ACM Transactions on Software Engineering and Methodology (to appear).

Interprocedural Control Dependence

Saurabh Sinha and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
801 Atlantic Avenue
Atlanta, GA 30332 USA
{sinha,harrold}@cc.gatech.edu

Gregg Rothermel
Computer Science
Oregon State University
Dearborn Hall 307-A
Corvallis, OR 97331 USA
grother@cs.orst.edu

Abstract

Program-dependence information is useful for a variety of applications, such as software testing and maintenance tasks and code optimization. Properly defined, control and data dependences can be used to identify semantic dependences. To function effectively on whole programs, tools that utilize dependence information require information about interprocedural dependences: dependences that are identified by analyzing the interactions among procedures. Many techniques for computing interprocedural data dependences exist; however, virtually no attention has been paid to interprocedural control dependence. Analysis techniques that fail to account for interprocedural control dependences can suffer unnecessary imprecision and loss of safety. This paper presents a definition of interprocedural control dependence that supports the relationship of control and data dependence to semantic dependence. The paper presents two approaches for computing interprocedural control dependences, and empirical results pertaining to the use of those approaches.

Keywords: Interprocedural control dependence, interprocedural analysis, semantic dependence.

1 Introduction

Program-dependence information is useful for a variety of applications, such as software testing and maintenance tasks and code optimization. Such information can be used, for example, to locate the cause of a software failure, to evaluate the impact of a modification, to determine the parts of a program that should be re-tested in response to a modification, or to identify parts of the code to which optimizing transformations can be applied. For such purposes, program dependences provide approximate but useful information [23]. Control-dependence information captures the effects of predicate statements on program behavior. Data-dependence information captures the effects of data interactions on program behavior. Tools such as program slicers use control- and data-dependence information for tasks such as debugging, impact analysis, and regression testing.

Much research (e.g., [3, 6, 8, 21, 24, 27]) has addressed the problems of computing and utilizing *intraprocedural dependences*: dependences within procedures that can be computed by analyzing procedures independently. That research has considered both control and data dependence.

To function effectively on whole programs, however, techniques that require dependence information must account for *interprocedural dependences*: dependences that can be computed only by analyzing the interactions among procedures. Various definitions of, and methods for computing and utilizing, interprocedural data dependences have been presented, and the necessity of considering these dependences in interprocedural analyses is well understood (e.g. [5, 13, 16, 20, 25, 28]). In contrast, virtually no attention has been paid to the definition or computation of interprocedural control dependence. Our search of the research literature reveals only one attempt to define and compute interprocedural control dependence [17]; however, as we show in Section 6, that definition and approach can omit dependences. Furthermore, we have found no interprocedural analysis techniques that explicitly consider the effects of interprocedural control dependences.

Our empirical studies indicate that the failure to account for interprocedural control dependences may significantly affect analysis results. When analysis techniques that utilize dependence information are applied to programs without accounting for interprocedural control dependences, the techniques can identify dependences that do not exist, which can lead to excessively large solutions to analysis problems; the techniques can also ignore dependences that do exist, which can lead to errors of omission in solutions to analysis problems. For some analyses, such as slicing for reverse engineering, errors of omission may be acceptable [19]; for other analyses, such as slicing for program integration, errors of omission are not allowable [14].

This paper addresses the issues surrounding interprocedural control dependences, and their potential effects on interprocedural analysis techniques. The main contributions of the paper are:

- A description of several ways in which control dependences computed intraprocedurally inaccurately model the control dependences that exist in whole programs.
- A precise definition of interprocedural control dependence. Unlike the previously presented definition [17], this definition supports the relationship between syntactic and semantic dependence [23] that must hold if analyses based on dependence information are to conservatively model the semantic dependences in programs.
- Two approaches for computing interprocedural control dependences: One approach computes precise interprocedural control dependences but may be inordinately expensive; the other approach summarizes control dependences, and efficiently obtains a conservative (safe) estimate of those dependences at the cost of some precision. The paper provides empirical results pertaining to the effectiveness and efficiency of these approaches.

The remainder of this paper is organized as follows. The next section provides background information necessary to support our definition of interprocedural control dependence. Section 3 demonstrates several effects related to interprocedural control dependence, and then provides our definition of interprocedural control dependence. Section 4 presents our algorithms for calculating interprocedural control dependence, and Section 5 presents empirical results obtained in the use of the second algorithm. Section 6 reviews related work, and illustrates the drawbacks of the existing definition of interprocedural control dependence. Finally, Section 7 presents conclusions and outlines possible future work.

2 Background

To demonstrate the semantic basis for uses of program dependences, and to evaluate some of those uses, Podgurski and Clarke [23] present a formal model of program dependences. They distinguish several types of control and data dependences, and describe conditions under which identification of such (syntactic) dependences may or may not imply identification of semantic dependences (cases where the behavior of a statement can indeed affect the execution behavior of another statement). Their results show that a maintenance tool such as a slicer, that uses control and data dependences to identify a superset of the statements that could semantically affect another statement, can omit semantic dependences if it utilizes inappropriate definitions or computations of data or control-dependence information.

Our definition of interprocedural control dependence builds on this previous work. This section presents definitions drawn directly from or based on those given in Reference [23] that are prerequisite to that definition.

Control dependences are typically defined in terms of control-flow graphs, paths in those graphs, and the postdominance relation.

> **Definition 1.** A *control-flow graph* (CFG) $G = (N, E)$ for procedure P is a directed graph in which N contains one node for each statement in P, and E contains edges that represent possible flow of control between statements in P. N contains two distinguished nodes, n_e and n_x, representing entry to, and exit from P, respectively, where n_e has no predecessors, and n_x has no successors. If P contains multiple exit points, E contains an edge from each node that represents an exit point to n_x. Each call site in P is represented by a call node and a return node in G, and there is an edge from each call node to its associated return node. Each node in N is reachable from n_e, and n_x is reachable from each node in N. Each node in N that represents a predicate statement is called a *predicate node* and has exactly two successors; all other nodes in N except n_x have exactly one successor.

> **Definition 2.** An n_1–n_k *path* in a CFG $G = (N, E)$ is a sequence of nodes $W = n_1 n_2 \ldots n_k$ such that $k \geq 0$, and such that, if $k \geq 2$, then for $i = 1, 2, \ldots, k-1$, $(n_i, n_{i+1}) \in E$.[1]

> **Definition 3.** Let $G = (N, E)$ be a CFG. A node $u \in N$ *postdominates* a node $v \in N$ if and only if every v–n_x path in G contains u.

Several forms of control dependence have been identified in the research literature. We restrict our attention to the form of control dependence found most commonly in the literature, described as "control dependence" [8], as "direct, strong control dependence" [23], and as "classical control dependence" [3].

> **Definition 4.** Let $G = (N, E)$ be a CFG, and let $u, v \in N$. Node u is *control dependent* on node v if and only if v has successors v' and v'' such that u postdominates v' but u does not postdominate v''.

[1] Podgurski and Clarke [23] use the term "walk" to refer to a sequence of adjacent nodes in a graph. We use the term "path" to refer to such a sequence because it is more standard in the literature.

For control-dependence computation, a CFG G is augmented with a unique predicate node, n_s, and edges (n_s,n_e), labeled 'true', and (n_s,n_x), labeled 'false' [8]. By this mechanism, nodes in G that are not control dependent on any predicate nodes are control dependent on entry to the procedure.

The following definitions extend the CFG to model data elements, and use this extension to define data dependence.

Definition 5. A *def/use graph* is a quadruple $G^{du} = (G, \Sigma, D, U)$, where $G = (N, E)$ is a CFG, Σ is a finite set of symbols called variables, and $D : N \rightarrow \mathbf{P}(\Sigma)$, $U : N \rightarrow \mathbf{P}(\Sigma)$ are functions.

Definition 6. Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$, and let $u, v \in N$. Node u is *data dependent* on node v if and only if there exists a path vWu in G^{du} such that $(D(v) \cap U(u)) - D(W) \neq \phi$, where $D(W) = \cup_{n_i \in W}(n_i \notin \{u, v\})D(n_i)$.

The next definition captures the notion that two nodes in a def/use graph may be connected by a chain of data and control dependences, resulting in a syntactic dependence.

Definition 7. Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$, and let $u, v \in N$. Node u is *syntactically dependent* on node v if and only if there is a sequence $n_1, n_2, \ldots, n_k$ of nodes, $k \geq 2$, such that $v = n_1$, $u = n_k$, and for $i = 1, 2, \ldots, k-1$ either n_{i+1} is control dependent on n_i or n_{i+1} is data dependent on n_i.[2]

Podgurski and Clarke [23] define semantic dependence, and relate it to syntactic dependence. Informally, when the semantics of statement s may affect the execution of statement s', s' is semantically dependent on s. A more formal definition is based on notions of interpretations, computation sequences, and execution histories, defined as follows [23].

Definition 8. Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$. An *interpretation* of G^{du} is an assignment of partial computable functions to the vertices of G^{du}. The function assigned to a vertex $v \in N$ is the function computed by the program statement that v represents; it maps values for the variables in $U(v)$ to values for the variables in $D(v)$ or, if v is a decision vertex, to a successor of v.

Definition 9. A *computation sequence* of a program is the sequence of states (pairs consisting of a statement and a function assigning values to all the variables in the program) induced by executing the program with a particular input.

Definition 10. Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$. An *execution history* of a vertex $v \in N$ is the sequence whose ith element is the assignment of values held by the variables of $U(v)$ just before the ith time v is visited during a computation.

Given these definitions, Podgurski and Clarke [23] define semantic dependence as follows:

Definition 11. A node u in a def/use graph G^{du} is *semantically dependent* on a node v in G^{du} if there are interpretations I_1 and I_2 of G^{du} that differ only in the function assigned to v, such that for some input, the execution history of u induced by I_1 differs from that induced by I_2.

[2] Podgurski and Clarke [23] define two types of syntactic dependence: *weak* and *strong*. We restrict our attention to the latter, and refer to it simply as syntactic dependence.

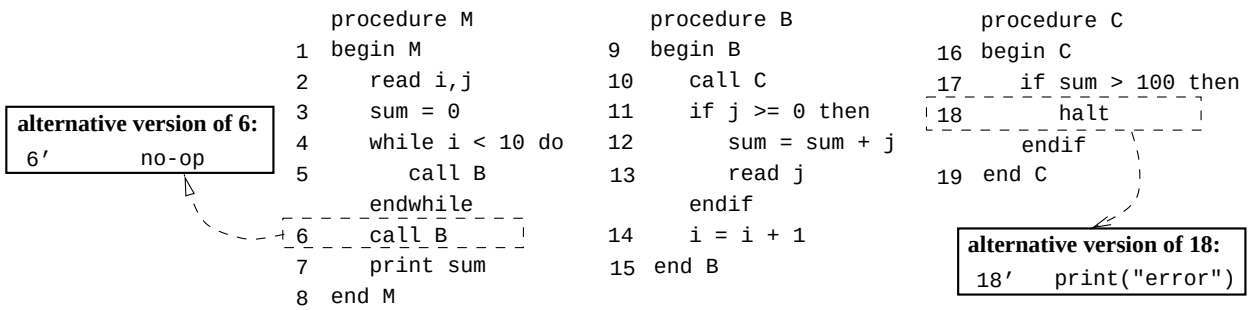

Figure 1: Program `Sum` with alternative versions of two of its statements.

Semantic dependence of u on v can be demonstrated in either of two ways: (1) if for some pair of interpretations, the execution histories of u differ in some pair of corresponding entries, or (2) if for some pair of interpretations, the execution histories of u have different lengths. When case (1) holds, or when case (2) holds with respect to finite portions of computations, the semantic dependence is said to be *finitely demonstrated*: this necessarily occurs when programs halt, but can also occur for non-halting programs.

There is no algorithm to determine, for arbitrary statements s and s', whether s' is semantically dependent on s; however, Podgurski and Clarke [23] demonstrate that given appropriate definitions of control and data dependence, there exist useful relationships between syntactic and semantic dependence. In this paper, we restrict our attention to the relationship stated in the following theorem:[3]

> **Theorem 1.** Let $G^{du} = (G, \Sigma, D, U)$ be a def/use graph with $G = (N, E)$, and let $u, v \in N$. If u is semantically dependent on v and this semantic dependence is finitely demonstrated, then u is syntactically dependent on v.[4]

Theorem 1 is significant because it shows that, given appropriate definitions of control and data dependence, syntactic dependence is a necessary condition for (finitely demonstrated) semantic dependence. Thus, the theorem provides justification for algorithms that use syntactic dependence to approximate semantic dependence. We refer to this desirable relationship between syntactic and semantic dependence as the *syntactic–semantic relationship*.[5]

3 Interprocedural Control Dependence

In this section, we illustrate three effects that impact interprocedural control dependences. We then define interprocedural control dependence.

[3] Podgurski and Clarke present additional definitions of control and syntactic dependence that provide a necessary condition for semantic dependence for programs that do not halt.

[4] A proof of this theorem is given in [22], and sketched in [23].

[5] Of course, there is a trivial way to construct an algorithm that preserves the syntactic-semantic relationship: the algorithm simply makes every statement syntactically (control or data) dependent on every other statement. Clearly, this approach is unsatisfactory. The goal of an algorithm for approximating semantic dependencies, therefore, is to compute sufficiently tight approximations.

3.1 Effects that impact interprocedural control dependences

Figure 1 presents a program `Sum` that consists of three procedures: `M` (the entry procedure), `B`, and `C`. The two insets in the figure provide alternative versions of two lines of the program; we use these alternatives to illustrate specific points. Intraprocedural control-dependence analysis operates independently on individual procedures, ignoring both the context in which each procedure is invoked, and the side-effects on control dependence that may be caused by a called procedure. Table 1 illustrates the intraprocedural control dependences for `Sum`.

Table 1: Intraprocedural control dependences for `Sum`

Statements	Control Dependent On	Statements	Control Dependent On
2, 3, 4, 6, 7	entry M	4, 5	4
10, 11, 14	entry B	12, 13	11
17	entry C	18	17

Considering `Sum` as a whole, however, we can observe three ways in which control dependences that are computed intraprocedurally inaccurately model the semantic dependences that exist between statements in the program.

First, consider the version of `Sum` created by substituting the alternative versions of lines 6 and 18: this version contains only one call to `B`, and halts (assuming normal termination) only on reaching the implicit halt in statement 8. In this version, statement 4 immediately determines whether statement 5 (the call to `B`) executes, and in so doing, immediately determines whether statements 10, 11, and 14 in `B` and statement 17 in `C` execute. It is easy to show that in terms of Definition 11, statements 10, 11, 14, and 17 are semantically dependent on statement 4, even in the absence of data dependences. To preserve the syntactic–semantic relationship, interprocedural control-dependence analysis must identify statements 10, 11, 14, and 17 as control dependent on statement 4; intraprocedural analysis alone does not do this. We call this the *entry-dependence effect*.

Second, consider the version of `Sum` created by substituting the alternative version of line 18, but not substituting the alternative version of line 6: this version contains both calls to `B`, but halts (assuming normal termination) only on reaching statement 8. The presence of the second, unconditional call to `B` in statement 6 means that, assuming normal termination, statements 10, 11, 14, and 17 necessarily execute at least once during any execution of `Sum`. Moreover, these statements execute regardless of the evaluation of statement 4. One possible application of the definition of postdominance (Definition 3) might seem to imply that statements 10, 11, 14 and 17 postdominate statement 4, and thus, cannot be control dependent on statement 4. (Reference [17] draws this conclusion.) However, despite the fact that the second call to `B` guarantees that statements 10, 11, 14, and 17 execute at least once, statement 4 does determine the number of times that those statements execute. Thus, statements 10, 11, 14, and 17 are semantically dependent on statement 4, even in the absence of data dependences. It follows that to preserve the syntactic–semantic relationship, interprocedural control-dependence analysis must identify statements 10, 11, 14, and 17 as control dependent on statement 4. We call this the *multiple-context effect*.

Third, consider the version of `Sum` presented in the figure, with neither alternative line substituted: in

this case, the program can also halt at statement 18. This explicit `halt` statement has far-reaching effects on the control dependences in `Sum`—effects that combine with the first two effects to further complicate the program's interprocedural control dependences. For example, in this version of `Sum`, statements 11 and 14 depend most immediately for their execution on statement 17, because of the explicit `halt` statement. It is easy to show that in terms of Definition 11, statements 11 and 14 are semantically dependent on statement 17, even in the absence of data dependences. As a second example, statement 4 is now also semantically dependent on statement 17: when the predicate in statement 17 is true, statement 4 executes at least one time fewer than when the predicate in statement 17 is false. Furthermore, statement 6 is now semantically dependent on statement 4: the presence of the `halt` that is reachable from the call to `B` has the effect that now, different interpretations of the function associated with statement 4 affect whether statement 6 is reached (and thus determine the number of times that it executes). To preserve the syntactic–semantic relationship, interprocedural control-dependence analysis must identify the control dependences that are responsible for these semantic dependences. We call this the *return-dependence effect*. We call the explicit `halt` statements that can cause this effect *embedded halts*, to distinguish them from implicit program termination points.

Embedded halts are not the only cause of return-dependence effects. Other language constructs, such as setjump–longjump statements in C, and exception-handling constructs in Java and C++, also cause these effects. In this paper, we restrict our attention to embedded halts.

Table 2: Presence of Embedded Halts in C Programs

Program Group	Number of Programs	Number of Programs that Contain Embedded Halts
Aristotle	20	10
Eli	23	11
Empire	1	1
GCC	20	19
Omega	1	1
Siemens	7	3
XVCG	1	1
Total	73	46

To obtain initial data about the use of embedded halts in practice in a language that supports them, we examined a variety of non-trivial C programs. Table 2 summarizes the programs we examined. We examined 20 programs from the `Aristotle` analysis system [11]; 23 programs from the `Eli` text processor generation system; the `Empire` Internet game; 20 programs from the `GCC` 2.3.3 compiler distribution; the `Omega` data-dependence analyzer;[6] seven programs that were used by researchers at Siemens for a study on data-flow testing [15]; and the `XVCG` tool for displaying graphs.

In C, halt functionality is provided by the `exit()` system call. Where possible, we used `Aristotle` to analyze the source code, and inspected the analysis information to determine if an `exit()` in some function other than `main` could be reached (statically) from the beginning of the program. (By ignoring `exit()` statements in `main` we were able to exclude "non-embedded" `exit()` statements, used unconditionally at the ends of the programs, that cannot affect control dependences.) For programs that `Aristotle` could not completely analyze, we determined this information by manual inspection of the source code. As Table

[6] See `http://www.cs.umd.edu/projects/omega/omega.html` for information about the Omega project.

2 illustrates, over 63% of the programs we examined, and at least 42% of the programs in each group of programs, contained `exit()` statements. Although further study is necessary to determine the extent to which these results generalize, the results do support hypotheses that (a) in C programs, embedded halts are used frequently, and (b) this frequent use is consistent over a range of programs.

Table 3 illustrates the complete set of interprocedural control dependences necessary to preserve the syntactic–semantic relationship for `Sum`. A comparison of these dependences with those computed intraprocedurally (see Table 1) reveals extensive differences. The intraprocedural and interprocedural dependences include seven in common, the intraprocedural dependences include seven not required in the interprocedural context, and the interprocedural dependences include seven not detected by the intraprocedural analysis.

Table 3: Interprocedural control dependences for `Sum`

Statements	Control Dependent On	Statements	Control Dependent On
2, 3, 4	entry M	5, 6, 10, 17	4
4, 7, 11, 14, 18	17	12, 13	11

3.2 Definition of interprocedural control dependence

The entry-dependence, multiple-context, and return-dependence effects constitute three ways in which intraprocedural control dependence computation fails to preserve the syntactic–semantic relationship with respect to control dependences for whole programs. Other effects may also exist. To preserve the syntactic–semantic relationship, a definition of interprocedural control dependence must account for all such effects; this section provides such a definition.

Our definition relies on an *interprocedural inlined flow graph* (IIFG). An IIFG is the graph, possibly infinite, that results when we inline all procedures at their call sites and construct a control flow graph from the resulting program; as such, an IIFG represents the control flow in a program that has been *rolled out* [4]. Like the invocation graph [7] and the context graph [1], the IIFG is fully context-sensitive: it accounts for the calling sequence that leads to each call. However, unlike the invocation and context graphs, the IIFG is also flow-sensitive: it accounts for the control flow of the individual procedures. We define the IIFG more formally as follows:

Definition 12. Let $\mathcal{P}$ be a program, and let , be a collection of CFGs, $G_k, k > 0$, that contains, for each procedure P_i, $i > 0$, in $\mathcal{P}$, one copy of the CFG for each call site in $\mathcal{P}$ that calls P_i. Furthermore, let E be the set of edges and N be the set of nodes in the $G_k, k > 0$. An *interprocedural inlined flow graph* (IIFG) $\mathcal{G}^I = (\mathcal{N}^I, \mathcal{E}^I)$ for $\mathcal{P}$ is a directed graph: $\mathcal{N}^I = N \cup \{n_{stop}\}$, $\mathcal{E}^I = (E - CR - HX) \cup CE \cup XR \cup HS$; n_{stop} is a unique node that represents exit from $\mathcal{P}$; CR is the set of edges from call nodes to the corresponding return nodes; HX is the set of edges from nodes that represent embedded halts to the exit nodes of the respective CFGs; CE is the set of edges from call nodes to the entry nodes of the G_k; XR is the set of edges from the exit nodes of the G_k to the return nodes; and HS is the set of edges from nodes that represent embedded halts to n_{stop}.

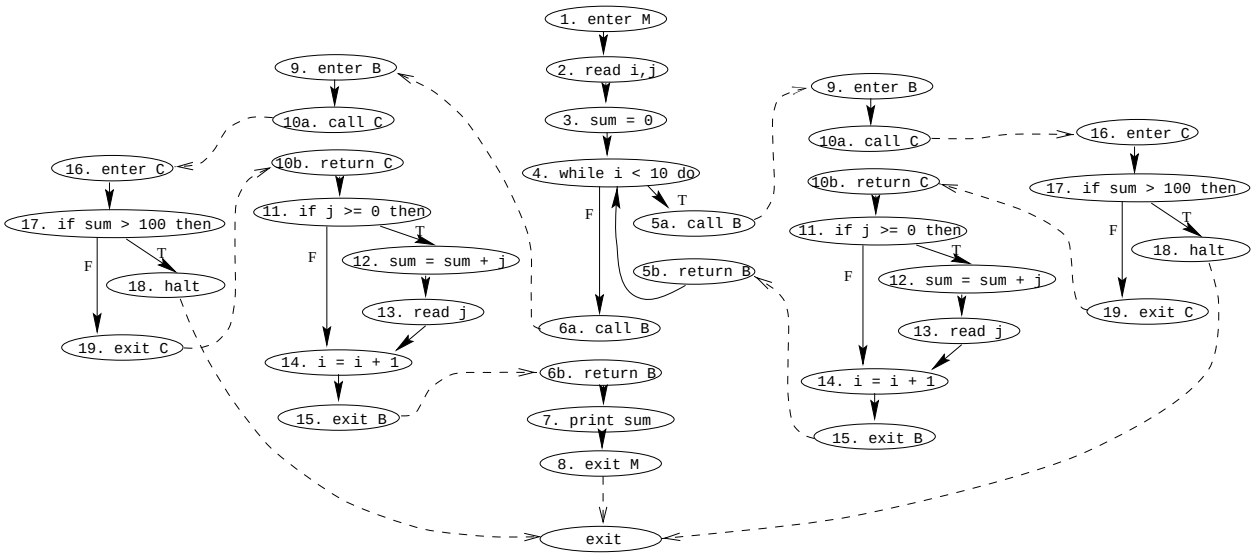

Figure 2: Interprocedural in-lined flow graph for program `Sum`.

Note that a given statement in $\mathcal{P}$ corresponds to a set of IIFG nodes—one for each calling context in which the statement can be executed. We denote the set of nodes in $\mathcal{G}^I$ to which a given statement s in $\mathcal{P}$ corresponds by $NodeSet(s)$.

Figure 2 depicts the IIFG for program `Sum`. Each call site is represented by call and return nodes; the CFG for the called procedure is inlined at each call node. The CFGs are connected by (call node, entry node) and (exit node, return node) edges, shown as dashed lines. The IIFG in Figure 2 contains two copies of the CFG for procedure `B`, corresponding to call nodes 5a and 6a; each in-lined CFG for `B` contains a call to `C`, and therefore the IIFG contains two copies of the CFG for `C` as well. Nodes from which control can exit the program (nodes 8 and 18) are connected to a unique exit node.

The definitions of paths, postdominance, control dependence, def/use graphs, data dependence, syntactic dependence, and semantic dependence presented in Section 2 apply to IIFGs as follows:

Definition 13. A *path* in an IIFG $\mathcal{G}^I = (N^I, E^I)$ is a sequence of nodes $W = n_1 n_2 \ldots n_k$, such that $k \geq 0$, and such that, if $k \geq 2$, then for $i = 1, 2, \ldots, k-1$, $(n_i, n_{i+1}) \in E^I$.

Definition 14. Let $\mathcal{G}^I = (N^I, E^I)$ be an IIFG. A node $u \in N^I$ *postdominates* a node $v \in N^I$ if and only if every v–n_{stop} path in $\mathcal{G}^I$ contains u.

Definition 15. Let $\mathcal{G}^I = (N^I, E^I)$ be an IIFG, and let $u, v \in N^I$. Node u is *control dependent* on node v if and only if v has successors v' and v'' such that u postdominates v' but u does not postdominate v''.

Definition 16. A *def/use IIFG* is a quadruple $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$, where $\mathcal{G}^I = (N^I, E^I)$ is an IIFG, Σ is a finite set of symbols called variables, and $D : N^I \to \mathbf{P}(\Sigma)$, $U : N^I \to \mathbf{P}(\Sigma)$ are functions.

Definition 17. Let $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$ be a def/use IIFG with $\mathcal{G}^I = (N^I, E^I)$, and let $u, v \in N^I$. Node u is *data dependent* on node v if and only if there exists a path vWu in $\mathcal{G}^{I-du}$ such that $(D(v) \cap U(u)) - D(W) \neq \phi$, where $D(W) = \cup_{n_i \in W}(n_i \notin \{u, v\})D(n_i)$.

Definition 18. Let $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$ be a def/use IIFG with $\mathcal{G}^I = (N^I, E^I)$, and let $u, v \in N^I$. Node u is *syntactically dependent* on node v if and only if there is a sequence $n_1, n_2, \ldots, n_k$ of nodes, $k \geq 2$, such that $v = n_1$, $u = n_k$, and for $i = 1, 2, \ldots, k-1$ either n_{i+1} is control dependent on n_i or n_{i+1} is data dependent on n_i.

Definition 19. Let $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$ be a def/use IIFG with $\mathcal{G}^I = (N^I, E^I)$. An *interpretation* of $\mathcal{G}^{I-du}$ is an assignment of partial computable functions to the vertices of $\mathcal{G}^{I-du}$. The function assigned to a vertex $v \in N^I$ is the function computed by the program statement that v represents; it maps values for the variables in $U(v)$ to values for the variables in $D(v)$ or, if v is a decision vertex, to a successor of v.

Definition 20. Let $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$ be a def/use IIFG with $\mathcal{G}^I = (N^I, E^I)$. An *execution history* of a vertex $v \in N^I$ is the sequence whose ith element is the assignment of values held by the variables of $U(v)$ just before the ith time v is visited during a computation.

Definition 21. A node u in a def/use IIFG $\mathcal{G}^{I-du}$ is *semantically dependent* on a node v in $\mathcal{G}^{I-du}$ if there are interpretations I_1 and I_2 of $\mathcal{G}^{I-du}$ that differ only in the function assigned to v, such that for some input, the execution history of u induced by I_1 differs from that induced by I_2.

Given these definitions, the following theorem holds:

Theorem 2. Let $\mathcal{G}^{I-du} = (\mathcal{G}^I, \Sigma, D, U)$ be a def/use IIFG with $\mathcal{G}^I = (N^I, E^I)$, and let $u, v \in N^I$. If u is semantically dependent on v and this semantic dependence is finitely demonstrated, then u is syntactically dependent on v.

The truth of the theorem follows from Podgurski and Clarke's proof of Theorem 1, and the relationship between the graphs used by Podgurski and Clarke in that proof and the IIFG. See Appendix A for further discussion.

Given this theorem, the syntactic–semantic relationship holds for the IIFG-based definition of control dependence (Definition 15). The theorem is significant for reasons similar to those that render Theorem 1 significant: it asserts that given appropriate definitions of control and data dependence, syntactic dependence is a necessary condition for (finitely demonstrated) semantic dependence. However, Theorem 2 applies in the interprocedural context, and thus, provides justification for (and a measure of success of) interprocedural algorithms that use syntactic dependence to approximate semantic dependence.

4 Computing Interprocedural Control Dependences

In this section, we present two approaches to computing interprocedural control dependences. The first approach computes precise interprocedural control dependences, but may be inordinately expensive. The second approach summarizes interprocedural control dependences, and computes a conservative estimate of those dependences more efficiently than the first approach, but at the cost of some precision.

4.1 Precise computation of interprocedural control dependences

One way to compute interprocedural control dependences for a program $\mathcal{P}$ is to build the IIFG $\mathcal{G}^I$ for $\mathcal{P}$, and apply an existing algorithm, such as those described in References [3, 6, 8, 21], to $\mathcal{G}^I$. For non-recursive programs, this approach computes precise interprocedural control dependences.

In practice, this approach may be expensive. The IIFG construction inlines a procedure at each call site to that procedure; thus, the size of an IIFG may be exponential in the size of the program that it represents. Moreover, for a recursive program the IIFG is infinite, and can be constructed only by limiting the number of expansions of the procedures involved in recursion (which in turn limits the precision of the control-dependence computation on that IIFG).

To investigate the cost of this IIFG-based approach, we examined the sizes of the IIFGs for several programs.[7] Table 4 describes the programs that we used in the study, and lists the number of non-comment lines of code in the programs. Table 5 provides data about the sizes of the IIFGs of those programs. For programs that contained recursion, we expanded the recursive procedures once, and determined the increase in IIFG size that would be caused by each additional expansion.

Table 4: Programs used for the empirical studies reported in the paper.

Subject	Description	LOC
armenu	`Aristotle` Analysis System [11] user interface	5835
dejavu	Interprocedural regression test selector [26]	2655
diff	File-differencing tool	1447
flex	Lexical analyzer generator	4357
mpegplayer	MPEG player	5380
netmaze	3-D maze combat game	4688
space	Parser for antenna-array description language	5889
unzip	Zipfile extract utility	2370

As the data illustrates, three of the programs—`armenu`, `flex`, and `mpegplayer`—exhibited between one and two orders-of-magnitude increases in their IIFG sizes over the sizes of their respective CFGs. For the remaining four programs, including one program that contained recursion, the IIFG sizes increased by factors of 2 to 5 over the sizes of the respective CFGs.

Figure 3 shows the increases in IIFG sizes as a bar graph; the vertical axis shows the factor of increase in IIFG size over the sizes of the CFGs. The factor of increase ranges from 2 to 59; for recursive programs, this factor would increase further with each additional expansion of the recursive procedures. Among the non-recursive programs, `mpegplayer` exhibits the largest increase in size—by a factor of 59—from 4,705 nodes in the CFGs to 278,267 nodes in the IIFG.

These results suggest that using the IIFG and a traditional algorithm [3, 6, 8, 21] to compute interprocedural dependences for whole programs may be inordinately expensive. This expense must be weighed, in practice, against the precision requirements of particular applications. Nevertheless, it seems reasonable to

[7] This set of programs differs from the set we used for our study of the occurrence of embedded halts, reported in Table 2. The objective of the embedded-halt study was to motivate our research, and the study required only limited processing of the programs. Thus, we were able to examine a large number of programs and conclude that embedded halts do occur often in practice. However, the empirical studies reported in this section, required extensive processing of the programs with our prototype tools, and examination of the results for correctness. Thus, we selected a subset of the programs for study. Future work includes more experimentation with additional subjects, including those from Table 2.

Table 5: IIFG sizes for programs with each recursion expanded once.

Subject	CFGs	Nodes	IIFG Size		Increase per Recursion Expansion	
			CFGs	Nodes	CFGs	Nodes
armenu	93	8027	16425	474650	15444	451656
dejavu	90	3485	227	6872	–	–
diff	41	1876	232	7193	–	–
flex	88	3728	4435	109289	3092	85803
mpegplayer	105	4705	18528	278267	–	–
netmaze	91	4585	402	15602	–	–
space	136	5725	1552	30662	–	–
unzip	37	2038	206	6074	10	270

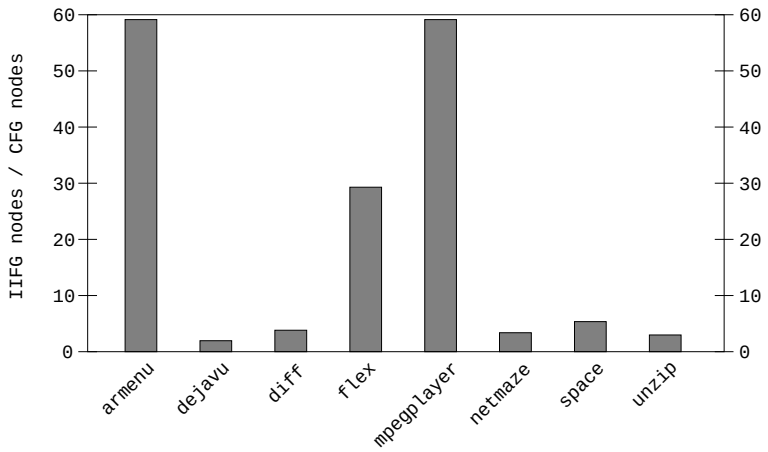

Figure 3: Factors of increase in IIFG sizes over the sizes of the CFGs.

seek alternative approaches to computing interprocedural control dependences, that sacrifice precision for efficiency, while remaining conservative in that they do not omit interprocedural dependences that do exist in a program. We next present one such approach.

4.2 Efficient, safe computation of interprocedural control dependences

The IIFG-based approach just presented computes interprocedural control dependences between nodes in the IIFG. A program statement may correspond to several nodes in an IIFG—one node for each calling context in which the procedure containing the statement can execute. Therefore, the IIFG-based approach computes distinct control dependences for each context in which a statement can execute; we call such dependences *context-based interprocedural control dependences.* An alternative approach to computing interprocedural control dependences is to ignore the context-based distinctions, and instead, compute those dependences by summarizing the control dependences that exist in at least one context of execution of a statement; we call such dependences *statement-based interprocedural control dependences.*

A precise definition of statement-based interprocedural control dependence, in terms of IIFG nodes and the *NodeSet* function, is as follows:

Definition 22: Let $\mathcal{P}$ be a program, let $\mathcal{G}^I$ be the IIFG for $\mathcal{P}$, and let s_1 and s_2 be statements in $\mathcal{P}$. Statement s_1 is control dependent on statement s_2 if and only if there exist nodes $u, v \in \mathcal{G}^I$ such that $u \in \mathit{NodeSet}(s_1)$, $v \in \mathit{NodeSet}(s_2)$, and u is control dependent on v.

Because statement-based control dependences summarize control dependences that exist in different contexts, they do not encode control-dependence information as precisely as do context-based control dependences. However, because context-based control dependences defined on the IIFG preserve the syntactic–semantic relationship, statement-based control dependences, which summarize those control dependences, also preserve that relationship. We present here a brief proof. The proof requires a definition of what it means for a statement to be semantically or syntactically dependent on another statement. Informally, we say such a dependence exists if it exists in some calling context. More formally, we say that a statement $s_1 \in \mathcal{P}$ is (semantically/syntactically) dependent on another statement $s_2 \in \mathcal{P}$ if and only if there exist nodes $n_1, n_2 \in \mathcal{G}^{I-du}$, the def/use IIFG for $\mathcal{P}$, such that $n_1 \in \mathit{NodeSet}(s_1)$ and $n_2 \in \mathit{NodeSet}(s_2)$, and such that n_1 is (semantically/syntactically) dependent on n_2. The proof then proceeds as follows. Suppose s_1 is semantically dependent on s_2. Then there exist nodes $n_1, n_2 \in \mathcal{G}^{I-du}$, the def/use IIFG for $\mathcal{P}$, such that $n_1 \in \mathit{NodeSet}(s_1)$ and $n_2 \in \mathit{NodeSet}(s_2)$, and such that n_1 is semantically dependent on n_2. But then, by Theorem 2, n_1 is syntactically dependent on n_2; thus, s_1 is syntactically dependent on s_2.

Given Definition 22, it follows that we could compute statement-based interprocedural control dependences by first computing context-based control dependences on the IIFG, and then using *NodeSet* associations to transform these into statement-based dependences. Such an approach, of course, would be more expensive than simply computing context-based control dependences. A more efficient algorithm exists, however, that does not require an IIFG. This algorithm uses a representation that is linear in the size of a program to precisely compute the same statement-based interprocedural control dependences that would be computed using the IIFG.

4.2.1 The algorithm

The algorithm proceeds in two phases: (1) Phase 1 identifies call sites to which control may not return due to the presence of embedded halts, and uses this information to compute partial control dependences and construct an augmented control-dependence graph for each procedure; (2) Phase 2 connects the augmented control-dependence graphs for the procedures to construct an interprocedural control-dependence graph for the program, and traverses the graph to compute interprocedural control dependences.

Phase 1: Computation of partial control dependences

The computation of partial control dependences, performed by the first phase of our algorithm, accounts for the effects of embedded halts. To compute partial control dependences, we augment the CFG with "placeholder" nodes that represent the potential effects of external control dependences on nodes within the CFG; we call the resulting graph an *augmented control-flow graph* (ACFG). We define an ACFG more formally as follows:

Definition 23. Let G be a CFG for procedure P in $\mathcal{P}$, let N be the set of nodes in G, let E be the set of edges in G, let $CN = \{CN_1, CN_2, \ldots, CN_j\}$ be nodes in G that represent call sites where control may not return from the called procedures due to the presence of embedded halts,

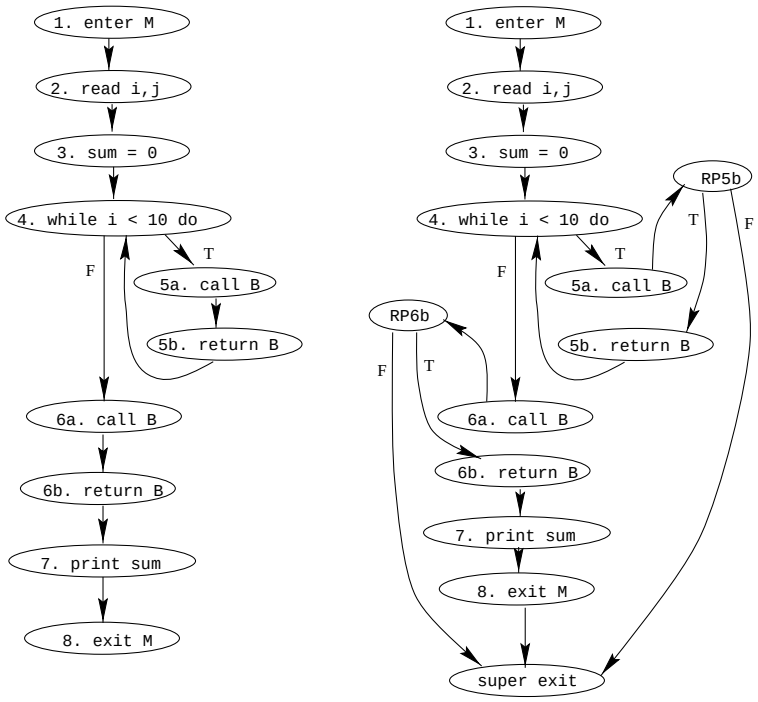

Figure 4: Control-flow graph for procedure `M` (left), and augmented control-flow graph for `M` (right).

---

and let RN be the set of return nodes associated with the call nodes in CN. An *augmented control-flow graph* (ACFG) $G^A = (N^A, E^A)$ is a directed graph: $N^A = N \cup \{n_{sx}\} \cup RP$; $E^A = (E - CR - HX) \cup CP \cup PE \cup SX$; n_{sx} is a unique *super-exit* node that represents all potential exits from P; RP is a set of *return-predicate* nodes, one for each call node in CN, that represent predicates that are external to P and affect the control dependences of statements in P; CR is the set of edges from each call node in CN to its corresponding return node; HX is the set of edges from nodes that represent embedded halts to the exit node; CP is a set of edges, one from each node $n \in CN$ to the node in RP associated with n; PE is a set of edges, one labeled 'T' from each node $n \in RP$ to the node in RN associated with n, and one labeled 'F' from each node $n \in RP$ to n_{sx}; and SX is a set of edges that connect each exit from P to n_{sx}.

To illustrate, Figure 4 displays the CFG and the ACFG for procedure `M` from our example program. The ACFG contains a super-exit node, representing all exit points from the procedure, that is connected to the rest of the graph by edge (exit M, super exit). The graph contains return-predicate nodes RP5b, representing the predicates on which the return from the call at 5a depends, and RP6b, representing the predicates on which return from the call at 6a depends. Edge (RP5b, 5b) with label 'T' represents control returning from `B`, and edge (RP5b, super exit) with label 'F', represents control not returning from `B`. The edge (5a, RP5b) represents the fact that following the call, predicates in the called procedures determine whether control returns to procedure `M`. The graph contains similar edges for return-predicate node RP6b.

The definitions of paths, postdominance, and control dependence apply to the ACFG as follows:

Definition 24. A *path* in an ACFG $G^A = (N^A, E^A)$ is a sequence of nodes $W = n_1 n_2 \ldots n_k$ such that $k \geq 0$, and such that, if $k \geq 2$, then for $i = 1, 2, \ldots, k-1$, $(n_i, n_{i+1}) \in E^A$.

Definition 25. Let $G^A = (N^A, E^A)$ be an ACFG. A node $u \in N^A$ *postdominates* a node $v \in N^A$ if and only if every v–n_{sx} path in G^A contains u.

Definition 26. Let $G^A = (N^A, E^A)$ be an ACFG, and let $u, v \in N^A$. Node u is *control dependent* on node v if and only if v has successors v' and v'' such that u postdominates v' but u does not postdominate v''.

Partial control dependences are the control dependences computed using the ACFG. As in the intraprocedural control-dependence computation [8], to compute partial control dependences, we add a dummy predicate node n_s, an edge (n_s, n_e) labeled 'true', and an edge (n_s, n_{sx}) labeled 'false' to the ACFG. Table 6 shows the partial control dependences computed from the ACFGs for the procedures in `Sum`.

Table 6: Partial control dependences for `Sum` computed using ACFGs

Statements	Control Dependent On	Statements	Control Dependent On	Statements	Control Dependent On
2, 3, 4	entry M	5a, 6a	4	4, 5b	RP5b
6b, 7, 8	RP6b	9, 10a	entry B	10b, 11	RP10b
12, 13	11	16, 17	entry C	18, 19	17

To represent partial control dependences, Phase 1 of the algorithm constructs an *augmented control-dependence graph* (ACDG); we define an ACDG more formally as follows:

Definition 27. Let G^A be an ACFG for procedure P in $\mathcal{P}$, let N^A be the set of nodes in G^A, and let $RP = \{RP_1, RP_2, \ldots, RP_j\}$ be return-predicate nodes in G^A with the corresponding return nodes $RN = \{RN_1, RN_2, \ldots, RN_j\}$. An *augmented control-dependence graph* (ACDG) $G^D = (N^D, E^D)$ is a directed graph: $N^D = N^A - RP - n_{sx}$; $E^D = CD \cup E \cup R$; CD is a set of edges, and it contains an edge (n_1, n_2), $n_1, n_2 \in N^D$, if the partial control dependences for n_2 include n_1; E is a set of edges, and it contains an edge (n_e, n), $n \neq n_e$, labeled 'T' if the partial control dependences for n include n_s; R is a set of edges, and it contains an edge (r, n), $n \neq r$, labeled 'T' if the partial control dependences for n include RP_r, where $RP_r \in RP$ is the return-predicate node associated with return node r. Each node $n \in N^D - (n_e \cup RN)$ has at least one predecessor and no successors; nodes in $(n_e \cup RN)$ have no predecessors.

Figure 5 shows the ACDG for procedure `M`. The ACDG contains control-dependence edges to represent partial control dependences. The source of each control-dependence edge is a predicate node or a placeholder node, which is either an entry node or a return node for a PNRC. If a node n is control dependent on the dummy start predicate n_s, the ACDG contains an edge from the entry node n_e to n. If a node n is control dependent on a return predicate, the ACDG contains an edge from the return node associated with that return predicate to n; thus, the ACDG contains no return-predicate nodes. For example, the partial control dependences for procedure `M` show that node 4 is control dependent on return predicate `RP5b`. Therefore,

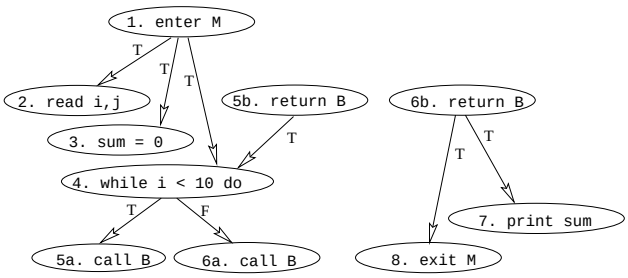

Figure 5: Augmented control-dependence graph for procedure `M`.

in the ACDG for `M`, there exists an edge from node 5b—the return node associated with return predicate `RP5b`—to node 4.

Figure 5 illustrates that the ACDG can have multiple root nodes. Each root node represents a point in the corresponding procedure P where control enters P—either through a call site that calls P or through a return site in P—and where external predicates control the statements in P that are reached from that entry. The entry node and the return nodes for PNRCs represent such points in a procedure, and therefore, appear as root nodes in the ACDG. Each root node in the ACDG is thus a placeholder for external predicates. As the figure illustrates, the ACDG can also have disconnected components.

Our approach to computing partial control dependences involves two main steps. In Step 1, given program $\mathcal{P}$, we identify the call sites in $\mathcal{P}$ where control, on entering the called procedure, may fail to return to the caller due to the presence of an embedded halt. In Step 2, we use this information to construct ACFGs and ACDGs for the procedures in $\mathcal{P}$.

Figure 6 presents our algorithm, `ComputePartialCD`. The algorithm takes as input the set of CFGs $\{CFG_1, CFG_2, \ldots, CFG_j\}$ for procedures $\{P_1, P_2, \ldots, P_j\}$, respectively, in program $\mathcal{P}$, and outputs, for each P_i in $\mathcal{P}$, the ACDG for P_i. The algorithm proceeds in two steps, which correspond to the steps described above. We next describe each step of the algorithm in turn.

Step 1 (line 1) of `ComputePartialCD` calls procedure `ClassifyCallSites()` to identify *potentially non-returning call sites* (PNRCs) in $\mathcal{P}$: call sites to which control may not return from called procedures due to the presence of embedded halts. To identify these call sites, `ClassifyCallSites` requires information about interprocedural flow of control in $\mathcal{P}$. `ClassifyCallSites` uses the procedure shown in Figure 7 to identify PNRCs in $\mathcal{P}$. To obtain PNRC information, the procedure (line 1) constructs an *interprocedural control-flow graph* (ICFG) that connects individual CFGs at call nodes [16]. We define the ICFG more formally as follows:

> **Definition 28.** Let $\mathcal{P}$ be a program with procedures $P_1, P_2, \ldots, P_j$, let , $= G_1, G_2, \ldots, G_j$ be the corresponding CFGs for the P_i, $1 \leq i \leq j$, let E be the set of edges in the G_i, let N be the set of nodes in the G_i. An *interprocedural control-flow graph* (ICFG) for $\mathcal{P}$, $\mathcal{G}^C = (\mathcal{N}^C, \mathcal{E}^C)$ is a directed graph: $\mathcal{N}^C = N$; $\mathcal{E}^C = (E - HX - CR) \cup CE \cup XR$; HX is a set of edges connecting nodes that represent embedded halts to exit nodes; CR is a set of edges connecting call nodes to return nodes; CE is a set of (call node, entry node) edges, one from each call node to the entry node of the called procedure; and XR is a set of (exit node, return node) edges, one to each return node from the exit node of the procedure returned from. Each statement in $\mathcal{P}$ corresponds to a unique node in the ICFG for $\mathcal{P}$.

```
algorithm ComputePartialCD
input     CFG        set of CFG_i: CFG for each procedure P_i in program P
output    ACDG       set of ACDG_i: augmented control-dependence graph for each procedure P_i
declare   PNRCList   set of PNRCList_i: call nodes in each CFG_i that represent potentially non-returning
                     call sites
          HNList     set of HNList_i: halt nodes in each CFG_i
          ACFG_i     augmented control-flow graph for procedure P_i

begin ComputePartialCD
  /* Step 1: Identify potentially non-returning call sites, and record on PNRCList */
1.   PNRCList = ClassifyCallSites()
  /* Step 2: Compute partial control dependences for P_i */
2.   foreach P_i in P do
3.       ACFG_i = CFG_i
4.       N_x = exit node of CFG_i
5.       Create node N_sx labeled 'super exit' and add to ACFG_i
6.       Create edge (N_x, N_sx)
7.       foreach call node C in PNRCList_i with associated return node R do
8.           Create node RP[C] and add to ACFG_i
9.           Remove edge (C, R)
10.          Create edge (C, RP[C])
11.          Create edge (RP[C], R) labeled 'T'
12.          Create edge (RP[C], N_sx) labeled 'F'
13.      endfor
14.      foreach node H in HNList_i do
15.          Remove edge (H, N_x)
16.          Create edge (H, N_sx)
17.      endfor
18.      Compute intraprocedural control dependences using ACFG_i
19.      Construct augmented-control dependence graph ACDG_i for P_i
20.  endfor
21.  return ACDG_is
end ComputePartialCD
```

Figure 6: The algorithm for computing partial control dependences.

```
procedure ClassifyCallSites
input     CFG          set of CFG_i: control-flow graph for each procedure P_i in program P
output    PNRCList     set of PNRCList_i: list of the PNRCs in procedure P_i
          CFG          set of CFG_i: CFG for procedure P_i with statically unreachable nodes removed
declare   G^C          ICFG for program P with entry node E
          DNRPList     list of procedures in P from which control cannot statically return
          HNList       set of HNList_i: list of nodes in procedure P_i that represent embedded halts
          UnreachList  list of nodes in G^C that are statically unreachable

begin ClassifyCallSites
1.   Construct ICFG G^C for P
2.   Call ComputeDNRPs to retrieve DNRPList, UnreachList, and HNList
3.   Remove all nodes on UnreachList from G^C and CFG_is
4.   PNRCList = ComputePNRCs(G^C, DNRPList, HNList)
5.   return PNRCList and modified CFG_is
end ClassifyCallSites
```

Figure 7: The algorithm for classifying call sites.

Figure 8 depicts the ICFG for program `Sum`. Each call site is represented by call and return nodes; the CFGs are connected by (call node, entry node) and (exit node, return node) edges, shown as dashed lines. Unlike the IIFG, the ICFG contains a single copy of the CFG for each procedure in a program. Also, in the ICFG, nodes that represent `halt` statements are not connected to a unique exit node.

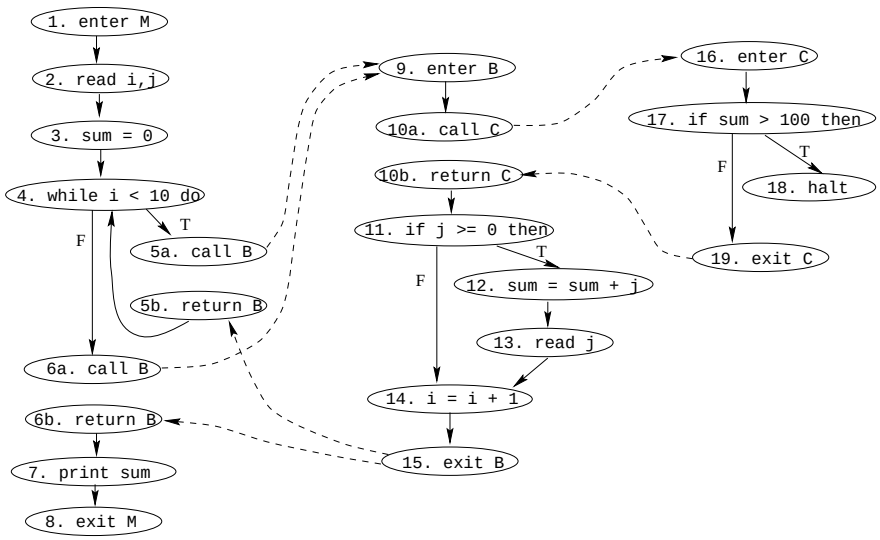

Figure 8: Interprocedural control-flow graph for program `Sum`.

After constructing the ICFG, `ClassifyCallSites` calls procedure `ComputeDNRPs` (line 2). `ComputeDNRPs` calculates three pieces of data: (1) *DNRPList*, the list of *definitely non-returning procedures* (DNRPs) in $\mathcal{P}$: procedures from which control (statically) cannot return, due to the presence of embedded halts; (2) *UnreachList*, a list of nodes in the ICFG that cannot be reached (statically) from the entry node; and (3) *HNList*, a list of nodes in the ICFG that represent embedded halts that can be reached (statically) from the entry node. To calculate this data, `ComputeDNRPs` performs a depth-first traversal along realizable paths[8] in the ICFG, marking nodes as it reaches them, until no unmarked nodes remain. During this traversal, `ComputeDNRPs` places all halt nodes that it reaches on *HNList*. Following the traversal, the procedure examines the exit nodes of individual CFGs in the ICFG. Any exit node that is not marked indicates that the procedure to which that exit node belongs is definitely non-returning; `ComputeDNRPs` places that procedure on *DNRPList*. Also, any unmarked nodes are statically unreachable; `ComputeDNRPs` places these on *UnreachList*.

Following the call to `ComputeDNRPs`, `ClassifyCallSites` uses *UnreachList* to remove all statically unreachable nodes (line 3) from the ICFG and all CFGs.

The algorithm can now determine PNRCs. The procedure for accomplishing this, `ComputePNRCs`, takes as input the ICFG, from which unreachable nodes have been removed, the list of definitely non-returning procedures *DNRPList*, and the list of halt nodes *HNList*. The procedure performs a reverse depth-first traversal of the ICFG, starting at halt nodes and nodes that represent calls to definitely non-returning procedures, ascending into calling procedures but not descending into called procedures. Any call site reached during the traversal is a PNRC, and the called procedure is potentially non-returning. The algorithm places these call nodes on *PNRCList*. When the traversal terminates, the procedure returns *PNRCList* and the modified CFGs.

To illustrate the operation of `ClassifyCallSites`, consider our example program. Called with the CFGs for this program, `ClassifyCallSites` first creates the ICFG shown in Figure 8. `ComputeDNRPs` determines

[8] A path in an ICFG is *realizable* if whenever control leaves a procedure through a normal procedure exit, such as the end of the procedure or a **return** statement, it returns to the procedure that invoked it.

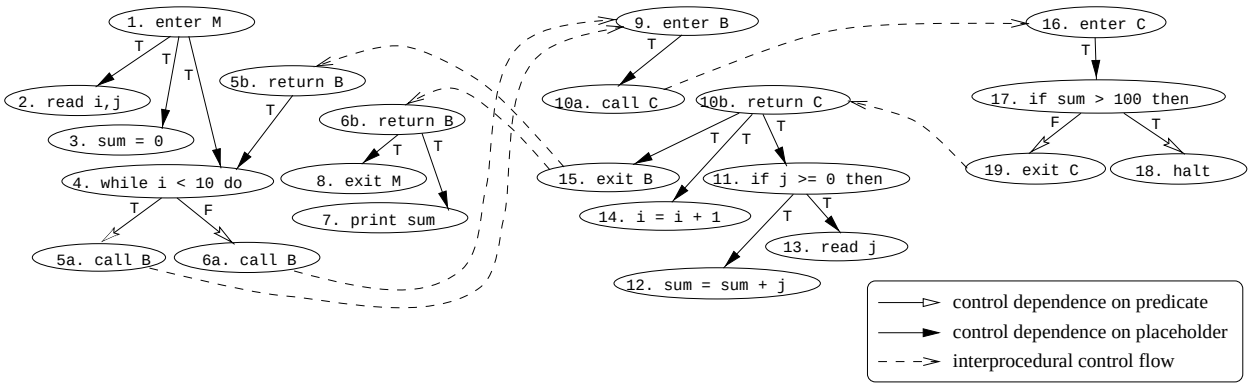

Figure 9: Interprocedural control-dependence graph for program `Sum`.

that no procedures in this program are definitely non-returning and that all nodes in the ICFG are reachable, and adds node 18 to *HNList*. `ComputePNRCs` then performs a reverse depth-first traversal of the ICFG from node 18. During this traversal, the algorithm adds call nodes 5a, 6a, and 10a to *PNRCList* because the associated call sites are potentially non-returning.

Following the identification of PNRCs, in Step 2 (lines 2–20), `ComputePartialCD` (shown in Figure 6) computes the set of partial control dependences and constructs the ACDG for each P_i in $\mathcal{P}$. To do this, `ComputePartialCD` first constructs the ACFG for P_i (lines 3–17). `ComputePartialCD` initializes the ACFG (line 3), and creates and adds the super-exit node N_{sx} to the ACFG (line 5). Next, the algorithm connects the exit node to N_{sx} (line 6). `ComputePartialCD` then iterates through each PNRC in a procedure (lines 7–13), creates a return-predicate node for that PNRC and adds it to the ACFG (line 8), removes the edge that connects the call node to the corresponding return node (line 9), connects the call node to the return-predicate node (line 10), and creates outgoing edges labeled 'T' and 'F' from the return-predicate node (lines 11–12). `ComputePartialCD` also removes edges from halt nodes to the exit node, and connects the halt nodes to the super-exit node (lines 14–17).

Having constructed the ACFG for P_i, `ComputePartialCD` next computes partial control dependences for P_i by applying an existing technique for control-dependence computation [3, 6, 8, 21] to the ACFG for P_i (line 18). Finally, the algorithm constructs the ACDG for P_i (line 19).

Phase 2: Computation of interprocedural control dependences

Intraprocedural control-dependence computation applied to an ACFG produces correct control dependences for all nodes that are control dependent on non-placeholder nodes in the ACFG; however, control dependences for nodes that are control dependent on placeholders—entry or return nodes—must be adjusted. Phase 2 of our algorithm performs this adjustment and computes interprocedural control dependences.

To compute interprocedural control dependences, the algorithm constructs an *interprocedural control-dependence graph* (ICDG). We define an ICDG more formally as follows:

> **Definition 29.** Let $\mathcal{P}$ be a program with procedures $P_1, P_2, \ldots, P_j$, let , $= G_1^D, G_2^D, \ldots, G_j^D$ be the corresponding ACDGs for the P_i, $1 \leq i \leq j$, let N^D be the set of nodes in the G_i^D, and let E^D be the set of edges in the G_i^D. An *interprocedural control-dependence graph* (ICDG) for

```
algorithm ComputeInterCD
input    ACDG     ACDG of each procedure P in program P
         CDPL     nodes whose partial control dependences include a placeholder
         CD(N)    partial control dependences (excluding placeholders) for node N
output   interCD  interprocedural control dependences for P
declare  ICDG     interprocedural control-dependence graph for P
         worklist ICDG nodes traversed by the algorithm

begin ComputeInterCD
 1.    initialize ICDG by connecting ACDGs using call and return edges
 2.    foreach node M in CDPL
 3.        mark each node in ICDG as unvisited
 4.        initialize worklist with placeholder predecessors of M; mark those nodes as visited
 5.        while worklist is not empty
 6.            remove node N from worklist
 7.            foreach predecessor P of N
 8.                if P is a predicate node
 9.                    add P to CD(M)
10.                else
11.                    if P is not visited
12.                        add P to worklist; mark P as visited
13.                    endif
14.                endif
15.            endfor
16.        endwhile
17.    endfor
18.    foreach N in ICFG do
19.        InterCD = InterCD ∪ CD(N)
20.    endfor
21.    return InterCD
end ComputeInterCD
```

Figure 10: The algorithm for computing interprocedural control dependences.

$\mathcal{P}$, $\mathcal{G}^D = (\mathcal{N}^D, \mathcal{E}^D)$ is a directed graph: $\mathcal{N}^D = N^D$; $\mathcal{E}^D = E^D \cup CE \cup XR$; CE is a set of (call node, entry node) edges, one from each call node to the entry node of the called procedure; and XR is a set of (exit node, return node) edges, one to each return node from the exit node of the procedure returned from.

Figure 9 shows the ICDG for program `Sum`. Apart from the control-dependence edges, the ICDG contains call and return edges. At each call site, a call edge connects the call node to the entry node of the called procedure; for example, a call edge connects call node 5a to entry node 9. At each call site, a return edge connects the exit node of the called procedure to the return node for the call site; for example, a return edge connects exit node 15 to return node 5b.

Figure 10 presents `ComputeInterCD`, the algorithm for Phase 2 of the interprocedural control-dependence computation. `ComputeInterCD` takes three inputs: (1) the ACDGs for the procedures in a program, (2) the list of nodes that are control dependent on placeholders (CDPL), and (3) partial control dependences (excluding placeholders) for each node. The algorithm constructs the ICDG by connecting the ACDGs using call and return edges (line 1). Then, the algorithm traverses the ICDG once for each node that is control dependent on a placeholder (lines 2–17). For each such node M, the algorithm traverses the ICDG backwards along all paths, starting at each predecessor P of M that is a placeholder, and identifies the closest predicates that are reachable from P along those paths; P is a placeholder for the external predicates that are reached along the paths. To identify such predicates, the traversal along a path terminates when it reaches a control-dependence edge whose source is a non-placeholder (neither an entry node nor a return

node). The algorithm uses *worklist* to traverse the ICDG, and marks nodes as they are visited.

For each node M that is control dependent on a placeholder, the algorithm initiates the ICDG traversal by marking the ICDG nodes as unvisited (line 3), and initializing *worklist* by adding placeholder predecessors of M to *worklist* (line 4). Following the initialization, the algorithm traverses the ICDG by removing a node from *worklist* and processing it, until *worklist* becomes empty (lines 5–16).

The algorithm removes a node N from *worklist* (line 6) and examines all predecessors of N in the ICDG (lines 7–15). If a predecessor P of N is a predicate node, the algorithm has identified a control dependence for node M. Therefore, the algorithm adds P to the set of control dependences for node M (line 9), and terminates the traversal of the ICDG along that path. For example, to process node 15, which is control dependent on a return node, the algorithm initializes node 15 on `worklist`. Then, the algorithm traverses the path (15, 10b, 19, 17) in the ICDG for `Sum`, and identifies node 17 as the predicate on which node 15 is control dependent. For another example, to process node 10a, which is control dependent on an entry node, the algorithm traverses the ICDG backwards along all paths, starting at node 10a, and identifies node 4 as the predicate on which node 10 is control dependent.

After the algorithm has processed each node that is control dependent on a placeholder, it has identified for each such node the external predicates on which the node is control dependent. Finally, the algorithm builds the set of interprocedural control dependences for the program, and returns it (lines 18–21).

4.2.2 Complexity of the algorithm

The cost of our algorithm for computing interprocedural control dependences is determined by the costs of the two phases of the algorithm—`ComputePartialCD` (Figure 6) and `ComputeInterCD` (Figure 10).

Step 1 of `ComputePartialCD` invokes the procedure `ClassifyCallSites` (Figure 7), which identifies DNRPs, removes unreachable nodes from the ICFG and the CFGs, and identifies PNRCs. Let N and E be the number of nodes and edges, respectively, in the ICFG. The procedure that identifies DNRPs, `ComputeDNRPs`, performs a depth-first traversal of the ICFG; therefore, the cost of `ComputeDNRPs` is $O(N+E)$. Next, `ClassifyCallSites` removes unreachable nodes from the ICFG and CFGs, which can be accomplished in time linear in the sizes of those graphs. Finally, using the list of halt nodes and call nodes for DNRPs, `ClassifyCallSites` identifies PNRCs. The procedure that identifies PNRCs, `ComputePNRCs`, traverses the ICFG once for each halt node and each call node for a DNRP. Let H and C_{DNRP} be the number of halt nodes and call nodes for DNRPs, respectively, in a program. Then, the cost of `ComputePNRCs` is $O((H + C_{DNRP}) * (N + E))$.

Step 2 of `ComputePartialCD` creates the ACFG and computes partial control dependences for each procedure. Let N_C and E_C be the number of nodes and edges in a CFG, and let C_{PNRC} be the number of PNRCs in a procedure. The cost of constructing the ACFG is $O(N_C + E_C + C_{PNRC})$. After constructing the ACFG, Step 2 of `ComputePartialCD` calculates partial control dependences by applying an existing technique for control-dependence computation [3, 6, 8, 21] to the ACFG of each procedure. The costs of these techniques vary from linear [3, 21] to quadratic [6, 8] in the size of the graph to which they are applied.

`ComputeInterCD` traverses the ICDG once for each node that is control dependent on a placeholder; each traversal is linear in the size of the ICDG. Let N be the number of nodes in the ICDG, E be the number of edges in the ICDG, and N_p be the number of ICDG nodes that are control dependent on a placeholder. Then, the worst-case complexity of `ComputeInterCD` is $O(N_p * (N + E))$.

4.2.3 Correctness of the algorithm

Our algorithm computes statement-based interprocedural control dependences by summarizing, for each statement, the control dependences that exist in different contexts for that statement in the IIFG. To demonstrate the correctness of our algorithm, we show that our algorithm computes the same statement-based control dependences as are computed by an alternative approach that constructs an IIFG, applies a traditional algorithm for control-dependence computation to the IIFG, and summarizes the control dependences for each statement using the *NodeSet* relations.

The overall structure of the proof is as follows. First, we classify paths in the IIFG based on the sequences of call and return edges that appear in the paths. Next, we characterize paths in the ICDG that are traversed by the algorithm. Finally, by considering the types of path in the IIFG along which an interprocedural control dependence relation occurs, and the types of paths that are traversed by our algorithm, we prove the following theorem. (The appendix provides an outline of the proof; further details of the proof can be found in Reference [30].)

> **Theorem 3.** Let $\mathcal{G}^I$ be the IIFG for program $\mathcal{P}$. Let u and v be nodes in $\mathcal{G}^I$. Let s_u and s_v be the statements in $\mathcal{P}$ such that $u \in \mathit{NodeSet}(s_u)$ and $v \in \mathit{NodeSet}(s_v)$. u is control dependent on v if and only if `ComputeInterCD` identifies s_u control dependent on s_v.

4.3 Summary

We have presented two approaches for computing interprocedural control dependences; the first approach computes context-based dependences, and the second computes statement-based dependences. These two approaches lie in a spectrum of approaches that compute interprocedural control dependences with various degrees of context-sensitivity. The context-based approach expands all contexts for a statement, and computes distinct control dependences for the statement in each context. The statement-based approach summarizes all contexts for a statement, and computes a single set of control dependences for that statement. Other approaches, intermediate between these two, may selectively expand the context of a procedure [1], and compute control dependences with varying degrees of precision and efficiency. The ability of such approaches to compute interprocedural control dependences safely can be evaluated using our definition of interprocedural control dependence.

5 Empirical Evaluation

To evaluate our algorithm, we conducted two empirical studies with implementations of `ComputePartialCD` and `ComputeInterCD`. To obtain the CFGs and the intraprocedural control-dependence information required for the studies, we used the analysis tools provided by the `Aristotle` Analysis System [11]; the control-dependence analyzer in the `Aristotle` Analysis System implements the control-dependence algorithm described by Ferrante, Ottenstein, and Warren [8]. We used the programs listed in Table 4 for both the studies.

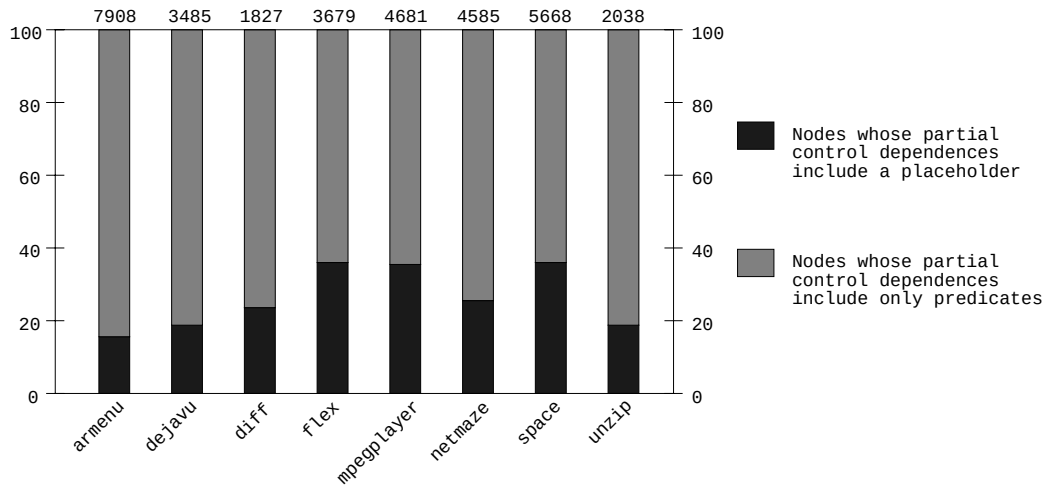

Figure 11: The percentage of nodes whose partial control dependences included an entry or a return node; such nodes were processed by `ComputeInterCD`.

5.1 Efficiency of interprocedural control-dependence computation

The goal of our first study was to evaluate the performance of `ComputeInterCD` in practice. Recall that the complexity of `ComputeInterCD` is $O(N_p * (N + E))$, where N_p is the number of ICDG nodes whose partial control dependences include a placeholder, and N and E are the number of nodes and edges, respectively, in the ICDG. To resolve the control dependences of nodes represented by N_p, `ComputeInterCD` traverses the ICDG starting at those nodes.

Figure 11 presents data about the percentage of nodes whose partial control dependences included an entry or a return placeholder. The number at the top of each bar is the total number of nodes in the ICDG for that program. Five of the programs—`armenu`, `diff`, `flex`, `mpegplayer`, and `space`—contain statically unreachable statements; for these programs, the number of ICDG nodes is less than the number of ICFG nodes (listed in Table 5) because Phase 1 of our algorithm identifies and removes nodes that correspond to statically unreachable statements. The percentage of nodes whose partial control dependences included a placeholder ranged from 15.6% for `armenu` to 36% for `flex`. On average, 26.3% of ICDG nodes were control dependent on a placeholder.

The second factor in the cost equation for `ComputeInterCD` measures the extent of the ICDG that is traversed by `ComputeInterCD` while resolving a node that is control dependent on a placeholder. Although theoretically `ComputeInterCD` can traverse the entire ICDG while processing a node, in practice, we expect it to traverse only a fraction of the ICDG. To test this hypothesis, we gathered data about the percentage of ICDG nodes and edges that were traversed by the algorithm while processing the nodes whose partial control dependences included a placeholder.

Figure 12 presents the percentage of ICDG nodes and edges that were traversed by `ComputeInterCD`; each bar in the figure represents the proportion of ICDG nodes and edges that were traversed, averaged over the nodes that were processed by `ComputeInterCD`. As the figure illustrates, for each program, `ComputeInterCD` traversed fewer than one percent of the nodes and edges in the ICDG: the average was highest at 0.71% for `armenu`, and was as low as 0.13% for `netmaze`. This data strongly supports our belief that the quadratic worst-case performance of `ComputeInterCD` may not be realized in practice, and that `ComputeInterCD` may

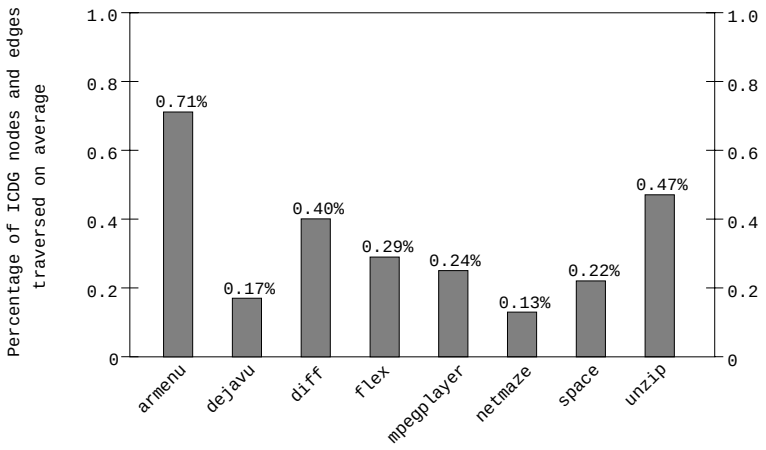

Figure 12: The percentage of nodes and edges in the ICDG that were traversed on average by `ComputeInterCD` for nodes whose partial control dependences included a placeholder.

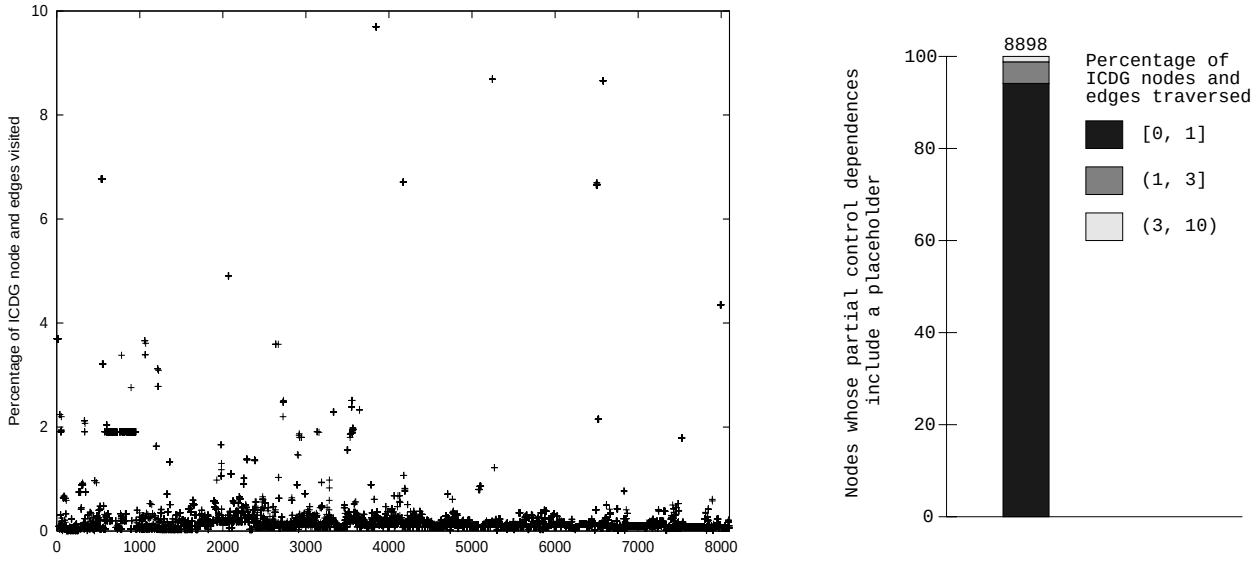

Figure 13: The percentage of nodes and edges in the ICDG that were traversed by `ComputeInterCD` for each node whose partial control dependences included a placeholder (left); the percentage of nodes for which `ComputeInterCD` traversed various percentages of the ICDG nodes and edges (right).

scale well for large programs.

Although Figure 12 shows the percentage of ICDG traversed on average for each program, it does not illustrate the distribution of those percentages. The scatter plot on the left in Figure 13 illustrates the distribution: it shows, for each node that was processed by `ComputeInterCD`, the percentage of the ICDG that was traversed. There are 8,898 data points in the scatter plot, which correspond to the nodes that were processed by `ComputeInterCD`. The cluster of points at the bottom of the plot illustrates that the algorithm traversed a small fraction of the ICDG for most of the nodes. For each node, the algorithm traversed less than 10% of the ICDG nodes and edges. The segmented bar on the right in Figure 13 provides a different view of the data: it partitions the nodes based on the percentage of the ICDG nodes and edges that were

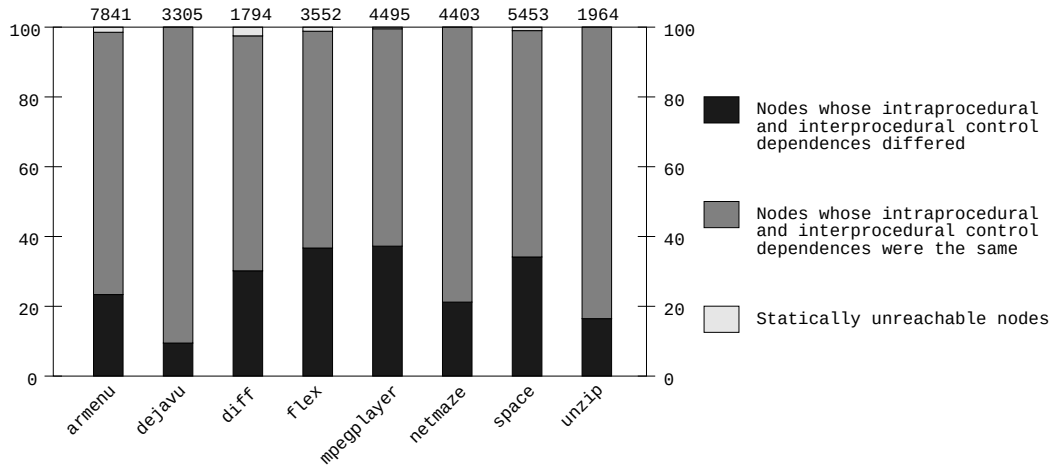

Figure 14: The percentage of nodes whose intraprocedural and interprocedural control dependences differed.

traversed by `ComputeInterCD`. As the figure shows, for over 94% of the nodes, `ComputeInterCD` traversed less than 1% of the ICDG nodes and edges.

5.2 Differences between intraprocedural and interprocedural control dependences

The goal of our second study was to examine the extent to which interprocedural control dependences (computed by our second approach) differ from intraprocedural control dependences (computed by applying a traditional algorithm for computing control dependences [3, 6, 8, 21] to each CFG in a program).[9]

Intraprocedural control-dependence computation does not consider the effects that interactions among procedures can have on control dependences. Therefore, intraprocedural control dependences can exclude dependences that exist because of interactions among procedures; interprocedural control dependences include such dependences. Also, intraprocedural control dependences can contain spurious control dependences—dependences that do not exist when interactions among procedures are considered; interprocedural control dependences exclude such dependences. Finally, intraprocedural control dependences can include dependences that are computed also by the interprocedural control-dependence computation; such common control dependences are unaffected by the interactions among procedures.[10]

Figure 14 shows the percentage of ICFG nodes whose control dependences were affected by the interactions among procedures; such nodes had different intraprocedural and interprocedural control dependences. The numbers at the top of the bars in the figure are the number of ICFG nodes, excluding the entry and the exit nodes, in the respective programs. The figure also shows the percentage of nodes that were statically unreachable; such nodes occur in `armenu`, `diff`, `flex`, `mpegplayer`, and `space`. The percentage of nodes whose control dependences differed ranged from 9.5% for `dejavu` to 37.2% for `mpegplayer`. On average, the control dependences of 26.8% of the nodes differed.

[9] As mentioned earlier, we used an implementation of the control-dependence algorithm described in Reference [8] for the empirical studies. However, the other algorithms [3, 6, 21] would also compute the same control dependences when applied to the CFGs; therefore, the discussion in this section applies to those algorithms as well.

[10] In some cases, intraprocedural control-dependence computation identifies a statement as control dependent on entry into the procedure to which the statement belongs, whereas the interprocedural control-dependence computation identifies that statement as control dependent on entry into the program. In the empirical results reported in this section, we considered such control dependences as common control dependences.

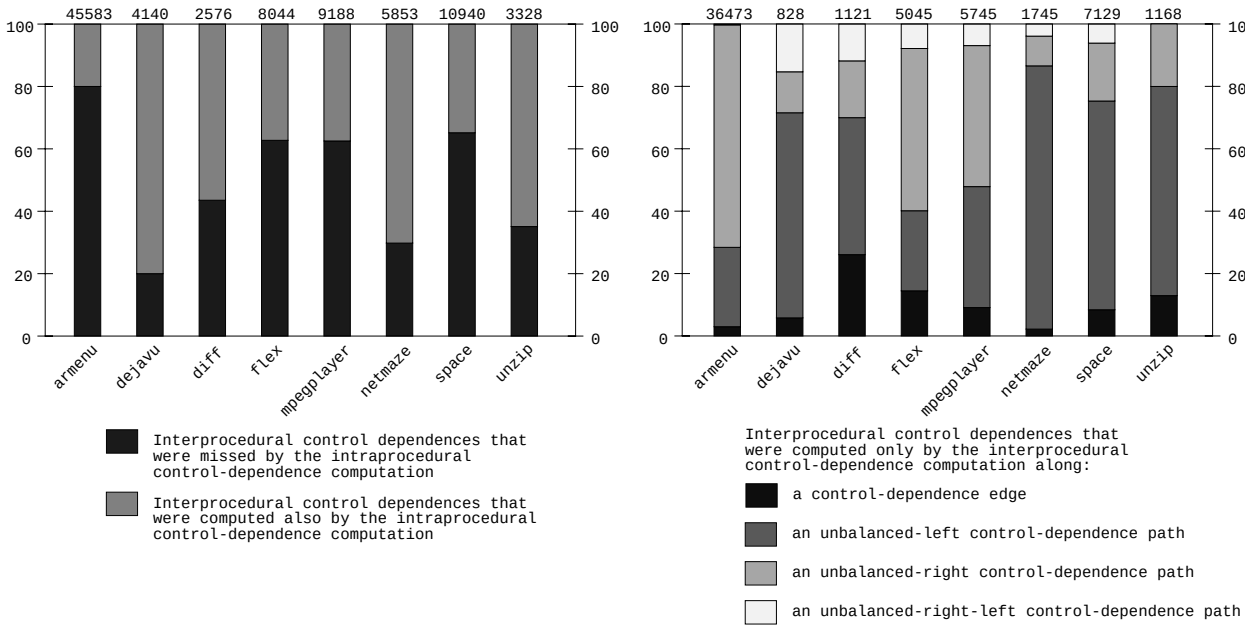

Figure 15: The percentage of interprocedural control dependences that were missed by the intraprocedural control-dependence computation (left), and the classification of those control dependences based on the type of path in the ICDG along which they were computed (right).

Figure 15 presents data about interprocedural control dependences computed for the programs. The graph on the left in the figure shows the percentages of interprocedural control dependences that were computed only by the interprocedural control-dependence computation, and those that were computed also by the intraprocedural control-dependence computation. The number at the top of each bar is the total number of interprocedural control dependences computed for that program. The percentage of control dependences that were missed by the intraprocedural control-dependence computation ranged from 20% for `dejavu` to 80% for `armenu`. On average, 66.1% of the interprocedural control dependences were missed by the intraprocedural control-dependence computation.

Each control dependence that is missed by the intraprocedural control-dependence computation is computed either by `ComputePartialCD` (and received by `ComputeInterCD` as an input), or by `ComputeInterCD` along a control-dependence path in the ICDG. Intuitively, a control-dependence path in the ICDG is a path from a predicate node to a node that is control dependent on a placeholder.[11] Each control-dependence path crosses procedure boundaries and contains one or more call and return edges. The sequence of call and return edges along a control-dependence path can contain (1) only call edges (the path is an unbalanced-left control-dependence path), (2) only return edges (the path is an unbalanced-right control-dependence path), or (3) a subsequence that contains only return edges followed by a subsequence that contains only call edges (the path is an unbalanced-right-left path). Control dependences computed along unbalanced-left control-dependence paths are caused because of call relations among the procedures, whereas control dependences computed during the partial-dependence computation or along unbalanced-right or unbalanced-right-left control-dependence paths are caused because of the effects of PNRCs.

The graph on the right in Figure 15 classifies the missed control dependences using the above criteria.

[11] See Appendix B for formal definitions of control-dependence paths and their types.

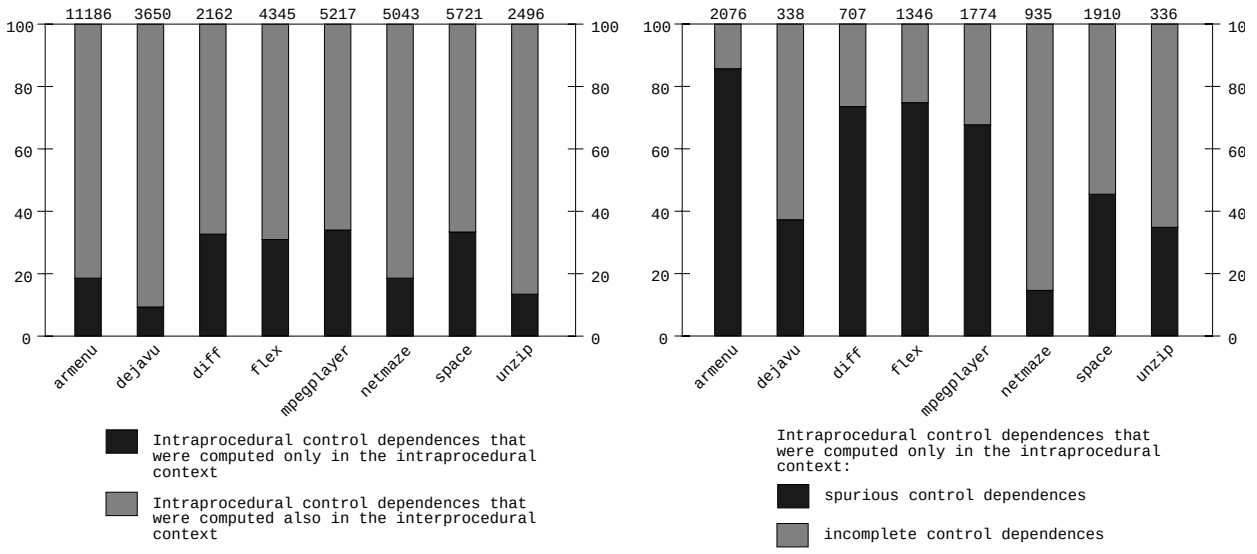

Figure 16: The percentage of intraprocedural control dependences that were computed only by the intraprocedural control-dependence computation (left), and a classification of those control dependences (right).

It shows the percentage of missed control dependences that were computed by `ComputePartialCD` (and represented as control-dependence edges in the ICDG), or by `ComputeInterCD` along different types of control-dependence paths. The number at the top of each bar is the total number of missed control dependences for that program; this number is also represented as a percentage by the darker segment in the graph on the left. The data in the figure illustrate that only a small fraction of the missed interprocedural control dependences were identified during the partial control-dependence computation: on average, the percentage of such control dependences was 5.9%. The percentage of missed control dependences that were computed along unbalanced-left paths ranged from 25.4% for `armenu` to 84.3% for `netmaze`. On average, 35.2% of the missed control dependences were computed along unbalanced-left paths, and were therefore caused because of call relations among the procedures. The remaining 58.9% of the missed control dependences were caused because of the effects of PNRCs: 56.1% were computed along unbalanced-right paths, and 2.8% were computed along unbalanced-right-left paths.

Figure 16 presents data about intraprocedural control dependences computed for the programs. The data illustrate the extent to which intraprocedural control dependences include spurious dependences. The graph on the left in the figure shows the percentages of intraprocedural control dependences that were computed only by the intraprocedural control-dependence computation, and those that were computed also by the interprocedural control-dependence computation. The number at the top of each bar is the total number of intraprocedural control dependences computed for that program. The percentage of spurious control dependences ranged from 9.3% for `dejavu` to 36.5% for `mpegplayer`. On average, 23.7% of the intraprocedural control dependences were spurious; such dependences do not exist when interactions among procedures are considered.

The graph on the right in Figure 16 classifies the spurious control dependences based on a semantic interpretation of those control dependences. A spurious control dependence is computed only by the intraprocedural control-dependence computation. Let s be a statement in procedure P such that s is control dependent on p and the control dependence is spurious. In this control-dependence relation, p is either a

predicate in P or the entry into P. If p is a predicate, the control-dependence relation is clearly spurious and provides misleading information because p does not control the execution of s. However, if p is the entry into P, the control-dependence relation can provide information that is not misleading but incomplete. Suppose that p is the entry into P. Then, the intraprocedural control-dependence relation is equivalent to stating that if control enters procedure P, s is definitely reached. This statement is valid also in the interprocedural context if all interprocedural control dependences for s are computed along only unbalanced-left or unbalanced-right-left paths. In such cases, although external predicates control the execution of s, it is still valid to state that if control enters P, s is definitely reached. Therefore, in such cases, the intraprocedural control-dependence relation does not provide misleading information; it provides incomplete information.

The graph on the right in Figure 16 classifies the intraprocedural control dependences as spurious or incomplete. The graph illustrates that, for `armenu`, `diff`, `flex`, and `mpegplayer`, a considerable percentage of the intraprocedural control dependences are spurious: 85.7% for `armenu`, 73.6% for `diff`, 74.8% for `flex`, and 67.7% for `mpegplayer`. On average, 61.1% of the intraprocedural control dependences are spurious.

6 Related Work

Definitions of control dependence appear frequently in the research literature (e.g., [3, 6, 8, 17, 21, 23]). In most cases (with the exception of the definition in [17], discussed below) these definitions are stated in terms of relationships between nodes in flow graphs that are said to represent "programs." However, these definitions seldom explicitly describe the relationship of these graphs to whole programs. Podgurski and Clarke [23] state that their definition of the control flow graph can represent any procedural program; however, as presented, that definition also applies to a class of ICFGs on which the syntactic–semantic relationship does not hold (see Appendix A for details). Our Definition 12 clarifies the application of Podgurski and Clarke's (and other flow-graph based) definitions of control dependence to the interprocedural setting.

Various algorithms for calculating control dependences exist (e.g., [2, 3, 6, 8, 10, 17]). Some of these algorithms (e.g., [2, 10]) operate on abstract syntax trees for individual procedures, and are therefore strictly intraprocedural. As presented, most other algorithms operate on control flow graphs. We have shown that when such algorithms are applied independently to control flow graphs for individual procedures in program $\mathcal{P}$ without accounting for the context in which those procedures are invoked in $\mathcal{P}$, the algorithms can calculate control dependences in a manner that does not support the syntactic–semantic relationship. Alternatively, given an IIFG, these algorithms can calculate correct control dependences for $\mathcal{P}$ for non-recursive programs; however, the size of the IIFG may be exponential in program size; thus, such an application may be inefficient.

A recent paper by Loyall and Mathisen [17] uses ICFGs to define interprocedural control dependence. The authors define an *interprocedural walk* in an ICFG $\mathcal{G}$ to be a sequence of nodes that represent a realizable path through $\mathcal{G}$. A node $u \in \mathcal{G}$ is said to *postdominate* a node $v \in \mathcal{G}$ if and only if every interprocedural walk from v to the exit node of the ICFG contains u. Control dependence is then defined in a manner similar to that of our Definition 4. However, this definition does not support the syntactic–semantic relationship. To see this, refer again to Figure 1, and consider the version of `Sum` created by substituting the alternative version of line 18, but not substituting the alternative version of line 6: this version contains both calls to `B`, but halts (assuming normal termination) only on reaching statement 8. Consider also the ICFG for this version of `Sum`, not pictured, but easily constructed from the ICFG of Figure 8 by replacing node 18

with its alternative version, and adding an edge from that node to node 19.[12] In this ICFG, because of the unconditional calls to `B` in node 6a and to `C` in node 10a, nodes 10, 11, 14, and 17 occur on every realizable path from both successors of node 4 (6a and 5a). Thus, according to Loyall and Mathisen's definitions, nodes 10, 11, 14, and 17 postdominate both successors of 4, and thus, they are not control dependent on node 4. According to Podgurski and Clarke's definition of semantic dependence, however, nodes 10, 11, 14, and 17 are semantically dependent on node 4, because the condition in node 4 determines (through its control of the call to `B` in 5a) the number of times these statements execute. Thus, in this case, Loyall and Mathisen's definitions of control dependence do not support the syntactic–semantic relationship.

Loyall and Mathisen extend their basic definitions, summarized above, to account for the presence of embedded halts. Their extended definitions utilize an ICFG in which `halt` nodes are connected to a unique ICFG exit node, and (we believe) correctly identify the effects of halts on control dependences (at least in cases where that effect does not interact with the multiple-context effect). However, this extended definition does not circumvent the difficulty described above; thus, the extended definition does not support the syntactic–semantic relationship.

Loyall and Mathisen do not provide an algorithm for calculating interprocedural control dependences between nodes or statements; their goal is to define and calculate control dependence between procedures, and they use their definitions of interprocedural control dependence between nodes to define control dependence between procedures. By their definition, a procedure P_i is control dependent on procedure P_j if and only if there exists some node n_i in the portion of the ICFG associated with P_i, and some node n_j in the portion of the ICFG associated with P_j, such that n_i is control dependent on n_j. Loyall and Mathisen provide an algorithm for calculating procedure-level control dependences without first calculating node-level dependences. Although procedure-level dependence is not our focus in this work, we observe that this algorithm has two drawbacks. First, the procedure-level control dependences calculated by Loyall and Mathisen's algorithm conflict with those identified by their definition. For example, applied to the version of `Sum` that does not contain the embedded halt, Loyall and Mathisen's definitions imply that no nodes in `B` are control dependent on the predicate in `M`, because they all postdominate the successors of that predicate through the second, unconditional call to `B` in node 6a. However, Loyall and Mathisen's algorithm does identify procedure `B` as control dependent on procedure `M`, on the basis of the existence of the conditional call to `B` in `M`. In this case then, and, we believe, in general, their algorithm for calculating procedure-level control dependences does accommodate the multiple-context effect — at least for programs that do not contain embedded halts.

The second drawback of Loyall and Mathisen's algorithm is that it does not accommodate the embedded-halt effect. Thus, the algorithm can incorrectly identify control dependences between procedures for programs that contain halts. For example, in `Sum`, the second call to `B` (from node 6a) is control dependent on predicate node 17 due to the embedded halt at node 18; thus, node 10a in `B` is control dependent on that predicate node. According to Loyall and Mathisen's definition of procedure-level control dependence, `B` is control dependent on `C`; similar reasoning also shows that by their definition, `C` is control dependent on `C`. Loyall and Mathisen's algorithm identifies neither of these procedure-level control dependences.

[12] An ICFG of Loyall and Mathisen's form also merges call and return nodes; however, this does not affect our argument.

7 Conclusions and Future Work

There are three primary contributions of this paper. First, the paper identifies and discusses several ways in which control dependences calculated intraprocedurally do not correctly represent control dependences that exist in programs. Second, the paper presents a precise definition of interprocedural control dependence that supports the syntactic–semantic relationship. Third, the paper presents two approaches for computing interprocedural control dependences: one approach computes precise interprocedural control dependences but may be expensive; the other approach efficiently obtains a conservative estimate of those dependences.

Interprocedural control dependences are useful for applications in testing and maintenance. For example, the partial control dependences computed in the first phase of our algorithm can be used by an interprocedural slicing algorithm to account correctly for interprocedural control dependences in programs that contain embedded halts [9]. For further example, statement-based interprocedural control dependences computed by our algorithm can be used to calculate procedure-level dependences [17], which provide a higher-level view of dependences than statement dependences for use in program comprehension, debugging, and impact analysis.

Our first approach to computing interprocedural control dependences distinguishes each context in which a procedure can be called, and computes distinct control dependences for each context. To compute such control dependences, the approach inlines the called procedure at each call site, and constructs a representation that can be exponential in the size of the program. Our study on the effects of such inlining (see Figure 3) shows that, for some programs, the resulting representation can be excessively large, which can cause our first approach to be impractical. However, for other programs, the representation grows by only a few factors over the program size; for such programs, our first approach may be applicable. Future experiments that study not only the effects of procedure inlining but also evaluate the performance of the traditional control-dependence algorithms [3, 6, 8, 21] on the inlined representations, would help establish the parameters that determine the feasibility and applicability of our first approach.

Our second approach to computing interprocedural control dependences does not distinguish the contexts in which a procedure can be called to compute control dependences efficiently. For applications, such as computation of procedure-level control dependence, this loss of context-specific information causes no imprecision in analysis results. In future work, we intend to investigate the precision that is lost in going from the context-based approach to the statement-based approach, and the effects of the loss of this precision on other analysis techniques.

Embedded halts belong more generally to a class of constructs that cause arbitrary interprocedural transfer of control, which, in practical programs, includes constructs such as exception handling and interprocedural jumps. Our definition of interprocedural control dependence applies to programs that contain such constructs. Our current work includes an investigation of the effects of such constructs on interprocedural control dependence computation and on other analysis techniques [29], with the aim of of generalizing the results presented in this paper to constructs that cause arbitrary interprocedural transfer of control.

We believe that our definitions extend to weak control dependence [23], and thus, can define interprocedural control dependences that preserve the relationship demonstrated in that work between weak syntactic dependence and (possibly non-finitely-demonstrated) semantic dependence. Future work could investigate this extension, and the relationship of these results to generalized control dependence [3].

ACKNOWLEDGMENTS

This work was supported in part by a grant from Microsoft, Inc., by NSF under NYI award CCR-9696157 to Ohio State University, by National Science Foundation Faculty Early Career Development Award CCR-9703108 to Oregon State University, by National Science Foundation Award CCR-9707792 to Ohio State University and Oregon State University, by an Ohio State University Research Foundation Seed Grant, and by funding through the Yamacraw Mission to Georgia Institute of Technology. Sujatha Sathi and Jim Jones helped with the development and implementation of `ComputePartialCD` and `ComputeInterCD`. The anonymous reviewers provided useful feedback that improved the paper.

A Flow Graphs, Interprocedural Control Dependence, and the Syntactic-Semantic Relationship in Reference [23]

Podgurski and Clarke [23] define control-flow graphs as follows:

> A *control-flow graph* G is a directed graph that satisfies each of the following conditions:
>
> 1. The maximum out-degree of the nodes is at most two [this restriction is made for simplicity only].
> 2. G contains two distinguished nodes: the initial node n_e which has in-degree zero, and the final node n_x which has out-degree zero.
> 3. Every node of G occurs on some n_e to n_x walk.[13]

Podgurski and Clarke [23] state that their definition of control flow graph, though somewhat restricted to simplify presentation, "can be used to represent any procedural program ... by employing straightforward representation conventions involving the use of dummy vertices and arcs." However, as stated, their definition of control flow graph fails to exclude a class of ICFGs for which, under the given definitions of postdominance and control dependence, the syntactic–semantic relationship does not hold. Specifically, consider the set of programs that contain no unreachable code, and no procedures that are called more than twice. An ICFG for these programs meets the three conditions for control flow graphs stated above. However, when Podgurski and Clarke's definitions of postdominance and control dependence are applied to the ICFG for our example program, the effect described in Section 3.1 as the multiple-context effect results: nodes 10, 11, 14, and 17, although semantically dependent on node 4, are not control dependent on node 4, because they do not postdominate either successor of node 4.

On closer examination of [23], focusing particularly on the proof of the syntactic–semantic relationship, it is clear that Podgurski and Clarke require additional properties of control flow graphs that are not stated in the definition cited above. The authors define the *context*[14] $CON(v, Wv)$ of a node v with respect to an initial walk Wv in a def/use graph G^{du} to be a directed tree that represents the cumulative flow of data to v along W. We refer the reader to [23] (page 975) for details; however, the idea is that the context of node v

[13] The discussion in this section is drawn directly from Reference [23]. For simplicity of reproducing that discussion, we retain the use of the term "walk" in that discussion to refer to a path.

[14] Podgurski and Clarke use the term "context" in a different sense than we do. Our usage pertains to the sequence of procedure calls that lead up to a particular procedure call. However, to avoid unnecessary complications in presenting their discussion, we retain their usage of the term.

on walk Wv is similar to the set of symbolic values held by variables in the use set U of the node along walk Wv. Next, the authors define a *hyperwalk* to be either an ordinary walk on graph G, or an infinite walk. A hyperwalk is *consistent* "if there are no two occurrences of a decision node d in W that have the same context but are followed by different successors of d." Because the context of a node determines the values of the variables in the use set of that node when a walk to that node is executed, that context determines the branch taken at that node; thus, an executable hyperwalk must be consistent. Finally, the authors define a pair of hyperwalks as *reciprocally v-consistent* if (informally), the fact that the hyperwalks diverge at a pair of nodes implies that either their contexts differ at those nodes, or the node v whose interpretation has changed causes the difference.

The notions of consistency and reciprocal v-consistency are central to Podgurski and Clarke's proof of the syntactic–semantic relationship. However, ICFGs do not properly support these notions. In ICFGs, exit nodes, like predicate nodes, may have multiple successors. The direction of control flow from such nodes is not, however, determined solely by the context of those nodes; instead, it is determined by the identity of the call node from which the exiting procedure was invoked. An exit node in an ICFG may occur twice in a hyperwalk, both times with the same context, but in each case followed by a different successor. Thus, an executable hyperwalk on an ICFG need not be consistent; and thus, Podgurski and Clarke's proof does not apply to ICFGs. In contrast, the IIFG, in which procedures are inlined, does support the notions of consistency and reciprocal v-consistency, because it explicitly depicts control flow from exit nodes to their successors, and is otherwise identical to the flow graphs defined by Podgurski and Clarke. The fact that IIFGs possess the properties of graphs that allow Podgurski and Clarke to prove Theorem 1 implies that those proofs apply also to IIFGs.

To state that our definition of the IIFG and of interprocedural control dependence corrects deficiencies in Podgurski and Clarke's is unduly strong; their definition must be intended to exclude ICFGs, and the natural extension of their graphs to the interprocedural context is to the IIFG. Thus, it is more appropriate to say that our definitions clarify, rather than correct, Podgurski and Clarke's definitions of control dependence, and the application of those definitions to interprocedural control dependence.

B Proof of Correctness of our Algorithm for Computing Statement-Based Interprocedural Control Dependences

Our algorithm computes statement-based interprocedural control dependences by summarizing for each statement, the control dependences that exist in different contexts for that statement in the IIFG. To demonstrate the correctness of our algorithm, we show that our algorithm computes the same statement-based control dependences as are computed by an alternative approach that constructs an IIFG, applies a traditional algorithm for control-dependence computation to the IIFG, and summarizes the control dependences for each statement using the *NodeSet* relations. We present here only an outline of the proof; further details of the proof can be found in Reference [30].

The overall structure of the proof is as follows. First, we classify paths in the IIFG. Next, we characterize paths in the ICDG that are traversed by the algorithm. Finally, we show by cases that (1) if a control-dependence relation occurs along a certain type of path in the IIFG, there exists a corresponding path in the ICDG that is traversed by the algorithm, and (2) if the algorithm traverses a certain type of path in the

ICDG, there exists a control-dependence relation along a corresponding type of path in the IIFG.

We classify IIFG paths based on sequences of call and return edges that appear in the paths; previous work [18] defined such paths in the ICFG. A path in the IIFG is a *same-level path* if each call edge in the path is matched with a return edge. A same-level path represents an execution sequence that begins and ends in the same CFG; the depth of the call stack is the same at the beginning and the end of such a path. A path is an *unbalanced-left path* if it contains at least one call edge that is not matched by a return edge. An unbalanced-left path represents an execution sequence in which some procedure calls have not completed; the call stack is deeper at the end of such a path than at the beginning. A path is an *unbalanced-right path* if it contains at least one return edge that is not preceded by a matching call edge. An unbalanced-right path represents an execution sequence in which a procedure call completes such that the sequence that led to the invocation of that procedure is not part of the path; the call stack is thus shallower at the end of an unbalanced-right path than at the beginning. Finally, a path is an *unbalanced-right-left path* if it contains an unbalanced-right subpath followed by an unbalanced-left subpath.

It follows from the definition of an IIFG that each path in an IIFG is a same-level path, an unbalanced-left path, an unbalanced-right path, or an unbalanced-right-left path. Moreover, as the following lemma shows, each path between two nodes in an IIFG is of the same type.

> **Lemma 1.** Let $\mathcal{G}^I$ be an IIFG, and let u and v be nodes in $\mathcal{G}^I$. Let $\psi_{u \xrightarrow{+} v}$ be the set of paths from u to v. Then, each $\psi \in \psi_{u \xrightarrow{+} v}$ is of the same type.

Proof. The proof considers each of the four types of paths that any $\psi \in \psi_{u \xrightarrow{+} v}$ can be, and shows that all other paths in $\psi_{u \xrightarrow{+} v}$ must also be of that type. The proof uses the properties of the IIFG that (1) in an IIFG, a different copy of the CFG is inlined at each call site, and (2) an IIFG contains no interprocedural cycles that are caused by recursion [30]. □

Next, we characterize paths in the ICDG that are traversed by `ComputeInterCD`. A *placeholder segment* in an ICDG is a path (X, P, N), where edge (X, P) is a call or a return edge, and edge (P, N) is a placeholder control-dependence edge. An *entry placeholder segment* is a placeholder segment in which (X, P) is a call edge and P is an entry placeholder. A *return placeholder segment* is a placeholder segment in which (X, P) is a return edge and P is a return placeholder. A *control-dependence path* in an ICDG is a path $\Psi = (P, X_1) \cdot PS_1 \cdot PS_2 \cdot \ldots \cdot PS_m$,[15] $m \geq 1$, where $PS_i = (X_i, P_i, N_i)$, $1 \leq i \leq m$, is a placeholder segment, and edge (P, X_1) is a predicate control-dependence edge.

A control-dependence path is composed of a control-dependence edge followed by one or more placeholder segments. Figure 17 illustrates a control-dependence path in the ICDG for `Sum`; the path consists of two return placeholder segments: (19, 10b, 15) and (15, 6b, 8). Like paths in the IIFG, we classify control-dependence paths according to calls and returns that appear along the paths. An *unbalanced-left control-dependence path* contains one or more unmatched call edges; each placeholder segment in such a path is an entry placeholder segment. An *unbalanced-right control-dependence path* contains one or more unmatched return edges; each placeholder segment in such a path is a return placeholder segment. An *unbalanced-right-left control-dependence path* is an unbalanced-right control-dependence path followed by one or more entry placeholder segments. For example, the path shown in Figure 17 is an unbalanced-right control-dependence

[15] The notation $\psi_1 \cdot \psi_2$ represents a concatenation of paths ψ_1 and ψ_2, where the last node in path ψ_1 is the same as the first node in path ψ_2.

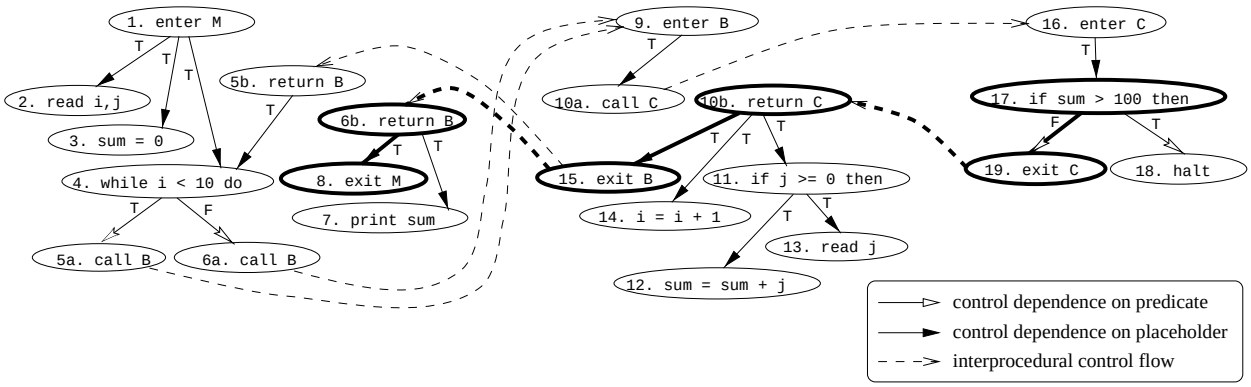

Figure 17: An unbalanced-right control-dependence path in the ICDG for `Sum`; the path consists of two return placeholder segments: (19, 10b, 15) and (15, 6b, 8).

---

path.

The following lemma shows that `ComputeInterCD` traverses all and only control-dependence paths in the ICDG.

> **Lemma 2.** Let $\mathcal{G}^D$ be an ICDG. There exists a control-dependence path Ψ in $\mathcal{G}^D$ if and only if `ComputeInterCD` traverses Ψ.

Proof. ($\Longrightarrow$) First, the proof states that (1) each control-dependence path is incident only on a node that is control dependent on a placeholder, and (2) `ComputeInterCD` processes each node that is control dependent on a placeholder. Next, the proof shows that, while processing a node that is control dependent on a placeholder, `ComputeInterCD` traverses each control-dependence path incident on that node.

($\Longleftarrow$) Each path traversed by `ComputeInterCD` starts at a node that is control dependent on a placeholder. Depending on whether the node is control dependent of an entry or a return placeholder, the proof shows—using the properties of a control-dependence path—that the path traversed by `ComputeInterCD` must be an unbalanced-left, an unbalanced-right, or an unbalanced-right-left path [30]. □

The next lemma shows that a postdominance relation between two nodes that belong to the same CFG in an IIFG is preserved in the corresponding ACFG.

> **Lemma 3.** Let $\mathcal{G}^I$ be an IIFG and let u and v be nodes in CFG G_i in $\mathcal{G}^I$. Let G^A be the ACFG that corresponds to G_i, and let U and V be the nodes in G^A that correspond to u and v, respectively. U postdominates V if and only if u postdominates v.

Proof. ($\Longleftarrow$) Suppose that u postdominates v. Show that U postdominates V. The proof uses contraposition: it assumes that U does not postdominate V, and shows that this causes u to not postdominate v. If U does not postdominate V, there exists a V–N_{sx} path $\Psi = (V, N_1, N_2, \ldots, N_j, N_{sx})$ in G^A such that U does not appear in the path. The proof considers two cases for Ψ—whether Ψ contains a node that represents a call site—and shows that, in each case, there exists a path in $\mathcal{G}^I$ from v to the exit node of $\mathcal{G}^I$ that does not contain u [30].

($\Longrightarrow$) The proof again shows that the contrapositive of the implication is true: it assumes that u does not postdominate v, and shows that this causes U to not postdominate V. Because u does not postdominate

v, there exists a path ψ from v to the exit node of $\mathcal{G}^I$ that does not contain u. The proof considers three cases for ψ—(1) ψ contains no call site, (2) ψ contains a definitely returning call site, or (3) ψ contains a PNRC—and shows that, in each case, there exists a V–N_{sx} path Ψ in G^A that does not contain U [30]. □

Lemma 4. Let $\mathcal{G}^I$ be the IIFG for program $\mathcal{P}$. Let u and v be nodes in $\mathcal{G}^I$. Let $\psi_{v \xrightarrow{+} u}$ be the set of paths from v to u such that each path $\psi \in \psi_{v \xrightarrow{+} u}$ is a same-level path. Let $\mathcal{G}^D$ be the ICDG for $\mathcal{P}$. Let U and V be the nodes in $\mathcal{G}^D$ that correspond to u and v, respectively. Then, u is control dependent on v if and only if there exists an edge from V to U in $\mathcal{G}^D$.

Proof. Because each ψ is a same-level path, u and v belong in the same CFG G_i in $\mathcal{G}^I$. Let G^A be the ACFG for G_i. Then, according to Lemma 3, the postdominance relation between any two nodes in G_i is equivalent to a postdominance relation between the corresponding nodes in G^A. Moreover, as a consequence of Lemma 3, a node belonging to G_i does not postdominate another node belonging to G_i if and only if the corresponding nodes in G^A have the same relation. Then, it is easy to show that u is control dependent on v if and only if U is control dependent on V (or equivalently, there exists an edge from V to U in $\mathcal{G}^D$) [30]. □

Lemma 5. Let $\mathcal{G}^I$ be the IIFG for program $\mathcal{P}$. Let u and v be nodes in $\mathcal{G}^I$. Let $\psi_{v \xrightarrow{+} u}$ be the set of paths from v to u. Let $\mathcal{G}^D$ be the ICDG for $\mathcal{P}$. Let U and V be the nodes in $\mathcal{G}^D$ that correspond to u and v, respectively.

(1) Let each path $\psi \in \psi_{v \xrightarrow{+} u}$ be an unbalanced-right path. Then, u is control dependent on v if and only if there exists an unbalanced-right control-dependence path Ψ in $\mathcal{G}^D$ from V to U.

(2) Let each path $\psi \in \psi_{v \xrightarrow{+} u}$ be an unbalanced-left path. Then, u is control dependent on v if and only if there exists an unbalanced-left control-dependence path Ψ in $\mathcal{G}^D$ from V to U.

(3) Let each path $\psi \in \psi_{v \xrightarrow{+} u}$ be an unbalanced-right-left path. Then, u is control dependent on v if and only if there exists an unbalanced-right-left control-dependence path Ψ in $\mathcal{G}^D$ from V to U.

Proof. (1) ($\Longrightarrow$) Suppose that u is control dependent on v. Show that there exists path Ψ in $\mathcal{G}^D$. The proof shows, by induction on the number of unmatched returns in ψ, that there exists a corresponding path Ψ in $\mathcal{G}^D$.

For brevity, we outline only the basis step of the proof.

Basis step. Each $\psi \in \psi_{v \xrightarrow{+} u}$ contains a single unmatched return. Let x and r be the exit node and the return node, in G_v and G_u respectively, that are the source and the target of the unmatched return edge. Let X and R be the corresponding ICDG nodes.

The proof for the basis step proceeds as follows. (1) First, the proof shows that x is control dependent on v. Then, because X and V belong in the same ACFG, according to Lemma 4, X is control dependent on V. Thus, $\mathcal{G}^D$ contains an edge (V, X). (2) Next, the proof shows that there exists an edge (R, U) in $\mathcal{G}^D$; this follows from Lemma 3 and the construction of the ACFG. (3) Finally, the proof shows that there exists a return edge (X, R) in $\mathcal{G}^D$. Then, concatenating edges (V, X), (X, R), and (R, U) yields the unbalanced-right control-dependence path Ψ.

In the inductive hypothesis, the proof assumes that if ψ contains k unmatched returns, there exists a corresponding unbalanced-right control-dependence path in $\mathcal{G}^D$ from V to U. Finally, in the inductive step,

the proof shows that if the number of unmatched returns increases by one, there still exists a corresponding unbalanced-right control-dependence path in $\mathcal{G}^D$ [30].

(1) ($\Longleftarrow$) Suppose that there exists an unbalanced-right control-dependence path Ψ from node V to node U in $\mathcal{G}^D$. Show that u is control dependent on v in $\mathcal{G}^I$.

In this case, the proof uses induction on the number of return placeholder segments in Ψ [30].

(2), (3) The proof proceeds in a similar manner; see Reference [30] for details. □

The proof of Theorem 3 follows directly from the preceding lemmas. For a given control-dependence relation, u control dependent on v, in the IIFG, Lemma 1 establishes the types that the paths from v to u can be. The proof considers each of these types, and shows that, in each case, there exists either a corresponding control-dependence edge (Lemma 4) or a corresponding control-dependence path (Lemma 5) in the ICDG, and that the algorithm traverses this edge or path [30].

For a given control-dependence relation, U control dependent on V, computed by the algorithm, either the relation is computed by `ComputePartialCD` (and `ComputeInterCD` receives that relation as an input) or by `ComputeInterCD` along a control-dependence path (Lemma 2). Then, using the results of Lemmas 4 and 5, the proof shows that there must exist a corresponding control-dependence relation in the IIFG [30].

References

[1] D. C. Atkinson and W. G. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, March 1996.

[2] R. Ballance and B. Maccabe. Program dependence graphs for the rest of us. Technical report, University of New Mexico, November 1992.

[3] G. Bilardi and K. Pingali. A framework for generalized control dependence. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 291–300, May 1996.

[4] D. Binkley. Using semantic differencing to reduce the cost of regression testing. *Proceedings of the 1992 Conference on Software Maintenance*, pages 41–50, November 1992.

[5] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):450–90, October 1991.

[7] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[8] 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, July 1987.

[9] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, April 1998.

[10] M. J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, May 1996.

[11] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, March 1997.

[12] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 11–20, March 1998.

[13] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[14] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, pages 191–200, May 1994.

[16] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[17] J. P. Loyall and S. A. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the Conference on Software Maintenance*, pages 282–291, September 1993.

[18] D. Melski and T. Reps. Interprocedural path profiling. Technical Report TR–1382, Computer Sciences Department, University of Wisconsin, September 1998.

[19] G. C. Murphy and D. Notkin. Lightweight lexical source model extraction. *ACM Transactions on Programming Languages and Systems*, 5(3):262–292, July 1996.

[20] H. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations in C programs. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[21] K. Pingali and G. Bilardi. Optimal control dependence computation and the Roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[22] A. Podgurski. *The significance of program dependences for software testing, debugging, and maintenance.* PhD thesis, University of Massachusetts, 1989.

[23] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[24] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, December 1989.

[25] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[26] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[27] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[28] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[29] 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*, pages 432–441, May 1999.

[30] S. Sinha, M. J. Harrold, and G. Rothermel. Interprocedural control dependence. Technical Report GIT-CC-00-17, College of Computing, Georgia Institute of Technology, June 2000.