified as being control dependent if the syntactic-semantic relationship is to be preserved. The use of an augmented control flow graph provides support for conservatively identifying sets of dependencies for programs with more complex interprocedural control flow than has been previously discussed in the literature.

An augmented control flow graph is a control flow graph in which each call vertex is split into two vertices, a "call" and a "return" vertex, and in which an "exit" vertex has been added that is the target of any embedded halts [22]. These additions allow the augmented control flow graph to represent the fact that control may not return from the called procedure. Additional dummy vertices and related arcs are added to represent control flow related to exceptions if needed [47]. Each of the Harrold/Sinha et al. papers describes an algorithm for computing control dependencies based on this control flow representation that identifies the same set of control dependencies as would be identified using an inlined control flow representation sufficient for representing the target control constructs (i.e., embedded halts and exceptions) and, therefore, that preserves the syntactic-semantic relationship for programs containing those constructs.

The Harrold/Sinha et al. projects are closely related to our work in that their goal is to create a model for interprocedural control dependencies that identifies conservative sets of control dependencies for programs with arbitrary control. Neither of their algorithms support computing procedure control dependencies in isolation. The algorithm described in [22] requires building an interprocedural control flow graph at the time of identifying interprocedural dependencies. The algorithm described in [47] requires the use of a set of control flow graphs along with the program call graph to identify the potential for interprocedural control dependencies. Therefore, although these algorithms identify the same set of control dependencies that our model supports, they do not provide support for composing control dependencies of procedures written in different programming languages nor do they support composing control dependencies without exposing what is, under certain conditions, an unacceptable amount of source code information.

6.3 Summary

In summary, each of these models provides some of the features of our model, but none provides them all. The SDG proposed by Horwitz et al. is a compositional approach and is language independent, but it does not support identifying control dependencies based on the potential for non-return from procedure calls, non-termination of loops, or exceptional returns, nor is it described in terms of a set of definitions that provides a foundation for extension. The ICFG proposed by Loyall and Mathisen is also a language independent model but does not address control flow associated with exceptional return. It is described rigorously, but their extension of forward dominance to interprocedural analysis does not reflect the nature of forward dominance among statements in procedures called from multiple call sites, therefore failing to identify conservative sets of control dependencies. Additionally, their model requires traversing an interprocedural control flow representation.

The two representations suggested by Harrold/Sinha et al. are direct extensions to the two described above. In the first case they provide a control flow representation that addresses control flow associated with Java exceptions and extend the SDG for application to this control flow representation, thus providing a language dependent model. Their extension to the ICFG addressed the difficulties associated with the multiple-context effect that creates problems for the ICFG and includes additional support for modeling embedded halts. As is the case with the ICFG, this model is not compositional as it requires traversing an interprocedural control graph in order to identify interprocedural dependencies.

Chapter 7

Conclusions and Future Work

In conclusion, we have shown that it is not only possible but also reasonable to approach interprocedural control dependence analysis in a compositional manner. We have shown that the proposed model supports efficient and reasonably precise sets of control dependencies. Additionally, this model supports identification of dependencies in isolation in support of analysis of systems composed of precompiled components.

A driving force for creation of this model is the need to develop automated methods for systems that exhibit complex control and communication structures; systems that may be composed of prefabricated components for which the source code is not made available, and systems that may in fact be distributed over a number of processors. This model brings us a step closer to identifying dependencies in systems that are composed of prefabricated components than has previously been available. We envision a paradigm in which component manufacturers identify potential dependencies with components and supply that information for use by developers when the components are composed into a system. The work in this dissertation provides a model for doing just this for the case that the components are composed of sequential, imperative programs if one is interested in just control dependencies. Before we can fully realize our ambition to perform compositional dependence analysis, a companion model for identifying data dependencies must be developed. Extensions to uni-procedure data dependence models have been explored by other researchers for application to multi-procedure programs. Developing a compositional model based on previous work and the work of this dissertation will provide support for reasoning about the ways in which statements in a program can affect or be affected by other statements in the program, that is, for performing compositional syntactic dependence analysis that preserves the syntactic-semantic relationship described by Podgurski and Clarke [41].

The model is straightforward and accessible to researchers with previous experience in the area of control dependence analysis. The model is a direct extension to an already formally defined and well understood model of control dependence for uni-procedural programs developed by Podgurski and Clarke [41]. The definitions and theorems of our model are not only founded on the structures of the Podgurski and Clarke model but they are also use similar terminology, making them accessible to researchers who have previous experience with the Podgurski and Clarke model. Additionally we employ reuse of algorithms that have previously been shown to be efficient and effective for performing specific tasks that are required in order to build the structures of our model. Taking the reuse approach provides an additional level of accessibility as well as providing a level of confidence as to the correctness and efficiency of the algorithms presented for constructing the structures of the model. Additionally, using a rigorous approach to describe the model provides support for reasoning about both the power and the limitations of the model.

Before this model will be useful in practice it will be necessary to develop the companion data dependence model and to create an implementation within a programming environment that will allow its automated application to actual programs. We envision many areas of follow-on research, some of which are directly related to this model, others inspired by it. We briefly describe the basic ideas for these projects below:

7.1 Compositional Data Dependence Model

As mentioned above, the most pressing need for follow-on research is the development of a companion compositional data dependence model. Previous work in interprocedural data dependence analysis [27, 33] can be leveraged for this purpose.

7.2 Implementation

After the data dependence model has been developed a natural next step is implementation of the models. Implementing the compositional dependence model in the SUIF compiler would allow immediate comparison to the interprocedural dependence analysis algorithms that have been developed in that environment. Liao et al. [33] have developed a tool for performing control and data dependence for the SUIF compiler. A compositional dependence analysis component can be developed based on our model using the SUIF intermediate representation. This component can replace the dependence analysis unit already used within the SUIF environment and results of the analysis of large programs could be generated and compared to those already produced by the compiler.

7.3 Extensions

The identification of dependencies in systems is a very important area of research. It is also a very difficult area. As mentioned in Chapter 2, the sources of dependencies and the types of things that cause dependencies varies with the type of system being analyzed. Even adding a relatively simple control structure such as the procedure call that is the focus of this dissertation causes difficulties when extending algorithms.

A rich area for future work is the investigation of limitations and requirements for extending this model further, or to create a new type of model, in order to address issues of other types of systems that are described in terms of additional control mechanisms. As mentioned in Section 2.2.5 work has begun in this area but there are many open research issues yet to be explored. For example, when considering a generalized notion of event-based interactions one realizes that the notion of forward dominance is not useful for reasoning about dependencies of systems that use this mode of communication among system elements because of the asynchronous nature of event interaction. One must determine what types of relationships are statically knowable among elements of the type of system to which the dependence analysis is targeted and must determine how knowledge of these relationships can be leveraged to improve system understanding. For instance, in an event-based system one might want to identify sets of events that are triggered as a result of observing a particular pattern of events. In this case the relationships are the means for an event to be identified as and element of a pattern of events and event triggering.

It may be that for some system types and/or analysis types it makes sense to relax the notion of conservativism in favor of lowered cost. If one is considering developing fault localization algorithms for event-based systems it probably makes sense to loosen this requirement in that the cost of not identifying a dependency simply means that the analysis did not provide the desired benefit but the precision of a conservative set is so low that the set is useless for helping locate the fault. However, if one is interested in reasoning about secure information flow, conservativism is required.

7.4 Component-Based System Dependence Analysis

We believe that this model will easily extend for application to systems described in terms of COTS (Commercial Off The Shelf) components that interact by way of synchronous communication mechanisms. We envision COTS components being supplied with forward dominance information, control dependence graphs, and interrupted flow arc information. With this information and a description of how the components are to interact one could perform dependence analysis on the system at a higher level of precision than is possible given only knowledge of the external connections among the components [51].

When identifying sources of dependencies for a particular output of a component it is helpful to be able to determine which inputs actually have potential to affect the output. If this is not possible then all inputs must be considered potential sources of dependence in which case a transitive closure backwards beginning at the output of interest will normally lead to inclusion of a very large portion of system components. For dependence analysis to be useful, one must be able to identify intra-component input-output pathways. One can view a procedure as a component whose inputs and outputs are the call and return arcs, the parameters, and return values. When viewed in this way one can envision that it is possible to extend our compositional model to apply to system components.

7.5 Software Architecture Dependence Analysis

We have considered the extension of this model for application to software architecture descriptions and found it to be inappropriate. This is because the foundation of this model is the use of forward dominance to identify dependencies and that is not a useful notion for application to software architecture description languages in which behaviors are generally expressed in terms of asynchronous event interactions. If a system is modeled in such a language, a control flow graph takes on the form of an event stimulation graph in which there are very few, if any, places where all paths through the graph meet to create a forward dominator. Defining what it means for one part of such a system to depend on another part, determining the sources of those dependencies, and developing methods for identifying reasonably precise sets of dependencies are a challenge to the program dependence analysis community. An additional challenge is the development of formal models from which algorithms can be developed in an orderly fashion [50].

REFERENCES

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[2] F.E. Allen. Program Optimization. In M.I. Halpern and C.J. Shaw, editors, *Annual review in automatic programming*, volume 5, pages 239–307. Pergamon Press Ltd, Oxford, UK, 1969.

[3] F.E. Allen. Control Flow Analysis. *SIGPLAN Notices*, pages 1–19, July 1970.

[4] F.E. Allen. A Basis for Program Optimization. In *Proceedings of the IFIP Congress*, pages 385–390, Amsterdam, 1971. North-Holland.

[5] F.E. Allen. Interprocedural Data Flow Analysis. In *Proceedings of the IFIP Congress*, pages 398–402, Amsterdam, 1974. North-Holland.

[6] F.E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3), March 1976.

[7] S. Baase and A. Van Gelder. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, Massachusetts, third edition, 2000.

[8] W.A. Babich and M. Jazayeri. The Method of Attributes for Data Flow Analysis, Part I: Exhaustive Analysis. *Acta Informatica*, 10(3):265–272, 1978.

[9] W.A. Babich and M. Jazayeri. The Method of Attributes for Data Flow Analysis, Part II: Demand Analysis. *Acta Informatica*, 10(3):265–272, 1978.

[10] S. Bates and S. Horwitz. Incremental Program Testing using Program Dependence Graphs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 384–396, Charleston, S.C., January 1993.

[11] G. Bilardi and K. Pingali. A Framework for Generalized Control Dependence. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 291–300, Philadelphia, Pennsylvania, May 1996. Uses loop dominator forest to efficiently compute CD.

[12] C. Bohm and G. Jacopini. Flow Diagrams, Turing Machines and Languages With Only Tow Formation Rules. *Communications of the ACM*, 9(5), May 1966.

[13] G. Chartrand and L. Lesniak. *Graphs & Digraphs.* Chapman & Hall, London, England, third edition, 1996.

[14] J. Cheng. Slicing Concurrent Programs — A Graph-Theoretical Approach. *Lecture Notes in Computer Science, Vol. 749, Automated and Algorithmic Debugging*, pages 223–240, 1993.

[15] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *Proceedings of 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Bad Herrenalb,Germany, September 1999.

[16] D.E. Denning and P.J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, January 1977.

[17] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, October 1987.

[18] L.D. Fosdick and L.J. Osterweil. Data Flow Analysis in Software Reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[19] H.N. Gabow. Path-Based Depth-first Search for Strong and Biconnected Components. *to appear in Information Processing Letters*, 2000.

[20] S. Graham and M. Wegman. A Fast and Usually Linear Algorithm for Global Flow Analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[21] M.J. Harrold, B. Malloy, and G. Rothermel. Efficient Construction of Program Dependence Graphs. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA '93)*, pages 160–170. ACM SIGSOFT, June 1993.

[22] M.J. Harrold, G. Rothermel, and S. Sinha. Computation of Interprocedural Control Dependence. In *Proceedings of the 1998 ACM International Symposium Software Testing and Analysis (ISSTA '98)*, pages 11–21. ACM SIGSOFT, March 1998.

[23] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Static Analysis Symposium (SAS'99)*, Venice, Italy, September 1999.

[24] M.S. Hecht. *Flow Analysis of Computer Programs.* North-Holland, Amsterdam, 1977.

[25] S. Horwitz, J. Prins, and T. Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Conference Record of the Fifteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–157, San Diego, California, January 1988. Association for Computer Machinery.

[26] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI'88)*, pages 35–46, Atlanta, Georgia, July 1988.

[27] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 22(1):26–60, January 1990.

[28] J.B. Kam and J.D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[29] K. Kennedy. Node Listings Applied to Data Flow Analysis. In *Conference Record of the Second Annual ACM Symposium on Principles of Programming Languages*, pages 11–21, New York, New York, 1975. Association for Computer Machinery.

[30] D.J. Kuck, Y. Muraoka, and S.C. Chen. On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Resulting Speed0up. *ACM Transactions on Computers*, C-21:1293–1310, December 1972.

[31] L. Larsen and M.J. Harrold. Slicing Object-Oriented Software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505. Association for Computer Machinery, March 1996.

[32] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[33] S. Liao, A. Diwan, R.P. Bosch Jr., A. Ghuloum, and M.S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. In *7th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 37–48, Atlanta, Georgia, May 1999.

[34] R. Linger, H.D. Mills, and B.I. Witt. *Structured Programming: Theory and Practice*. The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1979.

[35] D.B. Lomet. Data Flow Analysis in the Presence of Procedure Calls. *IBM Journal of Research and Development*, 21(6), November 1977.

[36] E.S. Lowry and C.W. Medlock. Object Code Optimization. *Communications of the ACM*, 12(1):13–22, January 1969.

[37] J.P. Loyall and S.A. Mathisen. Using Dependence Analysis to Support the Software Maintenance Process. In *Proceedings of the 1993 International Conference on Software Maintenance*, pages 282–291. IEEE Computer Society, September 1993.

[38] S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[39] K.J. Ottenstein and L.M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *ACM SIGPLAN/SIGSOFT Symposium on Practical Programming Development Environments*, pages 177–184, Pittsburgh, Pennsylvania, April 1984. ACM SIGPLAN/SIGSOFT.

[40] K. Pingali and G. Bilardi. Optimal Control Dependence Computation and the Roman Chariots Problem. *ACM Transactions on Programming Languages and Systems*, pages 462–491, May 1997.

[41] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[42] H.A. Podgurski. *The Significance of Program Dependences for Software Testing, Debugging, and Maintenance.* PhD thesis, University of Massachusetts, Amherst, Massachusetts, September 1989.

[43] R.T. Prosser. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *1959 Proceedings of the Eastern Joint Computer Conference*, New York, New York, December 1959. Spartan Books.

[44] D.J. Richardson, T.O. O'Malley, C.T. Moore, and S.L. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119. ACM SIGSOFT, December 1992.

[45] B.K. Rosen. High-Level Data Flow Analysis. *Communications of the ACM*, 20(10):712–724, October 1977.

[46] B.K. Rosen. Data Flow Analysis for Procedural Languages. *Journal of the ACM*, 26(2):322–344, April 1979.

[47] S. Sinha, M.J. Harrold, and G. Rothermel. System-Dependence-Graph-Based Slicing of Programs With Arbitrary Interprocedural Control Flow. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, California, May 1999.

[48] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1997.

[49] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems. Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[50] J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis in Support of Software Maintenance. In *Proceedings of the Third International Software Architecture Workshop*, pages 129–132, November 1998.

[51] J.A. Stafford and A.L. Wolf. Annotating Components to Support Component-Based Static Analyses of Software Systems. In *Proceedings of Grace Hopper Conference 2000 (to appear)*, Hyannis, Massachusetts, September 2000.

[52] V. Vyssotsky and P. Wegner. A Graph Theoretical Fortram Source Language Analyzer. manuscript, AT&T Bell Laboratories, 1963.

[53] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, Ann Arbor, Michigan, 1979.

[54] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[55] J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. In *New Technologies on Computer Software*, pages 135–142, September 1997.

[56] J. Zhao. Multithreaded Dependence Graphs for Concurrent Java Programs. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*, Los Angeles, California, May 1999. IEEE Computer Society.

[57] J. Zhao, J. Cheng, and K. Ushijima. Static Slicing of Concurrent Object-Oriented Programs. In *Twentieth Annual International Computer Software and Applications Conference*, pages 312–320, Seoul, Korea, August 1996. IEEE Computer Society.

Appendix A

Notation

The following is a list of abbreviations and notations used in this thesis.

CCDG	Compound Control Dependence Graph
CDG	Control Dependence Graph
CFDF	Compound Forward Dominance Forest
CFG	Control Flow Graph
$y \xrightarrow{dcd} x$	y is directly interprocedural control dependent on x
$y \xrightarrow{drcd} x$	y is directly resumption dependent on x
$y \xrightarrow{dicd} x$	y is directly invocation dependent on x
$y \xrightarrow{dpcd} x$	y is directly potential dependent on x
$y \xrightarrow{cd} x$	y is interprocedural control dependent on x
$y \xrightarrow{icd} x$	y is invocation dependent on x
$y \xrightarrow{pcd} x$	y is potential dependent on x
$y \xrightarrow{rcd} x$	y is resumption dependent on x
FDF	Forward Dominance Forest
fdom(X)	Forward Dominator of X
FDT	Forward Dominance Tree
ifdom(X)	Immediate Forward Dominator of X
indfdom(X)	Indirect Forward Dominator of X
G^{-1}	Inverse of graph G (i.e., all arcs in G are reversed)
PCDG	Procedure Control Dependence Graph
PCFG	Procedure Control Flow Graph
PFDF	Procedure Forward Dominance Forest
pfdom arc	Procedure forward dominance arc
RPFDF	Resolved Procedure Forward Dominance Forest

Appendix B

Dictionary of Related Terminology

A

acyclic A graph is said to be acyclic if it contains no cycles.

adjacent from [42][1] If (u,v) is an arc, v is adjacent from u.

adjacent to [42] If (u,v) is an arc u is adjacent to v.

annotated arc An annotated arc is an arc that contains zero or more predefined attributes/value pairs.

annotated call graph See call graph.

annotated vertex An annotated vertex is a vertex that contains zero or more predefined attributes/value pairs.

arc An arc (u,v) is a pair of vertices, u and v, in a digraph.

B

backarc An arc (u,v) in digraph G is a backarc if there is a closed walk $W = v_1 v_2 \ldots v_n v_1$ in G such that $u = v_n$ and $v = v_1$.

basic block A basic block is a sequence of statements $s_1, s_2, \ldots, s_k$ such that for $1 \leq i < k$, s_i is the only predecessor of s_{i+1}, and s_1 does not have a unique predecessor.

C

call arc See interprocedural control flow graph.

[1]When a definition contains a citation, the definition has been taken directly (possibly with minor syntactic changes) from the cited work.

call graph A call graph for a given program $\mathcal{P}$ is a digraph $\mathcal{CG} = \{V, A\}$ where V is a set of vertices, one for each procedure in the program; and A is a set of arcs (P_1,P_2) where the procedure represented by P_1 contains one or more calls to the procedure represented by P_2. The arcs in the call graph may be annotated with call site identifiers for its associated calls if that information is deemed useful to analysis for which the call graph is to be constructed. We call this an annotated call graph.

call vertex See interrupted-flow arc.

chain A chain is a sequence of elements $S = s_1, s_2, \ldots, s_n$ of some set $\mathcal{S}$, such that for a given relation R on $\mathcal{S}$, for each $s_i \in S$, $1 \leq i \leq n-2$, $s_i R s_{i+1} \Longrightarrow s_{i+1} R s_{i+2}$.

closed walk [42] A closed walk is a walk $v_1 v_2 \ldots v_n$ such that $n > 1$ and $v_1 = v_n$.

composed control dependence graph A composed control dependence graph (CCDG) $\mathcal{G} = (V, A, v^S, v^F)$ for a program $\mathcal{P} = \{P_1, \ldots, P_n\}$ is a graph that has one vertex for each vertex in the procedure control flow graph associated with each $P_i \in \mathcal{P}$. $A = CD \cup IC$ is a set of arcs, where an arc in CD represents the direct control dependence of its target on its source and arcs in IC represent inherited control dependencies. v^S and v^F represent the start and the finish of $\mathcal{P}$ respectively.

control dependence (non-inlined) Given program $\mathcal{P}$, a statement $y \in \mathcal{P}$ is directly control dependent on another statement $x \in \mathcal{P}$ if and only if one of the following conditions holds:

- x and y are statements in the same procedure and y is directly control dependent on x
- y is directly resumption control dependent on x
- y is directly invocation control dependent on x
- y is potential control dependent on $z \in \mathcal{P}$, y is not resumption control dependent on any statement in $\mathcal{P}$, and z is directly control dependent x.

and there exists a path from x to y that does not contain any other statement upon which y is control dependent. y is control dependent on x if and only if there exists a chain of direct interprocedural control dependencies from x to y. Control dependence is denoted $y \xrightarrow{cd} x$ and direct control dependence is denoted $y \xrightarrow{dcd} x$.

computation sequence [42] Let $\mathcal{G} = (g, \sigma, D, U)$ be a def/use graph, let $\mathcal{I} = (\mathcal{D}, \mathcal{F}, \mathcal{N})$ be an interpretation of $\mathcal{G}$, and let $d \in \mathcal{D}^\sigma$. The computation sequence of the program ($\mathcal{P} = (\mathcal{G}, \mathcal{I})$ on the input d is the (finite or infinite) sequence $\mathcal{C}_\mathcal{P}(d)$, defined inductively as follows:

1. $\mathcal{C}_\mathcal{P}(d)(1) = (v_\mathcal{I}, d)$
2. $\mathcal{C}_\mathcal{P}(d)(i+1)$ is defined if and only if each of the following is true:
   (a) $\mathcal{C}_\mathcal{P}(d)(i) = (v_i, val_i)$ is defined

(b) $v_i \neq v_F$

(c) $F(v_i)(val_i \mid U(v_i)) \downarrow$

(d) $v_i \in V_{dec}(G)$ implies $N(v_i)(val_i \mid I(v_i)) \downarrow$ If $\mathcal{C}_{\mathcal{P}}(d)(i{+}1)$ is defined, then $\mathcal{C}_{\mathcal{P}}(d)(i{+}1) = (v_{i+1}, val_{i+1})$, where

$$v_{i+1} = \begin{cases} N(v_i)(val_i \mid U(v_i)) & \text{if } v_i \in V_{dec}(G) \\ succ(v_i) & \text{if } v_i \in V(G) - V_{dec}(G) \end{cases}$$

concatenation [42] If $W = w_1w_2\ldots w_m$ and $X = x_1x_2\ldots x_n$ are walks such that either W is empty, X is empty, or w_m is adjacent to x_1, then the concatenation of walk W and walk X, denoted WX, is the walk $w_1w_2\ldots w_mx_1x_2\ldots x_n$.

connected graph [42] A graph G is connected if and only if for each pair u, v of vertices in G, there is a $u - v$ walk.

contracted graph [19] A contracted call graph is the graph that results after identifying and collapsing strong components of a graph.

control dependence (inlined) [42] Let G be a control flow graph, and let u,$v \in V_G$. Vertex u is strongly control dependent (or sc-dependent) on vertex v if and only if there exists a $v - u$ walk vWu in G not containing the immediate forward dominator of v; this walk is said to demonstrate that u is strongly control dependent on v.

Let G be a control flow graph, and let $u, v \in V_G$. Vertex u is directly strongly control dependent (or dsc-dependent) on v if and only if there is a walk vWu in G such that both of the following are true:

- vWu demonstrates that u is strongly control dependent on v
- u is not strongly control dependent on any vertex of W.

The walk vWu is said to demonstrate that u is directly strongly control dependent on v.

Let G be a control flow graph, and let $u, v \in V_G$. Then u is directly weakly control dependent (or, dwc-dependent) on v if and only if v has successors v' and v'' such that u strongly forward dominates v' but does not strongly forward dominate v''; u is weakly control dependent (or, wc-dependent) on v if and only if there exists a sequence, $v_1, v_2, ..., v_n$, of vertices, where $n \geq 2$, such that $u = v_1$, $v = v_n$, and v_i is directly weakly control dependent on v_{i+1} for $i = 1, 2, \ldots, n-1$.

control dependence graph A composed control dependence graph in which paths including inherited dependencies have been resolved into direct control dependencies is called a control dependence graph.

control flow graph [42] A control-flow graph G is a digraph that satisfies each of the following conditions:

- The out-degree of any vertex of G is at most two.
- G contains two distinguished vertices: the initial vertex v^I, which has in-degree zero, and the final vertex v^F, which has out-degree zero.
- Every vertex of G occurs on some $v^I - v^F$ walk.

cycle [42] A cycle C in a digraph G is a closed walk $v_1v_2 \ldots v_nv_1$ such that v_1, v_2, ... , v_n are distinct.

cycle [42] A cycle C in a graph G is a walk $v_1v_2 \ldots v_n$ such that $n > 3$ and $v_1, v_2, \ldots, v_n$ are distinct.

D

def/use graph A def/use graph is an annotated control flow graph G in which a $v \in V_G$ may contain two sets U and D. Each $u \in U_s$ is the name of a variable used at statement s. Each $d \in D_s$ is the name of a variable defined at statement s.

digraph [42] A directed graph or digraph G is a pair (V_G,A_G), where V_G is any finite set and A_G is a subset of $V_G \times V_G - \{(v,v)|v \in V_G\}$. The elements of V_G are called vertices and the elements of A_G are called arcs.

disjoint [42] Two digraphs G_1 and G_2 are disjoint if and only if $V_{G_1} \cup V_{G_2}$ is empty.

E

edge An edge (u,v) is a pair of vertices, u and v, in a graph.

empty walk An empty walk is a walk of length 0.

execution history [42] Let $\mathcal{P} = (\mathcal{G}, \mathcal{I})$ be a program, where $\mathcal{G} = (G, \sigma, D, U)$ and $\mathcal{I} = (\mathcal{D}, F, N)$, and let $v \in V(G)$, $d \in \mathcal{D}^\sigma$, and computation sequence $\mathcal{C}_P(d) = \{(v_i, vali)\}$. The execution history of v induced by the input d is the (finite or infinite, possibly empty) sequence $\mathcal{H}_P(v,d)$ defined as follows:

1. For $j \geq 1$, $\mathcal{H}_P(v,d)(j)$ is defined if and only if there is a jth smallest value $i(j)$ of i for which $\mathcal{C}_P(d)(i) = (v_i, val_i)$ is defined with $v_i = v$.
2. For $j \geq 1$, if $\mathcal{H}_P(v,d)(j)$ is defined, then $\mathcal{H}_P(v,d)(j) = val_{i(j)} \mid U(v)$.

execution trace An execution trace of a program $\mathcal{P}$ is a sequence of strings $T_{\mathcal{P}(I)} = t_1, t_2, \ldots, t_n$ where each $t_i \in T_{\mathcal{P}(I)}$ represents the execution of a statement in $\mathcal{P}$ based on a run of $\mathcal{P}$ given input I plus infeasible traces that are undectable based on program syntax alone.

F

forward dominance forest Given a digraph G and $V_A = \{v_{a_1}, \ldots, v_{a_n}\}$ is a set containing exactly one element for each childless vertex in G, a forward dominance forest $F = \{t_1, \ldots, t_n\}$ is a set of forward dominance trees where, for each $t_i = \{r_i, V_i, A_i\} \in F$, $V_i = \{v_{i_1}, \ldots, v_{i_n}\}$ represents vertices in G, $r_i = v$ for some $v \in V_A$, and $A_i = \{(\mathit{ifdom}(u), u) \mid u \in V_i - v_A\}$.

forward dominance tree Given a control flow graph G, a forward dominance tree $T = \{r, V, A\}$, where $V = \{v_1, \ldots, v_n, v^F\}$ represents the vertices in G, $r = v^F$ is the root of the tree and $A = \{(\mathit{ifdom}(u), u)$ where $u \in V - v^F\}$.

forward dominate [42] (also referred to as post dominate and inverse dominate) Let G be a control flow graph. A vertex u in V_G forward dominates a vertex v in V_G if and only if every $v - v^F$ walk contains u; u properly forward dominates v if and only if $u \neq v$ and u forward dominates v. The immediate forward dominator of a vertex $v \in V_G$, $v \neq v^F$ is the vertex that is the first proper forward dominator of v to occur on every $v - v^F$ walk; we denote the immediate forward dominator of v by $\mathit{ifdom}(v)$. A vertex $u \in V_G$ strongly forward dominates $v \in V_G$ if and only if u forward dominates v and there is an integer $k \geq 1$ such that every walk in G beginning with v and of length $\geq k$ contains u.

G

graph [42] A graph G is a pair (V_G,A_G), where V_G is any finite set and A_G is a set of unordered pairs of elements of V_G. The elements of V_G are called vertices and the elements of A_G are called edges.

H

I

ifa-source See interrupted-flow arc.

ifa-target See interrupted-flow arc.

immediate forward dominator [42] See forward dominate.

incident from [42] An arc (u,v) is incident from u.

incident to [42] An arc (u,v) is incident to v.

in-degree [42] The in-degree of a vertex v is the number of vertices adjacent to v.

indirectly forward dominates Given a procedure control flow graph G and $(u, v) \in I_G$, if $v \in \mathit{fdom}(u)$ then v indirectly forward dominates u. This relationship is denoted $\mathit{indfdom}(u) = v$.

induced [42] If U is a nonempty subset of the vertex set V_G of a digraph G, then the subdigraph$< U >$ induced by U is the digraph having vertex set U and arc set A_G intersected with $(U \times U)$.

interprocedural control dependence [37] Let $\mathcal{G}$ be an interprocedural CFG and let G_i and G_j be CFGs in $\mathcal{G}$. Let $u \in N_{G_i}$, and $v \in N_{G_j}$. Vertex u is directly strongly control dependent on v if and only if v has successors v' and v'' such that u forward dominates v' but does not forward dominate v''. Vertex u is strongly control dependent on v if and only if there exists a v-u interprocedural walk not containing the immediate forward dominator of v.

interprocedural control flow graph Given a program $\mathcal{P} = \{P_1, \ldots, P_n\}$, an interprocedural control flow graph is a digraph $\mathcal{G} = \{G_1, \ldots, G_n, v^S, v^F, C, R, J\}$ where

- there exists exactly one procedure control flow graph $G_i \in \mathcal{G}$ for each procedure $P_i \in \mathcal{P}$;
- v^S is a distinguished *start vertex* that has zero in-degree and represents the start of $\mathcal{P}$;
- v^F is a distinguished *finish vertex* that has zero out-degree and represents the stop of $\mathcal{P}$;
- C is a set of *call arcs* (u,v) where u is a call vertex in V_{G_i} and $v = v^I_{G_j}$ is the initial vertex for some procedure control flow graph G_j; $G_i, G_j \in \mathcal{G}$;
- R is a set of *return arcs* (u,v) where $u = v^R_{G_j}$ is the return vertex in procedure control flow graph G_j and $v \in V_{G_i}$ is a resumption vertex in V_{G_i}; $G_i, G_j \in \mathcal{G}$

that satisfies each of the following conditions:

- There is a one-to-one correspondence between C and R. For each $c_i = (u, v^I_{G_j}) \in C$ there exists exactly one $r_i = (v^R_{G_j}, v) \in R$, and there exists ($u$,$v$) $\in I_{G_i}$.
- Every vertex of $V_{\mathcal{G}}$ occurs on some $v^I - v^F$ path.

interprocedural walk An interprocedural walk $\mathcal{W}$ in an interprocedural control flow graph $\mathcal{G}$ is a digraph walk that satisfies the following conditions:

- For any two vertices v_i and $v_{i+1} \in \mathcal{W}$ there does not exist $[v_i, v_{i+1}]$ in $\mathcal{G}$.
- Given [c,r] $\in I_{G_i}$, for some $G_i \in \mathcal{G}$ and $c\mathcal{Y}r$ is a subsequence of vertices in $\mathcal{W}$, then if for any vertex v in $\mathcal{Y}$ there exists an interrupted-flow arc [u,v] in $\mathcal{G}$ then $u\mathcal{S}v$ is a subsequence in $\mathcal{Y}$ for some sequence of vertices $\mathcal{S}$.

interrupted-flow arc Given a procedure P, and statements $s, t \in P$, where u is a call statement, an interrupted-flow arc (u,v) is an arc incident from a graph vertex u representing s and incident to vertex v representing t. An interrupted-flow arc represents the expectation that, in the absence of abnormal termination, program execution will eventually resume at t from after a call at s. u is called an ifa-source or call vertex and v is called an ifa-target or resumption vertex.

intraprocedural control dependence Given a procedure control flow graph G and its associated procedure forward dominance forest F, a vertex v in G is intra-procedurally control dependent on another vertex u in G if and only if there is a $u - v$ intraprocedural walk $\mathcal{W}$ that does not include the immediate forward dominator of u. v is directly intraprocedurally control dependent on u if v is the first vertex in $\mathcal{W}$ that is intraprocedurally control dependent on u.

intraprocedural walk An intraprocedural walk W in a procedure control flow graph G is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $n \geq 0$ and $(v_i, v_{i+1}) \in A_G$ or $(v_i, v_{i+1}) \in I_G$ for $1 \leq i < n$ and is denoted p-walk. The *length* of a p-walk $W = v_1, v_2, \ldots, v_n$, denoted $|W|$, is the number n of vertex occurrences in W. Note that a p-walk of length zero has no vertex occurrences; such a p-walk is called *empty*.

invocation control dependence Given program $\mathcal{P}$ and two procedures $P, Q \in \mathcal{P}$ and statements x, y, and z in $\mathcal{P}$ such that:

- $x \in P$ is a call statement to Q,
- either $y \in P_i$ for some $P_i \in \mathcal{P}$ and x is directly control dependent on y or
- or y is $start^{\mathcal{P}}$ and P is the main procedure of $\mathcal{P}$,
- z is the entry to Q or a statement in Q that is not otherwise control dependent on any vertex in $\mathcal{P}$;

then z is directly invocation control dependent on y. This relationship is denoted $z \xrightarrow{dicd} y$. y is invocation control dependent on x if and only if there is a chain of direct control dependencies $s_1, \ldots, s_n$ such that $s_1 = y$, $s_n = x$, and $s_{i-1} \xrightarrow{dicd} s_i$, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{icd} x$.

J

K

L

length [42] The length of a walk $W = v_1 v_2 \ldots v_n$, denoted $|W|$, is the number n of vertex occurrences in W. A walk of length 0 is has no vertex occurrences.

M

N

O

out-degree [42] The out-degree of a vertex v is the number of vertices adjacent from v.

path [42] A path is a walk in which no vertex occurs more than once. This is the definition of simple path given by Lengauer and Tarjan in [2]. Their definition of path is equivalent to a walk.

post dominate see forward dominate.

potential forward dominance Given a procedure control flow graph G and $(u,v) \in I_G$, then v potentially forward dominates u. This relationship is denoted
$pfdom(u) = v$.

Potential control dependence Given program $\mathcal{P}$ and a procedure $P \in_{\mathcal{P}}$, statement x and a sequence of statements $Y = y_1, y_2, \ldots, y_n$ such that:

- $x \in P$ is a call statement,
- y_1 is the first statement to be executed if control returns to P from the call at x, and
- for every $y_i \in Y$, y_i is not a conditional statement or a call statement;

we say that each $y_i \in Y$ is directly potential control dependent on x. This relationship is denoted $y_i \xrightarrow{dpcd} x$. y is potential control dependent on x if and only if there is a chain of direct potential control dependencies $s_1, \ldots, s_n$ such that $s_1 = y$, $s_n = x$, and s_{i-1} $dpcd$ s_i, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{pcd} x$.

predecessor [42] A predecessor of a vertex v is a vertex adjacent to v.

PRED [42] $\mathrm{pred}_G(v)$ is the set of predecessors of v in digraph G (the subscript is omitted when the digraph in question is clear from the context).

prefix [42] If $W = XY$ is a concatenation of walks X and Y then X is a prefix of W.

procedure control dependence graph (PCDG) A procedure control dependence graph $G = (V, A)$ for a procedure P is a graph that has one vertex for each vertex in the procedure control flow graph that represents P. $A = CD \cup PC \cup IC$ is a set of arcs, where an arc in CD represents the direct control dependence of its target on its source, each arc in PC represents the potential for its target to inherit the dependence of its source, and an arc in IC represents the fact that its target inherits the control dependence of its source.

procedure control flow graph (PCFG) Given a procedure $P = \{s_1, \ldots, s_n\}$, a procedure control flow graph $G = (V, A)$ is a digraph where $V = \{v_1, \ldots, v_n, v^I, v^R\}$, $A = (F_G, I_G)$. V contains two distinguished vertices: an initial vertex v^I associated with invocation of the procedure, and an annotated return vertex v^R. F_G is a set of control flow arcs and I_G is a set of interrupted-flow arcs. For every vertex v with zero in-degree there exists a control flow arc (v^I, v) and for every vertex v_i that represents a procedure return, there exists a control flow arc $(v_i, v^R) \in F_G$. G satisfies each of the following conditions:

- in G there exists exactly one $v \in V$ to represent each $s \in P$.

- For every call statement in P there exists exactly one arc in I_G.

procedure forward dominance forest (PFDF) Given a procedure control flow graph G, a procedure forward dominance forest $F = \{R, T, V, A, D\}$ where $V = \{v_1, \ldots, vn\}$ is the set of vertices in G that have successors, $R = \{r_1, \ldots, r_n\}$ is the set of childless vertices in G, and for every $v \in V$ if v is an ifa-source, then there exist $d_i = (\mathit{pfdom}(v), v) \in D$, otherwise there exists an $a_i = (\mathit{ifdom}(v), v)$ $\in A$.

proper forward dominator [42] See forward dominate.

Q

R

relation A relation R from a set A to a set B is any subset of $A \times B$. The set A is called the domain of R, and the set B the range of R. We denote $(a,b) \in R$ by aRb.

resolved procedure forward dominance forest (RPFDF) Given a PFDF F, a resolved procedure forward dominance forest $R = F - D + I$ where $I = \{a_1, \ldots,$ $a_n\}$ where $a_i = (\mathit{indfdom}(v), v) \in A$.

resumption control dependence Given program $\mathcal{P}$ and two procedures $P_1, P_2 \in \mathcal{P}$, statements w, x, and y such that:

- $w \in P_1$ is a call statement to P_2,
- $x \in P_2$,
- $y \in P_1$ and $y \xrightarrow{dpcd} w$;

if any return statement in P_2 is control dependent on x, y is directly resumption control dependent on x. This relationship is denoted $y \xrightarrow{drcd} x$. y is resumption control dependent on x if and only if there is a chain of direct control dependencies $s_1, \ldots, s_n$ such that $s_n \xrightarrow{drcd} s_{n-1}$ where $s_1 = y$, $s_n = x$, and $s_{i-1} \xrightarrow{drcd} s_i$, $1 < i \leq n$. This relationship is denoted $y \xrightarrow{rcd} x$.

resumption vertex See interrupted-flow arc.

return arc See interprocedural control flow graph.

S

semantic dependence [42] Let $\mathcal{G} = (G, \sigma, D, U)$ be a def/use graph, and let $u, v \in V(\mathcal{G})$. Vertex u is semantically dependent on vertex v if and only if there exist interpretations $I_1 = (\mathcal{D}, F_1, N_1)$ and $(\mathcal{D}, F_2, N_2)$ of $\mathcal{G}$ and an input $d \in \mathcal{D}^\sigma$ such that, letting $P_1 = (G, I_1)$ and $P_2 = (G, I_2)$:

1. for all $w \in V(G) - \{v\}$, $F_1(w) = F_2(w)$ and if $w \in V_{dec}(G)$ then $N_1(w) = N_2(w)$; and
2. Either:
   (a) There is some $i \geq 1$ such that execution histories $\mathcal{H}_{P_1}(u, d)(i)$ and $\mathcal{H}_{P_2}(u, d)(i)$ are both defined but are unequal; or
   (b) $\mathcal{H}_{P_1}(u, d)$ is longer than $\mathcal{H}_{P_2}$, and either computation sequence $\mathcal{C}_{P_2}(d)$ $= \{(v_i, val_i)\}$ is infinite or there is some v_i from which u is unreachable

I_1, I_2, and d are said to demonstrate that u is semantically dependent on v. If condition 1 holds and either condition 2(a) holds or both of the following conditions are true:

1. $\mathcal{H}_{P_1}(u, d)$ is longer than $\mathcal{H}_{P_2}(u, d)$; and
2. u is unreachable from some vertex of $\mathcal{C}_{P_2}(d)$

then I_1, I_2, and d are said to finitely demonstrate that u is semantically dependent on v.

set A set is collection of objects in which each object appears exactly once; the order of objects is undefined.

source If a =(u,v) is an arc, u is the source of a.

strong control dependence See control dependence.

strongly forward dominate [42] See forward dominate.

subdigraph [42] A digraph G' is a subdigraph of a digraph G if and only if $V_{G'}$ is a subset of V_G and $A_{G'}$ is a subset of A_G.

subset Given two sets A and B, A is a subset of B if and only if every object of A is also an object of B. This relationship is denoted $A \subset B$.

successor [42] A successor of vertex v is a vertex adjacent from v.

SUCC [42] $\text{succ}_G(v)$ is the set of successors of v in digraph G (the subscript is omitted when the digraph in question is clear from the context).

suffix [42] If $W = XY$ is a concatenation of walks X and Y then Y is a suffix of W.

T

target If a =(u,v) is an arc, v is the target of a.

transitive Given a set A, a relation R on the elements of the set is transitive if for every a, b, c in A, if aRb and bRc then aRc.

tree [42] A tree is an acyclic, connected graph.

U

underlying graph [42] The underlying graph of a digraph G is the graph whose vertex set is V_G and whose arc set is $\{(u, v) \mid (u, v) \in A(G)\}$.

V

vertex A vertex v is an element of a graph or a digraph.

W

walk [42] A walk W in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that $n > 1$, $u = v_1$ $v = v_n$, and (v_i,v_{i+1}) in E_G for $i = 1, 2, \ldots, n-1$. A nonempty walk whose first element is u and whose last element is v is called a $u - v$ walk.

walk [42] A walk W in a digraph G is a sequence of vertices $v_1, v_2 \ldots, v_n$ such that $n > 0$, and (v_i,v_{i+1}) in A_G for $i = 1, 2, \ldots, n-1$. A nonempty walk whose first element is u and whose last element is v is called a $u - v$ walk.

weak control dependence See control dependence.

weakly connected [42] A digraph is weakly connected if and only if its underlying graph is connected.

X

Y

Z