A New Foundation for Control Dependence and Slicing for Modern Program Structures

VENKATESH PRASAD RANGANATH, TORBEN AMTOFT, ANINDYA BANERJEE, and JOHN HATCLIFF
Kansas State University
and
MATTHEW B. DWYER
University of Nebraska

The notion of control dependence underlies many program analysis and transformation techniques. Despite being widely used, existing definitions and approaches to calculating control dependence are difficult to apply directly to modern program structures because these make substantial use of exception processing and increasingly support reactive systems designed to run indefinitely.

This article revisits foundational issues surrounding control dependence, and develops definitions and algorithms for computing several variations of control dependence that can be directly applied to modern program structures. To provide a foundation for slicing reactive systems, the article proposes a notion of slicing correctness based on weak bisimulation, and proves that some of these new definitions of control dependence generate slices that conform to this notion of correctness. This new framework of control dependence definitions, with corresponding correctness results, is even able to support programs with irreducible control flow graphs. Finally, a variety of properties show that the new definitions conservatively extend classic definitions. These new definitions and algorithms form the basis of the Indus Java slicer, a publicly available program slicer that has been implemented for full Java.

This work was supported in part by the US Army Research Office under DAAD190110564, by the Air Force Office of Scientific Research under Contract FA9550-006-1-0223, by the DARPA/IXO PCES Program under AFRL Contract F33615-00-C-3044, by the NSF under CCR-0306607, CCR-0296182, CCR-0209205, ITR-0326577, and CCR-0444167, by Lockheed Martin, and by the Intel Corporation.
Part of this work was published in the 2005 *Proceedings of the European Symposium on Programming (ESOP)*, Lecture Notes in Computer Science, vol. 3444, Springer.
Authors' addresses: V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, Department of Computing and Information Sciences, Kansas State University, 234 Nichols Hall, Manhattan, KS 66506; email: {rvprasad,tamtoft,ab,hatcliff}@cis.ksu.edu; M. B. Dwyer, Department of Computer Science and Engineering, University of Nebraska, 256 Avery Hall, Lincoln, NE 68588-0115; email: dwyer@cse.unl.edu.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Nontermination, control dependence, order dependence, program slicing, bisimulation, Indus

ACM Reference Format:

Ranganath, V. P., Amtoft, T., Banerjee, A., Hatcliff, J., and Dwyer, M. B. 2007. A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst. 29, 5, Article 27 (August 2007), 43 pages. DOI = 10.1145/1275497.1275502 http://doi.acm.org/10.1145/1275497.1275502

1. INTRODUCTION

The notion of control dependence underlies many program analysis and transformation techniques that are used in numerous applications, including program slicing applied for program understanding [Podgurski and Clarke 1990], debugging [Francel and Rugaber 1999], partial evaluation [Andersen 1994], and compiler optimizations such as global scheduling, loop fusion, and code motion [Ferrante et al. 1987]. Intuitively, a program statement n_1 is control dependent on a statement n_2 if n_2 (typically, a conditional statement) controls whether n_1 will be executed or bypassed during an execution of the program.

While existing definitions and approaches to calculating control dependence and slicing are widely applied (as cited previously) and have been used for well over 20 years, there are several aspects of these definitions and associated notions of correctness that prevent them from being applied cost effectively to modern program structures, which rely significantly on exception processing and increasingly support reactive systems that are designed to run indefinitely.

Classic Definitions of Control Dependence. These are stated in terms of program control flow graphs (CFGs) in which the CFG has a unique end node; they do not apply directly to program CFGs with: (a) multiple end nodes; or (b) no end node. The first restriction implies that existing definitions cannot be applied directly to programs/methods with multiple exit points—a restriction that would be violated by any method that raises exceptions or includes multiple returns. Similarly, and probably more damaging for practical applications, restriction (b) implies that existing definitions cannot be applied directly to reactive programs or to system models with control loops that are designed to run indefinitely.

Restriction (a) is usually addressed by performing a preprocessing step that transforms a CFG with multiple end nodes into a CFG with a single end node by adding a new designated end node to the CFG and inserting arcs from all original exit states to the new end node [Hatcliff et al. 1999; Podgurski and Clarke 1990]. Such a transformation actually has some benefits, like providing a single node that contains the final values of global variables. Restriction (b) can also be addressed in a similar fashion by, for example, selecting a single node within the CFG to represent the end node. This case is significantly more problematic than the preprocessing for restriction (a) because the criteria for selecting end nodes that lead to the desired control dependence relation between program nodes is often unclear (as illustrated in Section 3.2). This is particularly true in threads such as event handlers, which have no explicit shut-down methods, but are "shut down" by killing the thread (thus, there is no explicit exit point in the thread's control flow).

Existing Definitions of Slicing Correctness. These either apply to programs with terminating execution traces, or often fail to state whether the slicing transformation preserves the termination behavior of the program being sliced. Thus, these definitions cannot be applied to reactive programs that are designed to execute indefinitely. Such programs are used in numerous modern applications such as event-processing modules in GUI systems, web services, and real-time systems with autonomous components (e.g., data sensors, etc.).

Despite these difficulties, it appears that researchers and practitioners do continue to apply slicing transformations to programs that fail to satisfy the aformentioned restrictions. However, in reality the preprocessing transformations related to the first issue introduce extra overhead into the transformation pipeline, clutter up program transformation and visualization facilities, necessitate the use/maintenance of mappings from the transformed CFGs back to the original CFGs, and introduce extraneous structure with ad hoc justifications that all downstream tools/transformations must interpret and build on in a consistent manner. Moreover, regarding the second issue, it will be infeasible to continue to ignore problems with termination, since slicing is increasingly applied in high-assurance applications such as reducing models for verification [Hatcliff et al. 2000] and for reasoning about security issues where it is crucial that liveness/nontermination properties be preserved.

Working on a larger project concerning slicing concurrent Java programs, we have found it necessary to revisit basic issues surrounding control dependence, and have sought to develop definitions that can be directly applied to modern program structures such as those found in reactive systems. In this article, we propose and justify the usefulness and correctness of simple definitions of control-based dependence that overcome the problematic aspects of the classic definitions described earlier. The specific contributions of this work are as follows.

—After reviewing and assessing classic definitions of control dependence in Sections 2 and 3, we propose new definitions that are simple to state, easy to calculate, and that apply directly to control flow graphs that may have no or nonunique end nodes, thus avoiding troublesome preprocessing CFG transformations. We formalize these definitions and also supplement the formalization by providing equivalent definitions in computation tree logic (CTL) (Section 4).
—To also enable slicing based on these new general definitions to preserve semantics (particularly reduction order) for CFGs (with or without unique end nodes) that are irreducible, we propose a new kind of control-based

dependence that captures the control-flow-imposed ordering relationships between a CFG's nodes (Section 4.4).

—We clarify the relationship between our new definitions and classic ones by showing that ours represent a form of "conservative extension" of classic definitions: When applied to CFGs that conform to the restriction of a single end node, our new definitions correspond to classic ones; they neither introduce additional dependences nor omit any (Section 4.5).

—We prove that some of the proposed definitions when applied to CFGs, yield slices that are correct according to generalized notions of slicing correctness based on a form of weak bisimulation that is appropriate for programs with infinite execution traces (Section 5.1).

—We provide polynomial-time algorithms to calculate control and order dependences according to the proposed definitions (Section 6).

Although we have developed our new control dependence definitions in the context of slicing, they are applicable to other domains. For example, they also seem useful for calculating control dependences for state machines, which often: (a) do not conform to the unique end node restriction; and (b) have irreducible control flow graphs.

The notions of control dependence proposed in this article have been implemented in our Java slicer that is publicly available as part of project Indus [SAnToS Laboratory 2007]. Our Java slicer can handle almost[1] all features of Java 1.4. We have successfully applied the slicer to Java applications constituting up to 10,000 lines of code. The slicer is also used by Kaveri [Jayaraman et al. 2004], a plugin that contributes Java program slicing features to the Eclipse platform. Besides the slicer's application as a stand-alone program understanding, debugging, and code transformation tool, it is being used in the next generation of our Bandera [Corbett et al. 2000] tool set for model-checking concurrent Java systems.

In this foundational work, we shall address only *intraprocedural* control dependences, and also assume that all data resides in variables. Although we addressed a number of challenging issues, including heap-allocated data, exceptions, and concurrency, while exploring the proposed theory in Indus, in this work, we focus closely on foundational issues of intraprocedural control dependences in a setting that excludes concurrency, exceptions, and heap-allocated data. Extending our theory to formally justify the slicing of programs in richer settings (as described earlier and as done by our tools (compare with the preceding)) is an exciting topic for future work, but outside the scope of this article.

Extensions to the Conference Version. This document extends its ESOP'05 predecessor [Ranganath et al. 2005] with a new notion of control-based dependence that is necessary to correctly handle irreducible CFGs with no end nodes. A detailed correctness proof for slicing both reducible and irreducible CFGs is presented; it relies on the proposed notion of dependences. We also

[1]With the exception of handling reflection, native methods, and dynamic class loading.

provide algorithms to calculate four forms of control-based dependences, along with their proof of correctness.

2. STANDARD DEFINITIONS

2.1 Control Flow Graphs

When dealing with foundational issues of control dependence, researchers often cast their work in terms of a simple imperative language phrased in terms of control flow graphs. We follow that practice here and base our presentation on a definition of the control flow graph adapted from Ball and Horwitz [1993].

Definition 1 (*Control Flow Graphs*). A control flow graph $G = (N, E, n_0)$ is a labeled directed graph in which:

—N is a set of nodes that represent statements in a program;
—N is partitioned into two subsets N^S and N^P, where N^S are *statement nodes* with each $n_s \in N^S$ having at most one successor, and where N^P are *predicate nodes* with each $n_p \in N^P$ having two distinct successors;
—$N^E \subseteq N^S$ denotes the nodes in N^S that have no successors, namely, the end nodes of G;
—E is a set of labeled edges that represent the control flow between graph nodes; and
—the start node n_0 has no incoming edges, and all nodes in N are reachable from n_0.

If N^E contains exactly one element, and this element is reachable from all other nodes of G, we say that G satisfies the *unique end node property*.

As stated earlier, existing presentations of slicing require that each CFG G satisfies the unique end node property. We shall present alternative definitions that do not rely on that property, but which are equivalent to existing definitions for CFGs that *do* have a unique end node.

It is easy to see how to translate a procedure (method) body into a CFG. To relate a CFG with the program that it represents, we use the function *code* to map a CFG node n to the code for the program statement that corresponds to that node. The function *def* maps each node to the set of variables defined (i.e., assigned to) at that node (always a singleton or empty set), and *ref* maps each node to the set of variables referenced at that node. Specifically, an assignment statement $x := E$ is represented as a node $n_s \in N^S$ with $code(n_s) = (x := E)$, $def(n_s) = \{x\}$, and $ref(n_s) = fv(E)$ (the free variables of E). A `goto` statement (or `break` statement) is represented as a node $n_s \in N^S$ with $def(n_s) = ref(n_s) = \emptyset$, and a branching statement is represented as a node $n_p \in N^P$ with $def(n_s) = \emptyset$, but $ref(n_s) \neq \emptyset$.

A CFG *path* π from n_i to n_k is a sequence of nodes $n_i, n_{i+1}, \ldots, n_k$ such that for every consecutive pair of nodes (n_j, n_{j+1}) in the path, there is an edge from n_j to n_{j+1}. A path between nodes n_i and n_k can also be denoted as $[n_i..n_k]$. When the meaning is clear from the context, we will use π to denote the set of nodes contained in π and we write $n \in \pi$ when n occurs in the sequence π. Path π is *nontrivial* if it contains at least two nodes. A path is *maximal* if it is infinite or terminates in an end node.

The following definitions describe relationships between graph nodes, and are expressed using the distinguished start node, or (if such exists) the distinguished end node [Muchnick 1997]. Node n *dominates* node m in G (written $dom(n, m)$) if every path from the start node s to m passes through n (note that this makes the dominates relation reflexive). For a CFG G with unique end node n_e, node n *postdominates* node m in G (written $post\text{-}dom(n, m)$) if every path from node m to n_e passes through n. Node n *strictly postdominates* node m in G if $post\text{-}dom(n, m)$ and $n \neq m$. Node n is the *immediate postdominator* of node m if $n \neq m$ and n is the first postdominator on every path from m to the end node n_e; it is easy to see that all nodes but the end node have a (necessarily unique) immediate postdominator. Node n *strongly postdominates* node m in G if n postdominates m and there is an integer $k \geq 1$ such that every path from node m of length $\geq k$ passes through n [Podgurski and Clarke 1990]. The difference between strong postdomination and the earlier simple definition of postdomination is that even though node n occurs on every path from m to n_e (and thus n postdominates m), it may be the case that n does not strongly postdominate m due to a loop in the CFG between m and n that admits an infinite path beginning at m and not containing n. Hence, strong postdomination is sensitive to the possibility of nontermination along paths from m to n.

A CFG G of the form (N, E, n_0) is *reducible* if E can be partitioned into disjoint sets E_f (the *forward* edge set) and E_b (the *back* edge set) such that (N, E_f) forms a DAG in which each node can be reached from the start node n_0 and for all edges $e \in E_b$, the target of e dominates the source of e. All "well-structured" programs, including those of Java, give rise to reducible control flow graphs. A CFG that is not reducible is referred to as an *irreducible* CFG. The Java virtual machine bytecode language allows for the construction of programs whose corresponding control flow graphs are irreducible. In this article, we shall present definitions and correctness results that apply to both reducible and irreducible control flow graphs.

2.2 Program Execution

The execution semantics of program CFGs is phrased in terms of transitions on program states (n, σ), where n is a CFG node and σ is a store mapping the corresponding program's variables to values. A series of transitions gives an *execution trace* through p's statement-level control flow graph. It is important to note that when execution is in state (n_i, σ_i), the code at node n_i has not yet been executed. Intuitively, the code at n_i is executed on the transition from (n_i, σ_i) to successor state (n_{i+1}, σ_{i+1}). Execution begins at the start node (n_0, σ_o), and the execution of each node possibly updates the store and transfers control to an appropriate successor node. Execution of a node $n_e \in N^E$ produces a final state (halt, σ) where the control point is indicated by a special label halt—this indicates a normal termination of program execution. The presentation of slicing in Section 5 involves arbitrary finite and infinite nonempty sequences of states written $\Pi = s_1, s_2, \ldots$.

2.3 Notions of Dependence and Slicing

A *program slice* consists of the parts of a program p that (potentially) affect the variable values that are referenced at some program points of interest [Tip 1995]. Traditionally, the "program points of interest" are called the *slicing criterion*. A slicing criterion C for a program p is a nonempty set of nodes $\{n_1, \ldots, n_k\}$, where each n_i is a node in p's CFG.

The definitions to follow are the classic ones of the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [Tip 1995].

Data dependence captures the notion that a variable reference is dependent upon any variable definition that "reaches" the reference.

Definition 2 (*Data Dependence*). Node n is *data dependent* on m in program p (written $m \stackrel{dd}{\rightarrow} n$ with the arrow pointing in the direction of data flow) if there is a variable v such that:

(1) $v \in \mathit{def}(m) \cap \mathit{ref}(n)$; and
(2) there exists a nontrivial path π in p's CFG from m to n such that for every node $m' \in \pi - \{m, n\}$, $v \notin \mathit{def}(m')$.

Control dependence information identifies the conditionals that may affect execution of a node in the slice. Intuitively, node n is control dependent on a predicate node m if m directly determines whether n is executed or "bypassed."

Definition 3 (*Standard Control-Dependence*). Node n is *control dependent* on m in program p (written $m \stackrel{cd}{\rightarrow} n$) if:

(1) There exists a nontrivial path π from m to n in p's CFG such that every node $m' \in \pi - \{m, n\}$ is postdominated by n; and
(2) m is not strictly postdominated by n.

For a node n to be control dependent on predicate m, there must be two paths that connect m with the unique end node n_e such that one contains n and the other does not. There are a number of slightly different notions of control dependence appearing in the literature, and we will consider several of these variants as well as relations between them in the rest of the article. Here we simply note that the preceding definition is standard and widely used (e.g., see Muchnick [1997]).

We write $m \stackrel{d}{\rightarrow} n$ when either $m \stackrel{dd}{\rightarrow} n$ or $m \stackrel{cd}{\rightarrow} n$. The algorithm for constructing a program slice proceeds by finding the set of CFG nodes S_C (called the *slice set*) from which the nodes in C are reachable via $\stackrel{d}{\rightarrow}$.

Definition 4 (*Slice Set*). Let C be a slicing criterion for program p. Then the slice set S_C of p with respect to C is defined as follows:

$$S_C = \{m \mid \exists n \,.\, n \in C \text{ and } m \stackrel{d}{\rightarrow}^* n\}$$

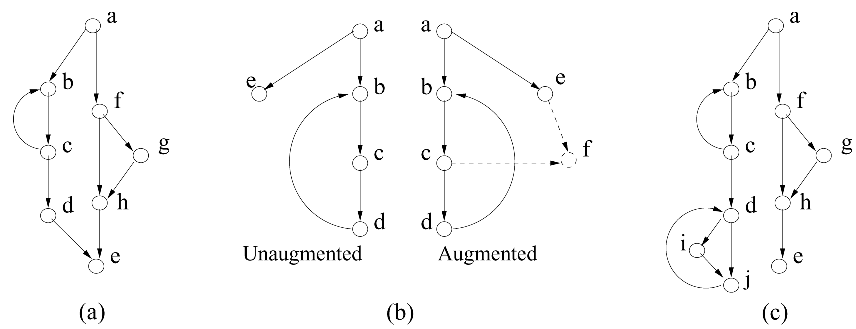


Fig. 1. (a) A simple CFG; (b) illustration of how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have such; (c) a CFG with multiple control sinks of different sorts.

The aforementioned notion of slicing is referred to as "backward static slicing" because the algorithm starts at the criterion nodes and looks backward through the program's control flow graph to find other program statements that influence the execution at the criterion nodes. In this article we consider only backward slices, but our definitions of control dependence can be applied when computing forward slices as well.

In many cases in the slicing literature, the desired correspondence between source program and slice is not formalized because the emphasis is often on applications rather than foundations, and this also leads to subtle differences between presentations. When a notion of "correct slice" is given, it is often stated using the notion of *projection* [Weiser 1984]. Informally, given an arbitrary trace Π of p and an analogous trace Π_s of p_s, one will say that p_s is a correct slice of p if projecting out the nodes in criterion C (and the variables referenced at those nodes) for both Π and Π_s yields identical state sequences. We will consider slicing correctness requirements in greater detail in Section 5.1.

3. ASSESSMENT OF EXISTING DEFINITIONS

3.1 Variations in Existing Control Dependence Definitions

Although the definition of control dependence that we stated in Section 2 is widely used, there are a number of (sometimes subtle) variations appearing in the literature. One dimension of variation is whether the particular definition captures only *direct* control dependence, or also admits *indirect* control dependences. For example, consider the CFG in Figure 1(a): Using the characterization of control dependence in Definition 3, we can conclude that $a \stackrel{cd}{\rightarrow} f$ and $f \stackrel{cd}{\rightarrow} g$, but $a \stackrel{cd}{\rightarrow} g$ does not hold because g does not postdominate f. The fact that a and g are indirectly related (a does play a role in determining whether g is executed or bypassed) is not captured in the definition of control dependence itself, but in the transitive closure used in the slice-set construction (Definition 4). However, as we will illustrate later, some definitions of control dependence [Podgurski and Clarke 1990] directly incorporate this notion of transitivity.

Another dimension of variation is whether the particular definition is sensitive to nontermination. Consider again Figure 1(a), where node c represents a posttest that controls a loop which may be infinite (one cannot tell by simply looking at the CFG). According to Definition 3, $a \stackrel{cd}{\rightarrow} d$ holds but $c \stackrel{cd}{\rightarrow} d$ does not (because d postdominates c), even though c may determine whether d executes or never gets to (due to an infinite loop that postpones d forever). Thus, Definition 3 is *non-termination insensitive*.

We now further illustrate these dimensions by recalling the definitions of strong and weak control dependence given by Podgurski and Clarke [1990] and used in numerous efforts, including the study of control dependence by Bilardi and Pingali [1996].

Definition 5 (*Podgurski-Clarke Strong Control Dependence*). n_2 is *strongly control dependent* on n_1 ($n_1 \stackrel{scd}{\rightarrow} n_2$) if there is a path[2] from n_1 to n_2 that does not contain the immediate postdominator of n_1.

The notion of strong control dependence is almost identical to control dependence in Definition 3, except that strong control dependence is indirect whereas the control dependence in Definition 3 is direct. For example, in Figure 1(a), in contrast to Definition 3, we have $a \stackrel{scd}{\rightarrow} g$ because there is a path afg which does not contain e, the immediate postdominator of a. However, given the directness difference between these variants, it is not surprising that when used in the context of Definition 4 (which computes the transitive closure of dependences), the two definitions give rise to the same slices.[3]

Definition 6 (*Podgurski-Clarke Weak Control Dependence*). n_2 is *weakly control dependent* on n_1 ($n_1 \stackrel{wcd}{\rightarrow} n_2$) if n_2 strongly postdominates n_1', which is a successor of n_1, but does not strongly postdominate n_1'', another successor of n_1.

The notion of weak control dependence captures dependences between nodes induced by nontermination, hence, is nontermination sensitive. Note that in Figure 1(a), $c \stackrel{wcd}{\rightarrow} d$ because d is a successor of c and strongly postdominates itself, and d does not strongly postdominate b: The presence of the loop controlled by c guarantees that there does not exist a k such that every path from node b of length $\geq k$ passes through d. Also, in contrast to the notion of strong control

[2] We could specify that this path should be nontrivial, as otherwise it will hold for all n_1 that $n_1 \stackrel{scd}{\rightarrow} n_1$, but since such spurious dependences do not contribute to the slice set, we shall not bother about that.

[3] To see this, first assume that $m \stackrel{cd}{\rightarrow} n$ with $m \neq n$, that is, n does not postdominate m, and there exists a path π from m to n such that n postdominates all nodes in π except for m. We shall prove that π is a witness that $m \stackrel{scd}{\rightarrow} n$ does hold, and do so by contradiction; we thus assume that π contains a node u which is the immediate postdominator of m. Since $u \neq m$, it holds that n postdominates u. As postdomination is transitive, n postdominates m, yielding the desired contradiction.

Conversely, assume that $m \stackrel{scd}{\rightarrow} n$ with $m \neq n$, that is, with u as the immediate postdominator of m, there exists a path π from m to n that does not contain u. We shall prove that $m \stackrel{cd}{\rightarrow}{}^{*} n$ does hold, by induction on the length of π. Since n does not postdominate m (as otherwise π would contain u) but does postdominate itself, there exists $n_1 \in \pi$ such that n does not postdominate n_1, but does postdominate all nodes after n_1 in π. This shows that $n_1 \stackrel{cd}{\rightarrow} n$. If $m = n_1$, we are done. Otherwise, since clearly $m \stackrel{scd}{\rightarrow} n_1$, the induction hypothesis yields the claim.

Table I. Various Control Dependences Existing in the Graph in Figure 1(a)

Nodes	$\stackrel{cd}{\rightarrow}$	$\stackrel{scd}{\rightarrow}$	$\stackrel{wcd}{\rightarrow}$	$\stackrel{ntscd}{\rightarrow}$	$\stackrel{nticd}{\rightarrow}$
a	b, c, d, f, h	b, c, d, f, g, h	b, c, f, h, e	b, c, f, h, e	b, c, d, f, h
c	b, c	b, c	b, c, d, e	b, c, d, e	b, c
f	g	g	g	g	g

Control dependences denoted by $\stackrel{ntscd}{\rightarrow}$ and $\stackrel{nticd}{\rightarrow}$ will be introduced in the following pages.

dependence, the notion of weak control dependence is direct. For instance, in Figure 1(a), we do not have $a \stackrel{wcd}{\rightarrow} g$. Hence, $n_1 \stackrel{scd}{\rightarrow} n_2$ does not imply $n_1 \stackrel{wcd}{\rightarrow} n_2$, but $n_1 \stackrel{scd}{\rightarrow} n_2$ does imply[4] $n_1 \stackrel{wcd^*}{\rightarrow} n_2$.

In assessing the aforementioned variants of control dependence in the context of program slicing, it is important to note that slicing based on strong control dependence (Definition 5, or equivalently, Definition 3) can transform a nonterminating program into a terminating one (i.e., nontermination is not preserved in the slice). In Figure 1(a), assume that the loop controlled by c is infinite. Using the slice criterion $C = \{d\}$, slicing using strong control dependence would generate a slice that includes a, but not b and c (we assume no data dependence between d and either b or c). Thus, in the sliced program, one would be able to observe an execution of d, but such an observation is not possible in the original program because execution diverges before d is reached. This shows that there is a profound difference between strong and weak control dependence. In contrast, the difference between direct and indirect statements of control dependence seems to amount to a largely technical stylistic decision in how the definitions are stated. Table I shows the control dependences that arise in the CFG of Figure 1(a) for various notions of control dependence that we are considering in this work.

Very few efforts consider the nontermination-sensitive notion of weak control dependence described earlier. We conjecture that there are at least two reasons for this. First, although it bears the qualifier "weak," weak control dependence produces a larger transitive closure and will thus include more nodes in the slice.[5] Second, many applications of slicing focus on debugging, program visualization, and understanding, and as such, in these applications having slices that preserve nontermination is less important than having smaller slices. However, slicing is increasingly used in security applications and as a model-reduction technique for software model-checking. In these applications, it is quite important to consider variants of control dependence that preserve nontermination properties, since failure to do so could allow inferences to be made that compromise security policies, for instance, invalidating checks of liveness properties [Hatcliff et al. 2000]. Our definitions of control dependence and slicing, to be presented later in this work, are motivated by careful consideration of nonterminating program behaviors.

---

[4]As can actually be shown by combining results presented later in this article: $\stackrel{scd}{\rightarrow}$ implies (see Footnote 3) $\stackrel{cd}{\rightarrow}^*$ implies (Theorem 1) $\stackrel{nticd}{\rightarrow}^*$ implies (Theorem 4) $\stackrel{ntscd}{\rightarrow}^*$ implies (Theorem 3) $\stackrel{wcd}{\rightarrow}^*$.

[5]In Figure 1(a), the transitive closure of strong and weak control dependence starting from d are $\{a\}$ and $\{a, c\}$, respectively.

3.2 Unique End Node Restriction on CFGs

All definitions of control dependences of which we are aware require that CFGs satisfy the unique end node requirement, but many software systems fail to satisfy this property. Existing works simply require that CFGs have this property, or suggest that CFGs can be augmented to achieve such, for example, by using the following steps: (1) Insert a new node e into the CFG, (2) add an edge from each exit node (other than e) to e, (3) pick an arbitrary node n in each nonterminating loop and add an edge from n to e. In our experience, such augmentations complicate the system being analyzed in several ways. Nondestructive augmentation performed by cloning the CFG and augmenting the clone would cost time and space. Destructive augmentation performed by directly augmenting the CFG may clash with the requirements of other clients of the CFG, thus necessitating reversal of the augmentation before subsequent clients use the CFG. Otherwise, graph and analysis algorithms should be made to operate on the actual CFG embedded in the augmented version.

Many systems have threads where the main control loop has no exit; the loop is "exited" simply by killing the thread. For example, in the Xt library, most applications create widgets, register callbacks, and call `XtAppMainLoop()` to enter an infinite loop that manages the dispatching of events to the widgets in the application. In PalmOS, applications are designed so as to start upon receiving a start code, execute a loop, and terminate upon receiving a stop code. However, the application may choose to ignore the stop code during execution. Hence, the application may not terminate, except when explicitly killed. In such cases, a node in the loop must be picked as the loop exit node for the purpose of augmenting the CFG of the application. But this can disrupt the control dependence calculations. In Figure 1(b), we would intuitively expect e,b,c, and d to be control dependent on a in the unaugmented CFG. However, $a \overset{wcd}{\rightarrow} \{e, b, c\}$ and $c \overset{wcd}{\rightarrow} \{b, c, d, f\}$ in the augmented CFG. It is trivial to prune dependences involving f. However, now there are new dependences $c \overset{wcd}{\rightarrow} \{b, c, d\}$ which did not exist in the unaugmented CFG. From the given example, one may be tempted to believe that a solution would be to delete any dependence on c, but this would fail if there exists a node g that is a successor of c and a predecessor of d. Also, $a \overset{wcd}{\rightarrow} d$ exists in the unaugmented CFG but not in the augmented one, and it is not obvious how to recover this dependence.

We address these issues head-on by considering alternate definitions of control dependence that do not impose the unique end node restriction.

4. NEW DEPENDENCE DEFINITIONS

In previous definitions, a control dependence relationship where n_j is dependent on n_i is specified by considering paths from n_i and n_j to a unique CFG end node; essentially, n_i and the end node delimit the path segments that are considered. Since we aim for definitions that apply when CFGs do not have an end node or have more than one, we aim to instead to specify that n_j is control dependent on n_i by focusing on paths between n_i and n_j. Specifically, we focus on path segments that are delimited by n_i *at both ends*, intuitively corresponding to the situation in a reactive program where rather than reaching an end node, a program's behavior begins to repeat itself by returning again to n_i. At a high level, the intuition behind control dependence remains the same as in, for instance, Definition 3: Executing one branch of n_i always leads to n_j, whereas executing another branch of n_i can cause n_j to be bypassed. The additional constraints (e.g., n_j always occurs before any occurrence of n_i) limit the region in which n_j is seen or bypassed to segments leading up to the next occurrence of n_i, ensuring that n_i is indeed *controlling* n_j.

Road map. We shall propose in Definition 7 a general notion of control dependence which is sensitive to nontermination, called $\stackrel{ntscd}{\rightarrow}$; it turns out that $\stackrel{ntscd}{\rightarrow}$ can be given an equivalent (Lemma 2) but simpler formulation (Definition 16). This new notion (Theorem 3) is a conservative extension of weak control dependence (compare with Definition 6) in that they agree on CFGs with the unique end node property.

Similarly, we propose in Definition 10 a general notion of control dependence which is *in*sensitive to nontermination, called $\stackrel{nticd}{\rightarrow}$. As expected, $\stackrel{nticd}{\rightarrow}$ produces (Theorem 4) a slice set which, in general, is a subset of what is produced by $\stackrel{ntscd}{\rightarrow}$. This new notion (Theorem 1) is a conservative extension of standard control dependence (compare with Definition 3) in that they agree on CFGs with the unique end node property.

In Theorem 6 we state the correctness of slicing; the formulation is based on bisimulation and therefore particularly requires slicing to preserve termination. For that to be the case, we must demand the slice set to be closed, not just under $\stackrel{nticd}{\rightarrow}$ (as well as under $\stackrel{dd}{\rightarrow}$), but even under $\stackrel{ntscd}{\rightarrow}$. Furthermore, for irreducible graphs we must in addition demand the slice set to be closed under "decisive order dependence" (Definition 12), a requirement which is void (Lemma 3) for reducible CFGs.

If the slice set is only closed under $\stackrel{nticd}{\rightarrow}$ but not under $\stackrel{ntscd}{\rightarrow}$, then loops may be sliced away so that the sliced program terminates more often than the original. In that case, we can aim for no more than "partial correctness," as will be established in a forthcoming paper by some of the authors.

Additional variations of control dependence are proposed that we believe will prove useful in specialized settings: (1) *decisive control dependence* (Definition 11) is a variant of $\stackrel{ntscd}{\rightarrow}$ which is relevant in identifying control frontiers ideal in control flow graphs with high branching (as in the case of exceptions); and (2) several variants (*strong* (Definition 13), *weak* (Definition 14), *data-sensitive* (Definition 15)) of *order dependence* relevant in reasoning about the influence of control flow in programs with irreducible control flow graphs.

4.1 Nontermination-Sensitive Control Dependence

The next definition considers maximal (which includes infinite) paths and thus is sensitive to nontermination.

Definition 7 ($n_i \stackrel{ntscd}{\rightarrow} n_j$). In a CFG, n_j is *(directly) nontermination-sensitive control dependent* on node n_i iff n_i has at least two successors, n_k and n_l, such that:

(1) For all maximal paths π from n_k, n_j always occurs in π and either $n_i = n_j$ or n_j strictly ($n_j \neq n_i$) precedes any occurrence of n_i in π; and
(2) there exists a maximal path π_0 from n_l on which either n_j does not occur, or n_i strictly precedes any occurrence of n_j in π_0.

Remark 1. When we, as earlier, write "n_j strictly precedes any occurrence of n_i in π," we mean that: *(a)* n_j occurs in π; and either *(b1)* n_i does not occur in π; or *(b2)* the first occurrence of n_j in π is earlier than the first occurrence of n_i.

We supplement[6] a traditional presentation of dependence definitions with definitions given as formulas in computation tree logic (CTL) [Clarke et al. 1999]. CTL is a logic for describing the structure of sets of paths in a graph, making it a natural language for expressing control dependences. Informally, CTL includes two path quantifiers, E and A, which indicate whether a path from a given node with a given structure exists, or if all paths from that node have the given structure. The structure of a path is defined using one of five modal operators (we refer to a node satisfying the CTL formula ϕ as a ϕ-node): $\mathsf{X}\phi$ states that the successor node is a ϕ-node, $\mathsf{F}\phi$ states the existence of a ϕ-node in the path, $\mathsf{G}\phi$ states that a path consists entirely of ϕ-nodes, $\phi\ \mathsf{U}\psi$ states the existence of a ψ-node and that the subpath leading up to that node consists of ϕ-nodes, and finally, the $\phi\ \mathsf{W}\psi$ operator is a variation on U that relaxes the requirement that a ψ-node exists (if not, all nodes in the path must be ϕ-nodes). In a CTL formula, path quantifiers and modal operators occur in pairs, for example, $\mathsf{AF}\phi$ says that on all paths from a node, the ϕ node occurs. A formal definition of CTL can be found in Clarke et al. [1999].

The following CTL formula captures the previous definition of control dependence.

$$n_i \stackrel{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{A}[\neg n_i\ \mathsf{U} n_j]) \wedge \mathsf{EX}(\mathsf{E}[\neg n_j\ \mathsf{W}(\neg n_j \wedge n_i)])$$

Here, $(G, n_i) \models$ expresses the fact that the CTL formula is checked against the graph G at node n_i. The two conjuncts are essentially a direct transliteration of the aforementioned natural language.

We have formulated the preceding definition to apply to execution traces instead of CFG paths. In this setting, one needs to bound relevant segments by n_i, as discussed before. However, when working on CFG paths, the conditions in Definition 7 can actually be simplified to read as follows: (1) For all maximal paths from n_k, n_j always occurs; and (2) there exists a maximal path from n_l on which n_j does not occur. The corresponding CTL formula would be

$$n_i \stackrel{ntscd}{\rightarrow} n_j = (G, n_i) \models \mathsf{EX}(\mathsf{AF}(n_j) \wedge \mathsf{EX}(\mathsf{EG}(\neg n_j)).$$

See Section 4.5 (Lemma 2 and Theorem 2) for the proof that Definition 7 and its simplification are equivalent on CFGs.

[6]The development in this article is based on traditional path reasoning (even the proof of Theorem 2, which states the equivalence between two CTL formulas). The reason for rephrasing our definitions into CTL is to encourage exploration of a model-checking approach to computing control dependences.

Examples. To see that Definition 7 is nontermination sensitive, note that $c \overset{ntscd}{\to} d$ in Figure 1(a), since there exists a maximal path (i.e., an infinite loop between b and c) where d never occurs. Moreover, the definition corresponds to our intuition in Section 3.2 in that, in the unaugmented part of Figure 1(b), $a \overset{ntscd}{\to} e$ because there is an infinite loop through b, c, d that does not contain e, and $a \overset{ntscd}{\to} \{b, c, d\}$ because there is maximal path ending in e that does not contain b, c, or d.

In Figure 1(c), we have $a \overset{ntscd}{\to} b$, as the first execution of b depends on the choice made at a. Likewise, $a \overset{ntscd}{\to} c$, $a \overset{ntscd}{\to} f$, $a \overset{ntscd}{\to} h$, $a \overset{ntscd}{\to} e$, and $f \overset{ntscd}{\to} g$. On the other hand, $f \overset{ntscd}{\not\to} h$, since, independent of the choice made at f, the control will always reach h. We have $c \overset{ntscd}{\to} b$, $c \overset{ntscd}{\to} c$, and also $c \overset{ntscd}{\to} d$, since if $b \to c \to b$ is an infinite loop, control will never reach d. Note that $d \overset{ntscd}{\to} i$ because there is an infinite path from j (cycle on jdj) on which i does not occur (though it is also possible that that the control will bypass i in one iteration, while reaching i in a subsequent iteration, depending on the choice made at d).

4.2 Nontermination-Insensitive Control Dependence

We now turn to constructing a nontermination-insensitive version of control dependence. The earlier described nontermination-sensitive definition considered all paths leading out of a conditional. Now, we need to limit the reasoning to finite paths that reach a terminal region of the graph. To handle this in the context of CFGs that do not have the unique end node property, we generalize the concept of *end node* to *control sink*: a set of nodes such that each in the set is reachable from every other in the set and there is no path leading out of the set. More precisely,

Definition 8 (*Control Sink*). A *control sink* κ is a set of CFG nodes that form a strongly connected component such that for each $n \in \kappa$, each successor of n is also in κ.

Observe that each end node forms a control sink, as does each loop without any exit edges in the graph. For example, $\{e\}$ and $\{b, c, d\}$ are control sinks in Figure 1(b) unaugmented, and $\{e\}$ and $\{d, i, j\}$ are control sinks in Figure 1(c).

Definition 9 (*Sink-Bounded Path*). The set of *sink-bounded paths from* n_k (denoted $SinkPaths(n_k)$) contains all *maximal* paths π from n_k with the property that there exists a control sink κ such that:

—π contains a node n_s from κ (hence, all nodes following n_s in π will also belong to κ); and
—if π is infinite, then all nodes in κ will occur in π infinitely often.

We shall discuss the latter requirement later in this section. Note that for a CFG with unique end node n_e, a path is sink bounded iff it ends in n_e; also note that if π_1 is a suffix of π_2, then π_1 is sink bounded iff π_2 is sink bounded. Given a control flow graph, the minor formed by contracting the strongly connected components of the control flow graph will be a DAG with the control sinks contracted into leaf nodes. This shows

Table II. Various Control Dependences (based on new definitions) Existing in the Graph in Figure 1(c)

Nodes	$\xrightarrow{ntscd}$	$\xrightarrow{nticd}$
a	b, c, f, h, e	b, c, d, i, j, f, h, e
c	b, c, d, j	b, c
d	i	–
f	g	g

LEMMA 1. *All finite paths can be extended into sink-bounded paths.*

Existing definitions [Ball and Horwitz 1993; Podgurski and Clarke 1990; Bilardi and Pingali 1996] of nontermination-insensitive control dependence rely on reasoning about paths from the conditional to the end node. We generalize this to reason about paths from a conditional to control sinks.

Definition 10 ($n_i \xrightarrow{nticd} n_j$). In a CFG, n_j is *(directly) nontermination-insensitive control dependent* on n_i iff n_i has at least two successors, n_k and n_l, such that:

(1) For all paths $\pi \in SinkPaths(n_k)$, $n_j \in \pi$; and
(2) there exists a path $\pi \in SinkPaths(n_l)$ such that $n_j \notin \pi$.

This definition is expressed in CTL as

$$n_i \xrightarrow{nticd} n_j = (G, n_i) \models \mathsf{EX}(\hat{\mathsf{A}}\mathsf{F}(n_j)) \wedge \mathsf{EX}(\hat{\mathsf{E}}\mathsf{G}(\neg n_j)),$$

where $\hat{\mathsf{A}}$ and $\hat{\mathsf{E}}$ represent quantification over sink-bounded paths only; note the similarity to the simplified formula for $\xrightarrow{ntscd}$ mentioned earlier.

Examples. To see that this definition is nontermination insensitive, note that $c \overset{nticd}{\not\to} d$ in Figure 1(a), since there does not exist a path from b to a control sink ($\{e\}$ is the only control sink) that does not contain d. Again, in Figure 1(b), unaugmented, $a \xrightarrow{nticd} e$ because there is a path from b to the control sink $\{b, c, d\}$ and neither the path nor the sink contains e, and $a \xrightarrow{nticd} \{b, c, d\}$ because there is a path ending in control sink $\{e\}$ that does not contain b, c, or d.

The dependences of Figure 1(c) are listed in Table II. Most of the nontermination-sensitive control dependences also hold in the nontermination-*in*sensitive case, except for three, explained next. First observe that in a nontermination-insensitive setting, loops are assumed to be terminating, provided that the loop has an exit node. Therefore we have $c \overset{nticd}{\not\to} d$, since the loop $b \to c \to b$ is assumed terminating because it has an exit edge $c \to d$. Also, we have $c \overset{nticd}{\not\to} j$, as j belongs to the control sink that terminates all sink-bounded paths from c.

Finally, we have $d \overset{nticd}{\not\to} i$, even though $\{d, i, j\}$ is a control sink and there is a maximal path from d that avoids i (by choosing j over i each time), but this path is not sink bounded, thanks to the last requirement in Definition 9, which requires i to occur infinitely often. This "fairness" requirement has as

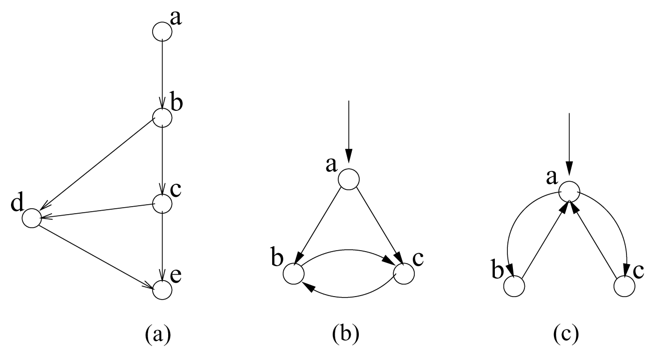


Fig. 2. More control flow graphs.

the consequence that even though there may be control structures inside of a control sink, they will not give rise to any control dependences. In applications where one desires to detect such dependences, one may apply the definition to control sinks in isolation with back edges removed, or use order dependence (described to follow in Definition 14).

Alternatively, we might drop the fairness requirement. That would make no difference for a CFG with a unique end node. For a CFG without a unique end node, the relation $n_i \stackrel{nticd}{\rightarrow} n_j$ might change, but would still satisfy the properties listed later in this work (e.g., Theorem 4). We leave this to further experiments, particularly when conducted in a concurrent context, to decide the respective merits of the two definitions.

4.3 Decisive Dependence

In languages like Java, exception-based control flow paths give rise to control flow graphs with shapes similar to that in Figure 2(a). In this CFG, $b \stackrel{cd}{\rightarrow} c$, $b \stackrel{cd}{\rightarrow} d$, and $c \stackrel{cd}{\rightarrow} d$. In the case of $b \stackrel{cd}{\rightarrow} d$, it is possible for the control to reach d even if the control flows along $b \rightarrow c$. Hence, b does not *decisively* determine whether control can bypass d. However, in case of $c \stackrel{cd}{\rightarrow} d$, c does *decisively* determine whether control can bypass d. The decisiveness stems from the fact that the choice at the control point (c), which prevents the control from reaching the given program point (d), is *final*. Hence, the decisive control dependence relation can be defined as follows.

Definition 11 ($n_i \stackrel{dcd}{\rightarrow} n_j$). In a CFG, n_j is *(directly) decisively control dependent* on node n_i iff n_i has at least two successors, n_k and n_l, such that:

(1) For all maximal paths from n_k, n_j always occurs and either $n_j = n_i$ or n_j strictly precedes n_i; and
(2) for all maximal paths from n_l, n_j does not occur or is strictly preceded by n_i.

For Figure 2(a), we do indeed have $c \stackrel{dcd}{\rightarrow} d$ and $b \stackrel{dcd}{\not\rightarrow} d$.

Although Definitions 7 and 11 are almost identical, they differ in quantification in the second clause. Hence, the previous definition implies Definition 7.

Decisive control dependence is useful to answer the question *"which is the control point beyond which the control cannot reach the given program point?"* This information is useful when trying to understand procedures with multiple exit points that are embedded in a nested control structure.

4.4 Order Dependence

Programs written in unstructured languages such as JVM bytecode can give rise to irreducible CFGs for which previous definitions prove to be insufficient to capture dependences. For example, in Figure 2(b), b and c cannot be related to a by any of the aforementioned dependences, as, given the shape of the CFG, the control will reach b and c once it enters the control sink $\{b, c\}$. However, a does influence whether b or c will be executed first. In other words, the order in which b and c are executed within the control sink is determined by a. To capture ordering relationships between nodes such as a, b, and c in irreducible regions of a CFG, we propose a new notion of dependence called order dependence.

Definition 12 ($n_1 \stackrel{dod}{\rightarrow} n_2 \leftrightharpoons n_3$). Let n_1, n_2, n_3 be distinct nodes. The pair of nodes n_2 and n_3 are strongly order-dependent on n_1, written $n_1 \stackrel{dod}{\rightarrow} n_2 \leftrightharpoons n_3$, if:

(1) All maximal paths from n_1 contain both n_2 and n_3;
(2) n_1 has a successor from which all maximal paths[7] contain n_2 before any occurrence of n_3; and
(3) n_1 has a successor from which all maximal paths contain n_3 before any occurrence of n_2.

We shall use decisive order dependence in our exposition about slicing and the associated correctness proofs.

Observe that the preceding definition is decisive, as it requires that n_1 be the final control point to decide the execution order between n_2 and n_3. By relaxing this requirement, we can arrive at a relatively weaker relation which we shall refer to as *strong order dependence*. As given in the following definition, universal quantification on maximal paths is required for one of n_2 and n_3, the successor nodes of n_1.

Definition 13 ($n_1 \stackrel{sod}{\rightarrow} n_2 \leftrightharpoons n_3$). Let n_1, n_2, n_3 be distinct nodes. Both n_2 and n_3 are strongly order-dependent on n_1, written $n_1 \stackrel{sod}{\rightarrow} n_2 \leftrightharpoons n_3$, if:

(1) All maximal paths from n_1 contain both n_2 and n_3;
(2) there exists a maximal path from n_1 where n_2 occurs before any occurrence of n_3;
(3) there exists a maximal path from n_1 where n_3 occurs before any occurrence of n_2; and
(4) n_1 has a successor n_4 such that either:
    (a) All maximal paths from n_4 contain n_2 before any occurrence of n_3; or
    (b) all maximal paths from n_4 contain n_3 before any occurrence of n_2.

[7]which will contain both n_2 and n_3, thanks to clause (1).

The strong order dependence definition can be further generalized to capture control dependence, hence, applicable to reducible regions of the CFG. The generalization is achieved by removing clause (1) from Definition 13, as done in the following definition.

Definition 14 ($n_1 \overset{wod}{\rightarrow} n_2 \leftrightharpoons n_k$). In a CFG, nodes n_2 and n_3 ($n_2 \neq n_3$) are weakly order dependent on n_1 iff:

—There exists a maximal path from n_1 where n_2 strictly precedes any occurrence of n_3;
—there exists a maximal path from n_1 where n_3 strictly precedes any occurrence of n_2; and
—n_1 has a successor n_l such that either:
  —On all maximal paths from n_l, n_2 strictly precedes any occurrence of n_3; or
  —on all maximal paths from n_l, n_3 strictly precedes any occurrence of n_2.

Although order dependence captures the ordering on nodes imposed by control flow, it is overly conservative in cases where such ordering is required only to preserve the data values observed during execution. In other words, if there is no variable that is used (defined) in b and defined (used) in c, then the data values observed during execution of b and c are independent of the order in which b and c are executed. In such cases, the execution order imposed by a on b and c is uninteresting if observed only by changes to the variables used in b and c, and not by the order of program points encountered during execution. This data-sensitive order relation is captured by *data-sensitive order dependence*, a stronger form of order dependence.

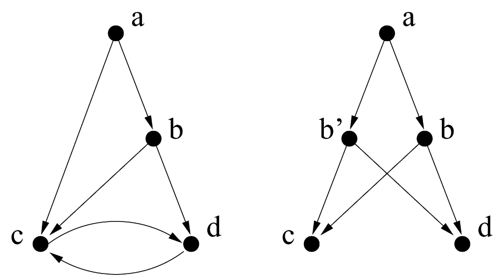
Definition 15 ($n_1 \overset{dsod}{\rightarrow} n_2 \leftrightharpoons n_3$). In a CFG, nodes n_2 and n_3 ($n_2 \neq n_3$) are *data-sensitive order dependent* on n_1 iff:

(1) $n_1 \overset{sod}{\rightarrow} n_2 \leftrightharpoons n_3$; and
(2) either $n_2 \overset{dd}{\rightarrow} n_3$ or $n_3 \overset{dd}{\rightarrow} n_2$.

Examples. In Figure 2(b), b and c are decisively order dependent on a ($a \overset{dod}{\rightarrow} b \leftrightharpoons c$), and also strongly ($a \overset{sod}{\rightarrow} b \leftrightharpoons c$) and weakly ($a \overset{wod}{\rightarrow} b \leftrightharpoons c$) order dependent on a. Hence, $b \overset{dd}{\rightarrow} c$ or $c \overset{dd}{\rightarrow} b$ implies $a \overset{dsod}{\rightarrow} b \leftrightharpoons c$. In Figure 1(c), i and j are weakly order dependent on d ($d \overset{wod}{\rightarrow} i \leftrightharpoons j$), but neither strongly or decisively order dependent on d, as there exists a maximal path from d not containing i. In Figure 3(a), both c and d are strongly as well as weakly order dependent on both b and a. However, c and d are decisively order dependent only on b. In Figure 3(b), c and d are weakly order dependent on b as well as on b', but neither strongly nor decisively order dependent on any of b, b', and a due to the absence of edges $c \rightarrow d$ and $d \rightarrow c$.

Given the definition of various forms of order dependences and the fact[8] that every cycle in a reducible CFG has one node that dominates all others of the cycle, it is tempting to conclude that there can be no order dependences of any

[8]Definition (f) in the abstract of Hecht and Ullman [1974].



(a) (b)

Fig. 3. Control flow graphs specific to order dependence.

form between n_i, n_j, and n_k, provided that they are distinct and occur in a reducible CFG. This is true (as proved in Lemma 3) in cases of decisive and strong variants of order dependence. However, it is not true in the case of weak order dependence. As an example, observe that b and c are weakly (but not strongly) order dependent on a ($a \stackrel{wod}{\rightarrow} b \leftrightharpoons c$) in the reducible graph of Figure 2(c). We *conjecture* that in a reducible CFG, $a \stackrel{wod}{\rightarrow} b \leftrightharpoons c \Longrightarrow a \stackrel{ntscd}{\rightarrow} b \vee a \stackrel{ntscd}{\rightarrow} c$.

4.5 Properties of the Dependence Relations

Conservative extension. We begin by showing that the new definitions of control dependence conservatively extend classic ones: When we consider our definitions in the original setting with CFGs having unique end nodes, they coincide with the classic definitions (as already suggested by the examples in Table I).

THEOREM 1 (COINCIDENCE PROPERTIES, I). *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$, $n_i \stackrel{cd}{\rightarrow} n_j$ if and only if $n_i \stackrel{nticd}{\rightarrow} n_j$.*

PROOF. First notice that for any n and m, m postdominates n if and only if every sink-bounded path from n contains m.

We shall first prove "only if": So let $n_i \stackrel{cd}{\rightarrow} n_j$. There thus exists a nontrivial path π from n_i to n_j such that every node in π except n_i is postdominated by n_j. Let π take the form $n_i, n_k, \ldots, n_j$; we can assume that if $n_j \neq n_i$, then $n_k \neq n_i$. Here n_k may equal n_j, but in any case it will hold that n_k is postdominated by n_j.

Also, we know from $n_i \stackrel{cd}{\rightarrow} n_j$ that n_i is not strictly postdominated by n_j. Therefore either $n_i = n_j$ or n_i is not postdominated by n_j. In either case, since the end node is reachable from all nodes, we infer that n_i has a successor n_l which is not postdominated by n_j.

We have thus found n_k and n_l, namely successors of n_i, such that all sink-bounded paths from n_k contain n_j, and such that there exists a sink-bounded path from n_l not containing n_j. This shows $n_i \stackrel{nticd}{\rightarrow} n_j$.

Next we prove "if": So let $n_i \stackrel{nticd}{\rightarrow} n_j$. Thus n_i has (at least) two successors, n_k and n_l, such that: (i) All sink-bounded paths from n_k contain n_j; and (ii) there exists a sink-bounded path from n_l not containing n_j. From (ii), we infer that either $n_i = n_j$ or n_i is not postdominated by n_j; in either case, n_i is not strictly postdominated by n_j.

Since the end node n_e is reachable from all nodes, we know from (i) that there exists a path from n_k to n_j; let π be a shortest such path. In order to show that $n_i \stackrel{cd}{\rightarrow} n_j$, it suffices to show that all nodes in π are postdominated by n_j. But this clearly follows from (i). □

Before we prove the coincidence property between weak control dependence and the nontermination-sensitive control dependence, we prove the equivalence between the original and simplified definitions of the latter. For readability, we restate the simplified definition of nontermination-sensitive control dependence.

Definition 16. In a CFG, n_j is nontermination-sensitive control dependent on n_i iff:

(a) n_i has at least two successors n_k and n_l;
(b) on all maximal paths from n_k, n_j occurs; and
(c) there exists a maximal path from n_l on which n_j does not occur.

LEMMA 2. *For a CFG, Definition* 16 *is equivalent to the original definition of nontermination-sensitive control dependence, Definition* 7.

PROOF. First, we restate the definition of directly nontermination-sensitive control dependence (i.e., Definition 7); we have $n_i \stackrel{ntscd}{\rightarrow} n_j$ iff:

—ntscd(i) n_i has at least two successors, n_k and n_l;
—ntscd(ii) for all maximal paths from n_k, n_j always occurs and either equals n_i or occurs before any occurrence of n_i; and
—ntscd(iii) there exists a maximal path from n_l on which either n_j does not occur, or n_j is strictly preceded by n_i.

Since **ntscd(ii)** implies **(b)**, and **(c)** implies **ntscd(iii)**, we are left to show two implications:

—First, we show that **(b)** implies **ntscd(ii)**: Let π be a maximal path from n_k. By **(b)**, n_j occurs there. Now assume towards a contradiction that in π, n_i occurs strictly before any occurrence of n_j. Since there is an edge from n_i to n_k, this means that the graph has a cycle containing n_k, but not containing n_j. But then we can find a maximal path from n_k where n_j does not occur, contradicting **(b)**.
—Next, we show that **ntscd(iii)** implies **(c)**: Let π be a maximal path from n_l on which n_i occurs strictly before the first (if any) occurrence of n_j. If π does not contain n_j, we are done. So assume that π does contain n_j, but that n_i occurs strictly before. But since there is an edge from n_i to n_l, this means that the graph has a cycle containing n_l but not containing n_j. Then we can find a maximal path from n_l where n_j does not occur, as desired.

This concludes the proof of Lemma 2. Note that we have not assumed the unique end node property. □

As can be expected, we have a similar result relating the corresponding CTL formulaes.

THEOREM 2 (SIMPLIFIED NTSCD CTL EQUIVALENCE). *The expression of NTSCD as a CTL formula over CFG paths (ϕ_{CFG}), namely,*

$$n_i \stackrel{ntscd}{\rightarrow} n_j = (G, n_i) \models \; \mathit{EX}(\mathit{AF}(n_j)) \wedge \; \mathit{EX}(\mathit{EG}(\neg n_j)),$$

is equivalent to the CTL formula over execution traces (ϕ_{trace}), namely,

$$n_i \stackrel{ntscd}{\rightarrow} n_j = (G, n_i) \models \; \mathit{EX}(\mathit{A}[\neg n_i \; \mathit{U} n_j]) \wedge \; \mathit{EX}(\mathit{E}[\neg n_j \; \mathit{W}(\neg n_j \wedge n_i)]).$$

PROOF. We shall use regular expressions over CFG node names to describe the structure of CFG paths. In this context, negation, that is, $\neg n_j$, is used to denote the absence of a particular control point n_j.

It suffices to prove that the pairs of subformulas under the EX operators in the two formulas are equivalent.

We prove that ϕ_{CFG} implies ϕ_{trace} in two steps:

(1) $\mathsf{EG}(\neg n_j)$ implies $\mathsf{E}[\neg n_j \; \mathsf{W}(\neg n_j \wedge n_i)]$:
The definition of EW requires its left operand to be true until the right operand holds. Thus, if the left operand holds throughout the trace, by the definition of $\mathsf{EG}(\neg n_j)$, then $\neg n_j$ must hold until $\neg n_j \wedge n_i$.

(2) $\mathsf{AF}(n_j)$ implies $\mathsf{A}[\neg n_i \; \mathsf{U} n_j]$):
The AU operator requires that a n_j state is reached, which holds by the definition of $\mathsf{AF}(n_j)$, and that all prefixes of traces ending in n_j must be free of n_i states.

Every path from a CFG node n_i either has a prefix that is cyclic in n_i, namely $n_i(\neg n_i)^* n_i$, or is a path that is acyclic in n_i, namely $n_i(\neg n_i)^*$. All proper suffixes of paths that are acyclic in n_i are free of n_i by definition. If there exists a path with a prefix that is cyclic in n_i, then there must exist a CFG path of the form $(n_i(\neg n_i)^* n_i)^*$. If $\mathsf{AF}(n_j)$ holds on such a path, then it must be the case that n_j appears in the body of the cycle $(\neg n_i)^*$. Thus, all paths that satisfy $\mathsf{AF}(n_j)$ and begin with a prefix that is cyclic in n_i must begin with a prefix of the form $n_i(\neg n_i)^* n_j$.

Thus ϕ_{CFG} implies ϕ_{trace}.

We prove that ϕ_{trace} implies ϕ_{CFG} in two steps:

(1) $\mathsf{A}[\neg n_i \; \mathsf{U} n_j]$) implies $\mathsf{AF}(n_j)$:
The AU operator requires that eventually its right operand n_j becomes true, which is the definition of $\mathsf{AF}(n_j)$.

(2) $\mathsf{E}[\neg n_j \; \mathsf{W}(\neg n_j \wedge n_i)]$ implies $\mathsf{EG}(\neg n_j)$:
If the right operand of the EW never becomes true in a trace, then $\neg n_j$ must hold throughout the trace, which is equivalent to enforcing $\mathsf{EG}(\neg n_j)$.

The EW operator, however, only requires $\neg n_j$ to hold up along the trace to the point where n_i holds. For the implication to hold we must show that $\neg n_j$ will persist through the rest of the path.

Consider a CFG path from n_i that is free of n_j up to the first occurrence of n_i; this satisfies the aforementioned EW. This path has a prefix of the form

$n_i(\neg n_j)^* n_i$ and by iterating that prefix we can construct a path $(n_i(\neg n_j)^* n_i)^*$ that satisfies $\mathsf{EG}(\neg n_j)$.

Thus ϕ_{trace} implies ϕ_{CFG}. □

THEOREM 3 (COINCIDENCE PROPERTIES, II). *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$, $n_i \stackrel{wcd}{\rightarrow} n_j$ if and only if $n_i \stackrel{ntscd}{\rightarrow} n_j$.*

PROOF. By Lemma 2, we can prove the equivalence by showing that Podgurski-Clarke's weak control dependence from Definition 6 is equivalent to Definition 16.

For readability, we restate Podgurski-Clarke's definition of weak control dependence; we have $n_i \stackrel{wcd}{\rightarrow} n_j$ iff:

—pcwcd(i) n_i has at least two successors, n_k and n_l;
—pcwcd(ii) n_j strongly postdominates n_k; and
—pcwcd(iii) n_j does not strongly postdominate n_l.

There are four steps.

(1) **pcwcd(ii)** implies **(b)**: Let π be a maximal path from n_k. We must show that n_j occurs in π. There are two possibilities.
 —*π is finite*: The last node of π must be an end node. Since n_j postdominates n_k, this shows that n_j occurs in π; or
 —*π is infinite*: We know that there exists q such that all paths from n_k longer than q contain n_j; in particular, π will contain n_j since π is infinite, hence longer than q.
(2) **(b)** implies **pcwcd(ii)**: First let us show that n_j postdominates n_k; so let π be a path from n_k to an end node. We must show that π contains n_j, but this follows from **(b)**, since π is maximal.
 Next we must find a q such that all paths from n_k longer than q contain n_j; we claim that we can choose q to be one more than the number of nodes in the CFG. Let π be a path from n_k longer than q: It contains a repetition, so if n_j does not occur in π, we can construct a maximal path from n_k with n_j not occurring, yielding a contradiction.
(3) **pcwcd(iii)** implies **(c)**: Here we have two cases.
 —*n_j does not postdominate n_l*: Then there exists a path π from n_l to an end node such that n_j does not occur in π. The claim now follows, since π is maximal; or
 —*For all q, there exists a path from n_l longer than q where n_j does not occur*: With q the number of nodes in the CFG, we infer that there exists a path from n_l containing repetitions, but not containing n_j; this shows that we can construct a maximal (infinite) path from n_l on which n_j does not occur.
(4) **(c)** implies **pcwcd(iii)**: Our assumption is that there exists a maximal path π from n_l with n_j not occurring in π. Now there are two cases:

—*π is finite, with the last node being an end node*: However, then n_j does not postdominate n_l; in particular, n_j does not strongly postdominate n_l; or

—*π is infinite*: But then for any q, π will be a path from n_l of length q not containing n_j, again showing that n_j does not strongly postdominate n_l.

This concludes the proof of Theorem 3. □

Nontermination sensitivity relates more nodes. For an arbitrary CFG, direct nontermination-insensitive control dependence (Definition 10) implies the *transitive closure* of direct nontermination-sensitive control dependence.

THEOREM 4. *For all CFGs (with or without the unique end node property), and for all nodes $n_i, n_j \in N$, $n_i \stackrel{nticd}{\rightarrow} n_j$ implies $n_i \stackrel{ntscd^*}{\rightarrow} n_j$.*

Note that this result is supported by the examples in Tables I and II. For example, in Figure 1(a), $a \stackrel{nticd}{\rightarrow} d$ holds but $a \stackrel{ntscd}{\rightarrow} d$ does not. Nonetheless, $a \stackrel{ntscd^*}{\rightarrow} d$ holds, as both $a \stackrel{ntscd}{\rightarrow} c$ and $c \stackrel{ntscd}{\rightarrow} d$ hold.

PROOF. Our assumption is that n_i has successors n_k, n_l such that: **(i)** n_j occurs on all sink-bounded paths from n_k; and **(ii)** there exists a sink-bounded path from n_l on which n_j does not occur.

Now consider a sink-bounded path π from n_i via n_k (there exists such a path, by Lemma 1). We can write $\pi = [u_0, u_1, \ldots, u_m, \ldots]$, where $m \geq 1$, $u_0 = n_i$, $u_1 = n_k$, $u_m = n_j$, $u_p \neq n_j$ for $1 \leq p < m$. Observe that for all $p \in \{1 \ldots m\}$, n_j occurs on all sink-bounded paths from u_p (otherwise **(i)** would be contradicted). So, if all sink-bounded paths from n_l would contain u_p, all sink-bounded paths from n_l would contain n_j, contradicting **(ii)**. Thus for all $p \in \{1 \ldots m\}$, there exists a sink-bounded path from n_l not containing u_p.

Now define predicates Q_p such that $Q_p(r)$ holds iff $0 \leq r \leq p \leq m$ and all maximal paths from u_r contain u_p. Observe that if $Q_p(r)$ does not hold but $Q_p(r+1)$ does, then $u_r \stackrel{ntscd}{\rightarrow} u_p$ (compare with Definition 16). Also observe that $Q_p(p)$ holds for all $p \leq m$, but if $u_p \neq u_0$, then $Q_p(0)$ does not hold (for if all maximal paths from u_0 contain u_p, then all maximal paths from n_l contain u_p so also all sink-bounded paths from n_l contain u_p, contradicting the aforementioned).

Now we are ready for the construction: If $u_m = u_0$, we are done. Otherwise, we can find p_1 such that $Q_m(p_1)$ does not hold but $Q_m(p_1 + 1)$ does, showing that $u_{p_1} \stackrel{ntscd}{\rightarrow} u_m$. If $u_{p_1} = u_0$, we are done. Otherwise, since $Q_{p_1}(p_1)$ holds but $Q_{p_1}(0)$ does not, we can find p_2 such that $Q_{p_1}(p_2)$ does not hold but $Q_{p_1}(p_2+1)$ does, showing that $u_{p_2} \stackrel{ntscd}{\rightarrow} u_{p_1}$. Now we can repeat as desired. □

Order dependency is relevant for irreducible graphs only.

LEMMA 3. *For a reducible CFG, the relations $\stackrel{dod}{\rightarrow}$ and $\stackrel{sod}{\rightarrow}$ are empty.*

PROOF. Assume that $n_1 \stackrel{sod}{\rightarrow} n_2 \leftrightharpoons n_3$, or $n_1 \stackrel{dod}{\rightarrow} n_2 \leftrightharpoons n_3$. Thus n_1,n_2,n_3 are distinct, and:

—od(i) All maximal paths from n_1 contain both n_2 and n_3;
—od(ii) in one maximal path from n_1, n_2 occurs before the first occurrence of n_3; and
—od(iii) in one maximal path from n_1, n_3 occurs before the first occurrence of n_2.

We shall show that from these assumptions, a contradiction can be derived when the CFG is reducible. First observe that

$$n_1 \text{ is reachable from neither } n_2 \text{ nor } n_3, \qquad (*1)$$

for otherwise, we could assume without loss of generally that there is a path from n_2 to n_1 not containing n_3, which, by **od(ii)**, entails that there exists a maximal path from n_1 not containing n_3, contradicting **od(i)**.

Since the CFG is assumed reducible, its edges E can be partitioned into forward edges E_f and back edges E_b. Here E_f forms an acyclic graph, so without loss of generality, we can assume that n_2 is *not* reachable in E_f from n_3. Since by **od(iii)** and **od(i)**, n_2 is reachable in E from n_3, there exists a node n_4 and

$$\text{in } E_f\text{, a path } [n_3..n_4] \text{ not containing } n_2. \qquad (*2)$$

and a back edge

$$n_4 \to n_5\text{, where } n_5 \text{ dominates } n_4. \qquad (*3)$$

With n_0 the start node of the CFG, due to (*1) there exists

$$\text{a path } [n_0..n_1] \text{ not containing } n_2. \qquad (*4)$$

Also, by assumption **od(iii)**, there exists

$$\text{a path } [n_1..n_3] \text{ not containing } n_2. \qquad (*5)$$

From (*4), (*5), and (*2), we see that there is

$$\text{a path } [n_0..n_4] \text{ containing } n_1 \text{ but not containing } n_2. \qquad (*6)$$

By (*3) we infer that n_5 is on that path, and that there is a path from n_4 to n_4 not containing n_2. Thus we can construct a maximal path from n_1 not containing n_2, contradicting **od(i)**. □

Observables. For the (bisimulation-based) correctness proof in Section 5.1, we shall need a few results about slice sets, the members of which are termed "observable." Typically, these results require slice sets Ξ to be closed under nontermination-sensitive control dependency, that is, if $n_1 \stackrel{ntscd}{\to} n_2$ and $n_2 \in \Xi$, then also $n_1 \in \Xi$. For certain weaker results, it is sufficient to demand that Ξ is closed under nontermination-insensitive control dependency, that is, if $n_1 \stackrel{nticd}{\to} n_2$ and $n_2 \in \Xi$, then also $n_1 \in \Xi$ (by Theorem 4, the latter closedness property is weaker than the former). For the main result (i.e., Theorem 5), we shall also demand Ξ to be closed under (decisive) order dependency, namely, if $n_i \stackrel{dod}{\to} n_j \leftrightharpoons n_k$ with $n_j, n_k \in \Xi$, then also $n_i \in \Xi$.

A key feature of our development is the notion of "first observable," where we now present a "may" definition.

Definition 17. For a node n, $obs^1_{may}(n)$ is the set of nodes $n' \in \Xi$ with the property that there exists a path $[n..n']$, where all nodes except n' are not in Ξ.

Clearly, if $n \in \Xi$, then $obs^1_{may}(n) = \{n\}$. Next, we present a "must" definition of "subsequent observable."

Definition 18. For a node n, $obs^*_{must}(n)$ is the set of nodes $n' \in \Xi$ with the property that all maximal paths from n contain n'.

A crucial property of a slice set is that "may" implies "must," that is, the first observable on any path will be encountered sooner or later on all other paths.

Lemma 4. *Assume the node set Ξ is closed under $\stackrel{ntscd}{\rightarrow}$. Then for all nodes n, $obs^1_{may}(n) \subseteq obs^*_{must}(n)$.*

Proof. Assume, in order to arrive at a contradiction, that there exists a node n_0 such that $obs^1_{may}(n_0)$ is not a subset of $obs^*_{must}(n_0)$; thus, there exists $n_1 \in \Xi$ with $n_1 \in obs^1_{may}(n_0)$, but $n_1 \notin obs^*_{must}(n_0)$. The situation is that there is a path π from n_0 to n_1 where all nodes except n_1 do not belong to Ξ. We infer that $n_0 \notin \Xi$, as otherwise we would have $n_1 = n_0$, contradicting $n_1 \notin obs^*_{must}(n_0)$. We define a predicate Q such that

$$Q(n) \text{ holds iff } n_1 \in obs^*_{must}(n).$$

By our assumption, $Q(n_0)$ does not hold; clearly, $Q(n_1)$ holds. Therefore, π can be written as $[n_0..n_2n_3..n_1]$ where $Q(n_2)$ does not hold but $Q(n_3)$ does (i.e., there is an edge from n_2 to n_3; note that n_2 may equal n_0 and that n_3 may equal n_1, but we know that $n_1 \neq n_2$).

We shall show that $n_2 \stackrel{ntscd}{\rightarrow} n_1$; then from $n_1 \in \Xi$ and from Ξ being closed under $\stackrel{ntscd}{\rightarrow}$, we get $n_2 \in \Xi$, which contradicts n_1 being the only node in π that is also in Ξ.

Since $Q(n_2)$ does not hold, there exists a maximal path starting at n_2 not containing n_1; that path has to have at least two elements (since n_2 has an outgoing edge) and the second element cannot be n_3 (as $Q(n_3)$ holds). Therefore, the second element is some node n_4 with $n_3 \neq n_4$, and there exists a maximal path from n_4 which does not contain n_1. Our final obligation (compare with Definition 16) is to prove that all maximal paths from n_3 contain n_1, which follows since $Q(n_3)$ holds. □

In a similar way we can show

Lemma 5. *Assume that Ξ is closed under $\stackrel{nticd}{\rightarrow}$. Assume $n_1 \in obs^1_{may}(n_0)$. Then all sink-bounded paths from n_0 will contain n_1.*

As a consequence we have the following result, giving conditions to preclude the existence of infinite unobservable paths.

Lemma 6. *Assume that $n_0 \notin \Xi$, but that there is a path π starting at n_0 which contains a node in Ξ.*

—*If Ξ is closed under nontermination-insensitive control dependency, then all sink-bounded paths starting at n_0 will reach Ξ.*

—*If Ξ is also closed under nontermination-sensitive control dependency, then all maximal paths starting at n_0 will reach Ξ.*

Our main result, Theorem 5 given next, states that from a given node there is a unique first observable. This does not hold without extra assumptions, however, as demonstrated by the (irreducible) CFG in Figure 2(b), where $\Xi = \{b, c\}$ is closed under nontermination-sensitive control dependency (since $a \overset{ntscd}{\not\to} b$ and $a \overset{ntscd}{\not\to} c$) and provides a with *two* possible first observables. Our remedy is to demand that the slice set Ξ be closed under decisive order dependency, as defined in Definition 12. Recall (Lemma 3) that a reducible graph is vacuously closed under decisive order dependency.

THEOREM 5. *If Ξ is closed under $\overset{ntscd}{\to}$ and $\overset{dod}{\to}$, then for all nodes n it holds that $obs^1_{may}(n)$ is at most a singleton.*

PROOF. Assume the contrary, and let n_0 be such that $|obs^1_{may}(n_0)| > 1$, implying (by Lemma 4) that $|obs^*_{must}(n_0)| > 1$. Then there cannot exist a maximal path π from n_0 such that $|obs^1_{may}(n)| > 1$ holds for all n occurring in π, for then π would contain no nodes in Ξ, contradicting that $obs^*_{must}(n_0)$ must be nonempty. Thus, there exists a node n_1 such that $|obs^1_{may}(n_1)| > 1$, and hence $n_1 \notin \Xi$, but for all n which are successors of n_1, $obs^1_{may}(n)$ is (at most) a singleton. Since $|obs^1_{may}(n_1)| > 1$, we can find $n_2, n_3 \in obs^1_{may}(n_1)$ with $n_2 \neq n_3$. Clearly, n_1 has a successor u_2 with $obs^1_{may}(u_2) = \{n_2\}$, and a successor u_3 with $obs^1_{may}(u_3) = \{n_3\}$.

We shall now argue that $n_1 \overset{dod}{\to} n_2 \leftrightharpoons n_3$, which, since Ξ is closed under $\overset{dod}{\to}$ and since $n_2, n_3 \in \Xi$, will imply $n_1 \in \Xi$. This yields the desired contradiction. Looking at Definition 12, we see that for reasons of symmetry, it is sufficient to show the following items:

—From n_1, all maximal paths contain n_2. This follows since $n_2 \in obs^1_{may}(n_1) \subseteq obs^*_{must}(n_1)$; and

—from a successor of n_1, all maximal paths contain n_2 before n_3. Such a successor is u_2 since $obs^1_{may}(u_2) = \{n_2\}$, so there is no way that a path from u_2 can contain n_3 before n_2. □

5. SLICING

We now describe how to slice a CFG G with respect to a slice set S_C, the smallest set containing C which is closed under data dependence $\overset{dd}{\to}$ and also under $\overset{ntscd}{\to}$ and under $\overset{dod}{\to}$.

Definition 19 (*Slicing Transformation*). The result of slicing is a program with the same CFG as the original, but with the code map $code_1$ replaced by $code_2$. Here for $n \in S_C$, we have $code_2(n) = code_1(n)$, and for $n \notin S_C$, we have:

—If n is a statement node, then $code_2(n)$ is the statement `skip`; and

—if n is a predicate node, then $code_2(n)$ is `cskip`, the semantics of which is that it nondeterministically chooses one of its successors.

The preceding definition is conceptually simple so as to facilitate the correctness proofs. Of course, one would want to do some postprocessing, like eliminating `skip` commands and `cskip` commands where the two successor nodes are equal; we shall not address this issue further, but remark that most such transformations are trivially meaning preserving.

5.1 Correctness Properties

The main intuition behind our notion of slicing correctness is that the nodes in a slicing criterion C represent "observations" that one is making about a CFG G under consideration. Specifically, for an $n \in C$, one can observe that n has been executed and also observe the values of any variables referenced at n. Execution of nodes not in C corresponds to *silent moves*, or nonobservable actions. The slicing transformation should preserve the behavior of the program with respect to C-observations, but parts of the program that are irrelevant with respect to computing C observations can be "sliced away." The slice set S_C built according to Definition 4 represents the nodes that are relevant for maintaining the observations C. Thus, to prove the correctness of slicing, we will establish the stronger result that G will have the same S_C-observations with respect to the original code map $code_1$ as with respect to the sliced code map $code_2$, and this will imply that they have the same C-observations.

The previous discussion suggests that appropriate notions of correctness for slicing reactive programs can be derived from the notion of weak bisimulation found in concurrency theory, where a transition may include a number of τ-moves [Milner 1989]. Recall from Section 2.2 that a state s is a pair (n, σ), where σ is a store mapping variables into values.

Definition 20. For $i = 1, 2$ we write:

—$i \vdash s \rightarrow s'$ to denote that with respect to code map $code_i$, the program state s rewrites in one step to s';

—$i \vdash s \overset{n}{\longmapsto} s'$ if $i \vdash s \rightarrow s'$ and $n \in \Xi$, where $s = (n, \sigma)$;

—$i \vdash s \overset{\tau}{\longmapsto} s'$ if $i \vdash s \rightarrow s'$ and $n \notin \Xi$, where $s = (n, \sigma)$;

—$\overset{\tau}{\Longrightarrow}$ for the reflexive transitive closure of $\overset{\tau}{\longmapsto}$; and

—$i \vdash s \overset{n}{\Longrightarrow} s'$ if there exists s_1 such that $s \overset{\tau}{\Longrightarrow} s_1$ and $s_1 \overset{n}{\longmapsto} s'$.

Definition 21 (*Weak Bisimulation*). A binary relation ϕ is a weak bisimulation if for all $i \in \{1, 2\}$, we have the following properties where $j = 3 - i$:

(1) If $s_1 \,\phi\, s_2$ and $i \vdash s_i \overset{\tau}{\longmapsto} s_i'$, then there exists s_j' such that $j \vdash s_j \overset{\tau}{\Longrightarrow} s_j'$ and $s_1' \,\phi\, s_2'$.

(2) If $s_1 \,\phi\, s_2$ and $i \vdash s_i \overset{n}{\longmapsto} s_i'$, then there exists s_j' such that $j \vdash s_j \overset{n}{\Longrightarrow} s_j'$ and $s_1' \,\phi\, s_2'$.

Remark 2. The notion of weak bisimulation just defined is slightly different from what is mostly seen in the literature in that $\overset{n}{\Longrightarrow}$ does not allow silent moves *after* the observable action.

Remark 3. If Ξ is closed under $\stackrel{ntscd}{\rightarrow}$ and $\stackrel{dod}{\rightarrow}$, we know from Theorem 5 that for any node n, $obs^1_{may}(n)$ is either a singleton set or empty. With abuse of notation, we shall write $obs^1_{may}(n) = n_1$ for $obs^1_{may}(n) = \{n_1\}$. Also, we know from Lemma 4 that if $obs^1_{may}(n) = n_1$, then all maximal paths from n will contain n_1.

Definition 22 (*Relevant Variables*). For each node n in G, we define $relv(n)$, namely the set of relevant variables at n, by stipulating that x is in $relv(n)$ iff there exists a node $n_k \in \Xi$ and a path π from n to n_k such that $x \in ref(n_k)$, but for all nodes n_j occurring before n_k in π, $x \notin def(n_j)$.

Strictly speaking, we should have defined (for i = 1,2) functions $ref_i(n)$ to return the variables referenced at node n with respect to code map $code_i$, functions $def_i(n)$ to return the variables defined at node n with respect to code map $code_i$, and functions $relv_i(n)$ and relation $\stackrel{dd}{\rightarrow}_i$ parametrized with respect to ref_i and def_i. However, the following result shows that we can safely ignore the subscripts, since the slicing transformation applied to S_C yields a node set that is also closed under data dependence and that has the same set of relevant variables for each node.

LEMMA 7. *Assume, with $\stackrel{dd}{\rightarrow}_i$ etc., as defined just before, that Ξ is closed under $\stackrel{dd}{\rightarrow}_1$. Then:*

(1) *Ξ is closed also under $\stackrel{dd}{\rightarrow}_2$; and*
(2) *for all n, $relv_1(n) = relv_2(n)$.*

PROOF. To show item (1), assume the contrary; then there exists a path π from $n_j \notin \Xi$ to $n_k \in \Xi$ such that $x \in ref_2(n_k)$ and $x \in def_2(n_j)$, but for all n' interior in π: $x \notin def_2(n')$. Observing that all variables in $code_2$ also occur in $code_1$, we see that $x \in ref_1(n_k)$ and $x \in def_1(n_j)$. Since Ξ is closed under $\stackrel{dd}{\rightarrow}_1$, we can infer that there exists a node n' interior in π with $x \in def_1(n')$; let n_1 be the last such n'. Since Ξ is closed under $\stackrel{dd}{\rightarrow}_1$, we infer that $n_1 \in \Xi$ and therefore $code_1(n_1) = code_2(n_1)$. But since $x \in def_1(n_1)$ and (compare with the preceding) $x \notin def_2(n_1)$, this yields the desired contradiction.

To show item (2), assume that $x \in relv_i(n)$ with $i \in \{1, 2\}$; we must prove that $x \in relv_j(n)$, where $j = 3 - i$. Our assumptions are that there exists a path π from n to $n_k \in \Xi$ such that $x \in ref_i(n_k)$, but for all nodes n' occurring before n_k in π, $x \notin def_i(n')$. Now, since $n_k \in \Xi$, $code_i(n_k) = code_j(n_k)$, so $x \in ref_j(n_k)$. We are done if we can prove that $x \notin def_j(n')$ for all nodes n' occurring before n_k in π. In order to arrive at a contradiction, assume that this is not the case. Let n_1 be the last node n' occurring before n_k in π with $x \in def_j(n')$. Then $n_1 \stackrel{dd}{\rightarrow}_j n_k$, since $n_k \in \Xi$, which by item (1) is closed under $\stackrel{dd}{\rightarrow}_j$; this implies $n_1 \in \Xi$. But then $code_j(n_1) = code_i(n_1)$, which gives the desired contradiction since $x \in def_j(n_1)$ but $x \notin def_i(n_1)$. □

After this digression, we return the the main development, where a key property is that the set of relevant variables is determined by the first observable.

LEMMA 8. *Assume that Ξ is closed under $\stackrel{ntscd}{\rightarrow}$, $\stackrel{dod}{\rightarrow}$, and $\stackrel{dd}{\rightarrow}$. Assume that n_1 and n_2 are such that $obs^1_{may}(n_1) = obs^1_{may}(n_2)$. Then $relv(n_1) = relv(n_2)$.*

PROOF. If $obs^1_{may}(n_1)$ and $obs^1_{may}(n_2)$ are both empty, no node in Ξ is reachable from n_1 nor from n_2, and therefore $relv(n_1) = relv(n_2) = \emptyset$.

Otherwise, let $n_3 = obs^1_{may}(n_1) = obs^1_{may}(n_2)$; for reasons of symmetry, it is sufficient to prove that $relv(n_1) \subseteq relv(n_2)$. So let $x \in relv(n_1)$ be given, we must prove $x \in relv(n_2)$. There exists a path π from n_1 to $n_k \in \Xi$ such that $x \in ref(n_k)$, but $x \notin def(n_j)$ for any node n_j occurring before n_k in π. Since $n_3 = obs^1_{may}(n_1)$, we can split π into $\pi_1 = [n_1..n_3]$ and $\pi_0 = [n_3..n_k]$. Since $n_3 = obs^1_{may}(n_2)$, there exists a repetition-free path $\pi_2 = [n_2..n_3]$, and thus a path $\pi' = \pi_2\pi_0$ from n_2 to n_k. Towards proving our goal $x \in relv(n_2)$, we are left to show that $x \notin def(n_j)$ for all nodes n_j occurring before n_k in π'. Assume the contrary, and let n' be the last node in π' serving as a counterexample. Since Ξ is closed under $\stackrel{dd}{\rightarrow}$, we infer that $n' \in \Xi$. Also, due to the properties of π, we infer that n' does not occur in π_0, and therefore n' occurs before n_3 in π_2. But this contradicts $n_3 = obs^1_{may}(n_2)$. □

We need one more auxiliary result.

LEMMA 9. *Assume that Ξ is closed under $\stackrel{ntscd}{\rightarrow}$, $\stackrel{dod}{\rightarrow}$, and $\stackrel{dd}{\rightarrow}$. If $i \vdash s_1 \stackrel{\tau}{\longmapsto} s_2$, where $s_1 = (n_1, \sigma_1)$, $s_2 = (n_2, \sigma_2)$, and $i \in \{1, 2\}$, then:*

(1) *$obs^1_{may}(n_1) = obs^1_{may}(n_2)$; and*
(2) *there exists a set of variables V such that;*
  (a) *$V = relv(n_1) = relv(n_2)$; and*
  (b) *$\sigma_1 =_V \sigma_2$.*

Here we write $\sigma_1 =_V \sigma_2$ when for all $x \in V$, $\sigma_1(x) = \sigma_2(x)$.

PROOF. First observe that $n_1 \notin \Xi$. For item (1), clearly $obs^1_{may}(n_2) \subseteq obs^1_{may}(n_1)$, so by Theorem 5 it is sufficient to prove that it cannot be the case that $obs^1_{may}(n_2) = \emptyset$ while $obs^1_{may}(n_1)$ is a singleton $\{n_3\}$. But if so, then Lemma 4 would tell us that $n_3 \in \Xi$ occurs on all maximal paths from n_1, and thus also on all maximal paths from n_2, contradicting that $obs^1_{may}(n_2) = \emptyset$.

Now (a) follows from Lemma 8. For (b), in order to arrive at a contradiction, we assume that $\sigma_1 =_V \sigma_2$ does not hold. For this to be the case, there must exist $x \in V$ with $x \in def(n_1)$. Since $x \in relv(n_1)$, there exists a path from n_1 to a node $n_k \in \Xi$ with $x \in ref(n_k)$ along which x is not defined. But since x is defined at n_1, this yields the desired contradiction. □

We now stipulate when a program state in the original program is related to a program state in the sliced one.

Definition 23 (*Bisimilar Relation*). For Ξ closed under $\stackrel{ntscd}{\rightarrow}$, $\stackrel{dod}{\rightarrow}$, and $\stackrel{dd}{\rightarrow}$, we define a relation $R : s_1\, R\, s_2$ iff:

—$obs^1_{may}(n_1) = obs^1_{may}(n_2)$; and
—$\sigma_1 =_V \sigma_2$,

where $s_1 = (n_1, \sigma_1)$, $s_2 = (n_2, \sigma_2)$, and $V = relv(n_1) = relv(n_2)$.

By Lemma 8, this is well defined. We now state the key part of the correctness result.

THEOREM 6. *If Ξ is closed under $\stackrel{ntscd}{\to}$, $\stackrel{dod}{\to}$, and $\stackrel{dd}{\to}$, then the relation R from Definition* 23 *is a weak bisimulation (compare with Definition* 21*).*

PROOF. For $i \in \{1, 2\}$ and $j = 3 - i$, we must show that:

(1) If $s_1 \, R \, s_2$ and $i \vdash s_i \stackrel{\tau}{\longmapsto} s_i'$, then there exists s_j' such that $j \vdash s_j \stackrel{\tau}{\Longrightarrow} s_j'$ and $s_1' \, R \, s_2'$.

(2) If $s_1 \, R \, s_2$ and $i \vdash s_i \stackrel{n}{\longmapsto} s_i'$, then there exists s_j' such that $j \vdash s_j \stackrel{n}{\Longrightarrow} s_j'$ and $s_1' \, R \, s_2'$.

For item (1), assume that $i \vdash s_i \stackrel{\tau}{\longmapsto} s_i'$. Choose $s_j' = s_j$. The claim then trivially follows from Lemma 9.

For item (2), assume that $i \vdash s_i \stackrel{n}{\longmapsto} s_i'$. Thus s_i is of the form (n, σ_i); also let $s_j = (n_j, \sigma_j)$ and $s_i' = (n', \sigma_i')$. We have $n = obs^1_{may}(n) = obs^1_{may}(n_j)$; let $V = relv(n) = relv(n_j)$. Since by Lemma 4, $n \in obs^*_{must}(n_j)$, any execution sequence starting from n_j will sooner or later hit n; also, since n is the only node in $obs^1_{may}(n_j)$, that execution sequence will contain no other nodes in Ξ. All this shows that there exists $s_j'' = (n, \sigma_j'')$ such that $j \vdash s_j \stackrel{\tau}{\Longrightarrow} s_j''$. By repeated application of Lemma 9 we infer that $\sigma_j'' =_V \sigma_j$ and since $\sigma_i =_V \sigma_j$, thus also $\sigma_i =_V \sigma_j''$. In particular,

$$\sigma_i \text{ and } \sigma_j'' \text{ agree on } ref(n). \tag{*}$$

Therefore, s_j'' will choose the same branch as s_i (if n is a predicate node, otherwise vacuously). In other words, there exists s_j' of the form (n', σ_j') such that $j \vdash s_j'' \stackrel{n}{\longmapsto} s_j'$ and thus $j \vdash s_j \stackrel{n}{\Longrightarrow} s_j'$. We are left to show that with $V' = relv(n')$, we have $\sigma_i' =_{V'} \sigma_j'$. So let $x \in V'$, we must prove $\sigma_i'(x) = \sigma_j'(x)$. If $x \in def(n)$ (and n is thus a statement node), then the claim clearly follows from (*). Otherwise, if $x \notin def(n)$, then $x \in relv(n) = V$ and the claim follows from $\sigma_i =_V \sigma_j''$, since $\sigma_i'(x) = \sigma_i(x) = \sigma_j''(x) = \sigma_j'(x)$. □

Observe that R is reflexive. Therefore, by Theorem 6, the initial state of the original CFG is weakly bisimilar to that of the sliced CFG. Also, since two states that are related by R produce the same "output," and since bisimulation generalizes Weiser's notion of projection [Weiser 1984] to infinite traces, this demonstrates that if Ξ is closed under $\stackrel{ntscd}{\to}$, $\stackrel{dod}{\to}$, and $\stackrel{dd}{\to}$, then the sliced program has the same "observable behavior" as the original.

Let us elaborate on the preceding argument, and informally argue why Theorem 6 entails the "standard" [Ball and Horwitz 1993] way of phrasing correctness of slicing, that is, the sequence of observed values (i.e., values of the variables referenced) at each node in the slicing criterion C is the same for the original as for the sliced program. Let n_1 be the first S_C-node hit by executing the original CFG, then Theorem 6 tells us that n_1 will also be the first S_C-node hit by executing the sliced CFG, and vice versa. Moreover, the two executions will agree on the values of relevant variables. Repeating the argument, we can show that if n_2 is the second S_C-node hit by executing the original CFG, then n_2 is also the second S_C-node hit by executing the sliced CFG, and the two executions will agree on the values of relevant variables. Repeating as desired, we conclude that for each node in S_C, the sequence of values of relevant variables is the same for the original as for the sliced program. Since S_C includes C, and since a referenced variable is relevant, we have as a special case the desired result: For each node in C, the sequence of observed values is the same for the original as for the sliced program.

6. ALGORITHMS

In this section we present algorithms to calculate various forms of control and order dependences that were presented earlier. Each algorithm is accompanied by an overview, a proof of correctness, and the complexity analysis of the worst-case time requirement. The algorithms are presented to suggest that the proposed dependences can be calculated by algorithms with time complexity that is polynomial in the number of nodes/edges. We conjecture that more optimal algorithms can be designed to calculate the same information.

6.1 Nontermination-Sensitive Control Dependence (NTSCD)

We adopt an approach similar to symbolic data flow analysis to calculate control dependences. Basically, control dependences are determined by reasoning about properties of sets of CFG paths; those sets are represented symbolically in our algorithm. Specifically, for each node n with more than one successor in G, the set of all maximal paths that start with $n \rightarrow m$ is represented by a symbol t_{nm}. The algorithm propagates these symbols to collect the effects of particular control flow choices at program points in the CFG. For each node p, a set of symbols S_{pn} is maintained for every node n in the CFG that has more than one successor; these sets record the maximal paths that originate from n and contain p. Hence, based on the interpretation, $t_{nm} \in S_{pn}$ indicates that *all maximal paths starting with* $n \rightarrow m$ *contain* p. We shall use T_n to denote the number of successors ($|succs(n, G)|$) of node n in G. Also, $condNodes(G)$ denotes the set of nodes in G that have multiple successors. The algorithm is presented in Figure 4.

6.1.1 *Proof of Correctness.* The correctness of the algorithm (Figure 4) is presented as the following theorem.

THEOREM 7. *Upon the termination of phase* (2) *of the algorithm,* $t_{nm} \in S_{pn}$ *iff all maximal paths starting with* $n \rightarrow m$ *contain* p.

PROOF. We shall use the "only if" direction as an invariant on the loops in phase (2). We shall then prove the "if" direction via contradiction.

"Only if" direction. The finiteness of N ensures the termination of phase (1). Upon completion of phase (1), the invariant is trivially established at the

```
NONTERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n0) :  a control flow graph
 2  S[|N|, |N|] :  a matrix of sets where S[p, n] represents S_pn
 3  CD[|N|] :  a sequence of sets
 4  workbag :  a set of nodes
 5
 6   # (1) Initialize
 7  workbag ← ∅
 8  for  each n in condNodes(G)
 9  do for  each m in succs(n, G)
10       do S[m, n] ← {t_nm}
11          workbag ← workbag ∪ {m}
12
13   # (2) Calculate all-path reachability
14  while workbag ≠ ∅
15  do n ← remove(workbag)
16       # (2.1) One successor case
17       if T_n = 1 and n ∉ succs(n, G)
18         then m ← select(succs(n, G))
19                for p in condNodes(G)
20                do if S[n, p]\S[m, p] ≠ ∅
21                     then S[m, p] ← S[m, p] ∪ S[n, p]
22                          workbag ← workbag ∪ {m}
23       # (2.2) Multiple successors case
24       if |succs(n, G)| > 1
25         then for m in N
26                do if |S[m, n]| = T_n
27                     then for p ∈ condNodes(G)\{n}
28                            do if S[n, p]\S[m, p] ≠ ∅
29                                 then S[m, p] ← S[m, p] ∪ S[n, p]
30                                      workbag ← workbag ∪ {m}
31   # (3) Calculate non-termination sensitive control dependence
32  for  each n in N
33  do for  each m in condNodes(G)
34       do if 0 < |S[n, m]| < T_m
35            then CD[m] ← CD[m] ∪ {n}
36
37  return CD
```

Fig. 4. Algorithm to calculate nontermination-sensitive control dependence.

beginning of phase (2). If n has only one successor m, then all maximal paths containing n will contain m. Hence, the assignment at line 21 establishes the invariant at the end of the loop at line 19 (and the conditional at line 17). If n has multiple successors and all maximal paths through the successors contain m, then all maximal paths containing n will also contain m. This is captured by the assignment at line 29 and the invariant is established at the end of the loops at lines 25 and 27.

As the graph has a finite number of nodes, the number of successors of a node is finite. Hence, the total number of symbols (t_{nm}) in G is finite as well. This implies that the size of S_{nm} has a finite bound for every pair of nodes n and m. In each iteration of the `while` loop at line 14, either a symbol set S_{nm} increases in size or all of the symbol sets remain unchanged. The former case contributes an iteration (lines 22 and 30). As the size of the symbol set is finitely bound, the `while` loop in line 14 will terminate establishing the "only if" direction.

"If" direction. Suppose there are nodes n, m, and p such that all maximal paths starting with $n \rightarrow m$ contain p, but $t_{nm} \notin S_{pn}$. This implies that in every maximal path starting with $n \rightarrow m$, ending with p, and containing nodes q and r (in the given order), $t_{nm} \in S_{in}$ for every node from m to q and $t_{nm} \notin S_{jn}$ for every node from r to p. We consider two cases.

—*r is the only successor of q*. In this case, when t_{nm} is injected into S_{qn}, q will be marked for processing (line 21). Upon processing, t_{nm} will be injected into S_{rn}. Hence, the supposition cannot be true.

—*q has multiple successors*. By the supposition, there should be a first common node to occur on all maximal paths originating from the successors of q. Let r be this common node. Also assume that there are no conditional nodes in the paths from q to r. From the previous clause of the proof and the nonbranching property of the paths between q and r, $|S_{rq}| = T_q$. This implies $S_{qn} \subseteq S_{rn}$, hence the supposition is falsified. When nested conditional nodes occur on the paths from q to r, similar reasoning can be applied to conditional nodes in decreasing order of nesting.

The preceding reasoning can be applied inductively when r is not the immediate successor of q, or when r is not the first common node to occur on all maximal paths originating from the successors of q. □

Based on the interpretation attached to t_{mn} and S_{pn} and Theorem 7, it is trivial to see that phase (3) correctly calculates nontermination-sensitive control dependence.

6.1.2 *Complexity Analysis.* Phases (1) and (3) of the algorithm (see Figure 4) have a worst-case complexity of $O(|N|^2 \times \lg|N|)$, where $\lg|N|$ is the complexity of set operations. The complexity of the loop at line 25 is $O(|N|^2 \times \lg|N|)$ and it dominates the complexity of the loop at line 14. In the worst case in phase (2), for a node p, all token sets $S[p, i]$ of p will stabilize in $\sum T_n$ iterations. Hence, the overall complexity of phase (2) will be $O(\sum T_n \times |N|^3 \times \lg(|N|)$. This will also be the overall complexity of the algorithm.

6.2 Nontermination-Insensitive Control Dependence (NTICD)

The proposed algorithm (see Figure 5) to calculate nontermination-insensitive control dependence is very similar to the NTSCD algorithm. The only differences are the presence of phase (2.3) and the interpretation attached to t_{nm}. In the NTSCD algorithm, any token t_{nm} injected into S_{nn} is not propagated to non-m successors of n, hence preserving nontermination sensitivity. Phase (2.3) in the NTICD algorithm induces nontermination insensitivity by undoing this preservation. Also, t_{nm} represents all extensible finite paths[9] starting with $n \rightarrow m$ in the NTICD algorithm.

[9]A finite path is *extensible* if it can be extended by adding an edge.

```
NONTERMINATION-INSENSITIVE-CONTROL-DEPENDENCE(G)
G(N, E, n0) :  a control flow graph
S[|N|, |N|] :  a matrix of sets where S[p, n] represents S_pn
CD[|N|] :  a sequence of sets
workbag :  a set of nodes

 # (1) Initialize
workbag ← ∅
for  each n in condNodes(G)
do for  each m in succs(n, G)
     do S[m, n] ← {t_nm}
          workbag ← workbag ∪ {m}

 # (2) Calculate all-path reachability
while workbag ≠ ∅
do n ← remove(workbag)
      # (2.1) One successor case
     if T_n = 1 and n ∉ succs(n, G)
        then m ← select(succs(n, G))
                for p in condNodes(G)
                do if S[n, p]\S[m, p] ≠ ∅
                        then S[m, p] ← S[m, p] ∪ S[n, p]
                                workbag ← workbag ∪ {m}
      # (2.2) Multiple successors case
     if |succs(n, G)| > 1
        then for m in N
                do if |S[m, n]| = T_n
                        then for p in condNodes(G)\{n}
                                do if S[n, p]\S[m, p] ≠ ∅
                                        then S[m, p] ← S[m, p] ∪ S[n, p]
                                                workbag ← workbag ∪ {m}
      # (2.3) Erase non-termination sensitivity
     if |S[n, n]| > 0
        then for m in succs(n, G)\n
                do if S[n, n]\S[m, n] ≠ ∅
                        then S[m, n] ← S[m, n] ∪ S[n, n]
                                workbag ← workbag ∪ {m}

 # (3) Calculate non-termination insensitive control dependence
for  each n in N
do for  each m in condNodes(G)
     do if 0 < |S[n, m]| < T_m
             then CD[m] ← CD[m] ∪ {n}

return CD
```

Fig. 5. Algorithm to calculate nontermination-insensitive control dependence.

6.2.1 *Proof of Correctness.* Given the similarity of the NTSCD and NTICD algorithms, we prove the correctness of the latter by proving that phase (3), along with the interpretation attached to t_{nm}, calculates nontermination-insensitive control dependence.

The key observation is that phase (2.3) induces nontermination insensitivity. Succinctly, if $t_{nm} \in S_{nn}$, then t_{nm} is added to S_{pn}, where p is a successor of n.

This establish that all finite paths that start with $n \rightarrow m$ and reach n can be finitely extended to reach p, hence inducing nontermination insensitivity.

LEMMA 10. *If $t_{nm} \in S_{pn}$ and p belongs to a control sink, then for all nodes $q \in c\text{-}sink(p).t_{nm} \in S_{qn}$, where $c\text{-}sink(p)$ is the set of nodes in the control sink containing p.*

PROOF. If $|c\text{-}sink(p)| \leq 1$, then we are done. If $|c\text{-}sink(p)| > 1$, then let q be a node such that $q \in c\text{-}sink(p)$ and $t_{nm} \notin S_{qn}$. Since q and p belong to the same control sink, every finite path from p can be extended to q (since there is a path between any two nodes belonging to the same SCC and every control sink is a SCC). Hence $S_{pn} \subseteq S_{qn}$. Similarly, we can prove $S_{qn} \subseteq S_{pn}$. Hence $S_{pn} = S_{qn}$. □

THEOREM 8. *Phase* (3) *of NTICD calculates nontermination-insensitive control dependence.*

PROOF. $t_{nm} \in S_{pn}$ implies that all finite paths starting with $n \rightarrow m$ can be extended to p. Hence, $0 < |S_{mn}| < T_n$ implies that there are some successors m of n for which all finite paths starting at m can be extended to p, while for some successors q, not all finite paths starting at q can be extended to p. Hence, $n \overset{ntscd}{\rightarrow} p$.

When $|S_{pn}| = 0$ or $|S_{pn}| = T_n$, this implies that for all successors of n, either none or all finite paths can be extended to contain p. Hence $n \overset{ntscd}{\not\rightarrow} p$. Also, by Lemma 10, $|S_{pn}| = T_n$ for all conditional nodes n in the control sink of p, hence $n \overset{ntscd}{\not\rightarrow} p$.

So, phase (3) correctly calculates nontermination-insensitive control dependence. □

6.2.2 *Complexity Analysis.* Phase (2.3) of the NTICD algorithm contributes $O(\sum T_n \times |N| \times \lg(|N|))$ to the overall complexity of phase (2) of the NTSCD algorithm. As $O(\sum T_n \times |N|^3 \times \lg(|N|)$ dominates $O(\sum T_n \times |N| \times \lg(|N|)$, the overall complexity of the NTICD algorithm is identical to that of NTSCD algorithm.

6.3 Decisive Control Dependence (DCD)

As Definition 11 implies Definition 7, we calculate decisive control dependence by pruning nontermination-sensitive control dependence. It is evident that clause (2) in Definition 11 is stronger than that in Definition 7. Hence, we use the negative form of clause (2) in Definition 11—namely, for all successors n_l of n_i, there exists a maximal path such that n_j occurs before any occurrence of n_i—to prune nontermination-sensitive control dependence to calculate decisive control dependence.

In the algorithm of Figure 6, t_{nm} represents a path π that starts with $n \rightarrow m$ and is maximal or terminates with n, while $t_{nm} \in S_{pn}$ represents a path starting with $n \rightarrow m$ that can be extended to contain p. In phase (2) of the algorithm, tokens are propagated to calculate reachability between conditional and other

```
DECISIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n0) :  a control flow graph
 2  S[|N|, |N|] :  a matrix of sets where S[n1, n2] represents S_{n1 n2}
 3  T[|N|] :  a sequence of integers where T[n1] denotes T_{n1}
 4  CD[|N|] :  a sequence of sets
 5  workbag :  a set of nodes
 6
 7   # (1) Initialize
 8  workbag ← ∅
 9  for  each n in condNodes(G)
10  do succs ← succs(n, G)
11       for  each m in succs
12       do workbag ← workbag ∪ {m}
13          S[m, n] ← {t_nm}
14
15   # (2) Calculate exists-a-path reachability
16  while workbag ≠ ∅
17  do n ← remove(workbag)
18       for  each m in succs(n, G)
19       do for  each p in condNodes(G)
20          do if S[n, p]\S[m, p] ≠ ∅
21                then S[m, p] ← S[m, p] ∪ S[n, p]
22                     workbag ← workbag ∪ m
23
24   # (3) Calculate decisive control dependence
25  CD ← NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
26  for  each n in N
27  do for  each m in CD[n]
28       do if |S[n, m]| = T_m
29             then CD[n] ← CD[n]\{m}
30
31  return CD
```

Fig. 6. Algorithm to calculate decisive control dependence.

nodes of the G. This information is later used in phase (3) to calculate decisive control dependence.

6.3.1 *Proof of Correctness.* To prove the correctness of the DCD algorithm, it is sufficient to prove that phase (2) of the algorithm calculates reachability between the successors of conditional nodes and other nodes of G.

THEOREM 9. *At the end of phase* (2) *in the DCD algorithm,* $t_{nm} \in S_{pn}$ *iff there exists a path starting with* $n \rightarrow m$ *that can be extended to* p.

PROOF. We shall use the "only if" direction as an invariant on the loop in phase (2). We shall then prove the "if" direction via contradiction.

"Only if" direction. As the number of edges in the G is finite, phase (1) will terminate. The invariant is trivially established at the beginning of phase (2). The loops at lines 18 and 19 extend a path starting with $n \rightarrow m$ and leading to p to every successor q of p, if it has not already been extended. Also, q is queued for processing at line 22. Hence, at the end of the loop, the invariant is established.

Each iteration of the outer `while` loop at line 16 in phase (2) will either result in an increase in size of a symbol set while contributing an iteration, or there will be no change in the data. The size of the symbol sets are finite, as the tokens/symbols in G are finite. Hence, the outer `while` loop in phase (2) will terminate.

"If" direction. Upon termination of phase (2), suppose that there are nodes n, m, and p such that there exists a path starting with $n \rightarrow m$ that contains p but $t_{nm} \notin S_{pn}$. This implies that along a path starting with $n \rightarrow m$ and containing p, there should be two consecutive nodes q and r, in the given order, such that $t_{nm} \in S_{qn}$ and $t_{nm} \notin S_{rn}$. However, this leads to a contradiction, as upon termination of phase (2), the condition on line 20 will evaluate to false for all nodes in G. Hence the supposition cannot be true. □

6.3.2 *Complexity Analysis.* Based on the structure of phase (2), it is trivial to see that the complexity of the DCD algorithm is identical to that of NTSCD algorithm.

6.4 Decisive Order Dependence (DOD)

Given nodes $n = n_1, m = n_2$, and $p = n_3$, we need to check whether the three clauses in the definition of decisive order dependence[10] are satisfied. We can use information from any graph reachability algorithm to check whether m and p satisfy the first clause in Definition 12 (as done in the first and second conjuncts on line 6 of ORDER-DEPENDENCE()).

As for the second and third clauses, we encode the order dependence calculation as a problem of constructing a colored bound directed acyclic graph (DAG). The bounding condition is that outgoing edges of m and p are not explored. The coloring condition contains three parts: (1) m and p are assigned colors white and black, respectively; (2) every node in the DAG is colored white(black) iff all its children are colored white(black); and (3) nodes with children of different colors, all uncolored children, and/or nodes that are sources of back edges are uncolored.

Given such a colored bound DAG rooted at n, it is trivial to observe that for an acyclic graph, a node q will be colored white(black) only if all of its successors are colored white(black). Given the encoding, this implies that all maximal paths from q contain $m(p)$ before any occurrence of $p(m)$. Hence, we can conclude that m and p are decisively order dependent on any node n that has at least one black child and at least one white child.

In the case of a cyclic graph, the source q of a back edge is uncolored, indicating the existence of a maximal path that does not contain $m(p)$. In such cases, given the coloring condition, every ancestor of q will be uncolored, hence falsifying clauses (2) and (3) of Definition 12.

6.4.1 *Proof of Correctness.* Based on the aforementioned description/intuition, we need to prove that the coloring and bounding of the DAG does indeed capture the information required to decide whether $n \stackrel{dod}{\rightarrow} m \leftrightharpoons p$.

[10] In this subsection, we shall refer to decisive order dependence as order dependence.

```
ORDER-DEPENDENCE()
OD[|N|][|N|] :  a matrix that captures order dependence
G(N, E, n0) :  a control flow graph
for each n in condNodes(G)
do for each m in N
    do for each p in N\{m}
        do if REACHABLE(m, p, G) ∧ REACHABLE(p, m, G) ∧ DEPENDENCE(n, p, m, G)
              then OD[m][p] = OD[m][p] ∪ {n}
return OD

DEPENDENCE(n, m, p, G)
color[|N|] :  a sequence of values ranging over {unknown, white, black}
for each q in N
do color[q] ← uncolored
color[m] = white
color[p] = black
visited ← {m, p}
COLORED-DAG(G, n, color, visited)
whiteChild ← false
blackChild ← false
for each q in succs(n, G)
do if color[q] = white
      then whiteChild ← true
   if color[q] = black
      then blackChild ← true
return whiteChild ∧ blackChild

COLORED-DAG(G, n, color, visited)
if n ∉ visited
  then visited ← visited ∪ {n}
       for each q in succs(n, G)
       do COLORED-DAG(G, q, color, visited)
       c ← color[select(succs(n, G))]
       for each q in succs(n, G)
       do if color[q] ≠ c
             then c ← uncolored
                  break
       color[n] ← c
return
```

Fig. 7. Algorithm to calculate decisively strong order dependence. Here, reachable(m, p, G) returns *true* if p is reachable from m in the graph G.

We shall prove the correctness of the algorithm by proving the following theorems.

THEOREM 10. *Given a CFG G, a white node, and a black node,* COLORED-DAG() *creates a colored bound DAG such that:*

(1) *A node is colored white if all its immediate successors are colored white;*
(2) *a node is colored black if all its immediate successors are colored black; and*
(3) *a node is uncolored if all its immediate successors are uncolored, it has at least two children of different colors, or is the source of a back edge in G.*

PROOF. It is trivial to see (by induction) that COLORED-DAG() will visit all unvisited nodes reachable from the given node n as in a depth-first search. As each visited node is recorded in `visited`, the bounding condition is established by the addition of m and p to VISITED at lines 4 and 5 of DEPENDENCE() and maintained by the check at line 1 of COLORED-DAG(). This record keeping, along with the finiteness of nodes in the CFG, ensures the termination of COLORED-DAG().

After every child of node n has been fully explored in the loop at line 3 in COLORED-DAG(), the color of n is determined by the loop at line 6 in the same procedure. The loop will terminate normally only when the color of every child of n is the same as the color of an arbitrarily chosen child at line 5. The abnormal termination of the same loop (via `break`) indicates that there are at least two children of the node that have different colors. In situations where one of the successor q is a visited but partially explored node, the color of q will be uncolored due to initialization at line 3 of DEPENDENCE(). Hence, the loop at line 6 will terminate either abnormally or normally (when every child of n was uncolored) and color n as uncolored. □

LEMMA 11. *In the colored bound DAG constructed by* COLORED-DAG()*, a node n is white(black) if all nodes reachable from n in the DAG are white(black).*

PROOF. Follows trivially from the first and second clauses of Theorem 10. □

THEOREM 11. *Given a colored bound DAG created by* COLORED-DAG() *from CFG G,* DEPENDENCE() *returns true iff clauses* (2) *and* (3) *of Definition* 12 *are satisfied in G.*

PROOF. At the beginning of DEPENDENCE(), m and p are designated as white and black nodes, respectively. After COLORED-DAG() returns on line 7 of DEPENDENCE(), let q and r be immediate successors of n such that q is white and r black.

"Only if" direction. From Lemma 11, on all paths in the DAG from $q(r)$, $m(p)$ will be encountered before any $p(m)$ is encountered. The absence of uncolored nodes on such paths rules out the possibility of an infinite path from $q(r)$ that does not contain the $m(p)$. Hence, for all maximal paths from $q(r)$ in G, $m(p)$ will be encountered before any $m(p)$ is encountered. Thus q and r satisfy clauses (2) and (3) of Definition 12, respectively, when DEPENDENCE() returns true.

"If" direction. Suppose all maximal paths from $q(r)$ contain $m(p)$ before any occurrence of $p(m)$. This implies that there can be no node n_i on any path between $q(r)$ (inclusive) and $m(p)$ (exclusive) such that n_i has an outgoing edge that can lead to a cycle not containing $m(p)$. Hence, all nodes on these paths can be colored white(black). As a DAG rooted at n will not contain back edges leading to infinite paths and as no such edges emanate from nodes on the paths between $q(r)$ (inclusive) and $m(p)$ (exclusive), COLORED-DAG() will achieve the coloring as described earlier. Hence, DEPENDENCE() will return true when q and r satisfy clauses (2) and (3) of Definition 12. □

6.4.2 *Complexity Analysis.* COLORED-DAG() will be executed at least for every edge in the graph. As line 7 in COLORED-DAG() can be executed $|N|$ times for each execution of COLORED-DAG(), the worst-case complexity of the COLORED-DAG() will be $O(|E| \times |N| \times \lg(N))$.

The conditional at line 11 in DEPENDENCE() can execute $|N|$ times for each execution of DEPENDENCE(). By factoring in the complexity of COLORED-DAG(), the worst-case complexity of DEPENDENCE() will be $O(|N|+|E|\times|N|\times\lg(N)) = O(|E|\times |N| \times \lg(N))$.

The worst-case complexity of the graph reachability algorithm is $O(|N|^3)$. The loops at lines 3, 4, and 5 in ORDER-DEPENDENCE() will contribute $|N|^3$ iterations. Hence, the worst-case complexity of ORDER-DEPENDENCE() will be $O(|N|^3 + |N|^3 \times |E| \times |N| \times \lg(N)) = O(|N|^4 \times |E| \times \lg(N))$.

7. RELATED WORK

Fifteen years ago, control dependence was rigorously explored by Podgurski and Clarke [1990]. Since then, there has been a variety of work related to calculation and application of control dependence in the setting of CFGs that satisfy the unique end node property.

In the realm of calculating control dependence, Johnson and Pingali [1993] proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges. Later, Bilardi and Pingali [1996] proposed new concepts related to control dependence, along with algorithms based on these concepts, to efficiently calculate weak control dependence. In comparison, in this article we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz et al. [1990] presented what has now become the standard approach to interprocedural slicing via dependence graphs. However, the last decade has seen the prominence of C++, Java, and other languages that support semantically different procedure exit points (exceptional and normal). Hence, the work of Horwitz et al. [1990] cannot be applied directly as data dependence changes due to the semantic differences between exit points. This issue was recently addressed by Allen and Horwitz [2003]. In their effort, they extended the previous work [Horwitz et al. 1990] to handle exception-based interprocedural control flow. In their work, they inject normal exit nodes and exceptional exit nodes in the CFG, but then preserve the *unique exit node* property by connecting the normal and exceptional exit node to the unique exit node. They also consider the first statements of `try` and *catch* blocks, and *throw* statements, as predicate statements. In contrast, our approach is simpler, as the CFG is untouched even in case of exceptional exit nodes and/or multiple normal exit nodes.

As for control dependence across procedure boundaries, Stafford [2000] proposed compositional control dependences (and algorithms) in which intraprocedural control dependences of call sites and return points were appropriately extended to the statements in the called and calling procedure, respectively. These dependences were sensitive to the (non-)termination aspect of the called procedures. In comparison, as our definitions are path based, they are similar to Stafford's definitions in interprocedural settings when the interprocedural control flow paths of the program are constructed by appropriate splicing of intraprocedural control flow paths.

For relevant work on slicing correctness, Horwitz et al. [1989] used a semantics-based multilayered approach to reason about the correctness of slicing in the realm of data dependence. Alternatively, Ball and Horwitz [1993] used an approach based on program-point-specific history to prove the correctness of slicing for control flow graphs that are arbitrary, but still assumed to have a unique end point. Their correctness property (which holds also for irreducible CFGs) is a weaker property than bisimulation in that it does not require ordering to be maintained between observable nodes if there is no dependence between these nodes. We build off of their work to consider arbitrary control flow *possibly without* a unique end node; for irreducible CFGs, we need the extra notion of "order dependency" to achieve the stronger correctness property.

In terms of handling dependences in a concurrent setting, Krinke [1998] considered static slicing of multithreaded programs with shared variables, and focused on issues associated with interthread data dependence, but did not consider nontermination-sensitive forms of control dependence. Millett and Teitelbaum [1998] studied static slicing of Promela (the model description language for the model checker SPIN) and its application to model checking, simulation, and protocol understanding. They reused existing notions of slicing which—as we argue in this article—do not account for the subtleties of multithreaded execution. They did not discuss the appropriateness of those notions for an inherently multithreaded language like Promela, nor did they for their applications formalize a notion of a correct slice. Hatcliff et al. [1999] presented notions of dependence for concurrent CFGs to capture Java-like synchronization primitives. They proposed a notion of bisimulation as the correctness property, but did not provide a detailed definition or proof of correctness, as has been done in this work.

8. CONCLUSION

The notion of control dependence is used in myriads of applications, researchers as well as tool builders increasingly seek to apply it to modern software systems and high-assurance applications (even though the control flow structure and semantic behavior of these systems do not mesh well with the requirements of existing control dependence definitions). In this article, we have proposed conceptually simple definitions of control dependence that: (a) can be applied directly to the structure of modern software, thus avoiding unsystematic preprocessing transformations that introduce overhead, conceptual complexity, and sometimes dubious semantic interpretations; and (b) provide a solid semantic foundation for applying control dependence to reactive systems where program executions may be nonterminating.

We have rigorously justified these definitions by detailed proofs, by expressing them in temporal logic which provides an unambiguous definition and allows them to be mechanically checked/debugged against examples using automated verification tools, by showing their relationship to existing definitions, and by implementing and experimenting with them in a publicly available slicer for full Java. In addition, we have provided algorithms for computing these new control dependence relations. We applied our experience in realizing the algorithms while building the Indus Java slicer to research with several different tool chains, and found that any additional cost in computing these relations is negligible relative to the cost and ill-effects of the preprocessing steps required for previous definitions. Indeed, we were led to propose these new definitions because of frustrations experienced when using previous definitions in the construction of several large-scale tool frameworks. However, further empirical studies are needed to provide a precise characterization of the impact of the different definitions. Nevertheless, we believe that there are many benefits for widely applying these definitions in static analysis tools.

In ongoing work, we continue to explore the foundations for statically and dynamically calculating dependences for concurrent Java programs for slicing, program verification, and security applications. In particular, we are exploring the relationship between the dependences extracted from execution traces and those extracted from control flow graphs. This is being undertaken in an effort to systematically justify a comprehensive set of dependence notions for the rich features found in concurrent Java programs.

Also, we would like to further understand the relationship between order dependence and control dependence; the latter is based on whether nodes are reachable, whereas the former is based on the order in which nodes are reached. Given the numerous applications of control dependences, it may be interesting to explore applications of order dependences in the realm of compiler optimizations and program understanding. We conjecture that special cases and/or other variants of order dependences may be useful in reverse engineering the high-level structure (i.e., source code) of programs from their intermediate forms (i.e., machine instructions), based on patterns of control flow orderings.

REFERENCES

ALLEN, M. AND HORWITZ, S. 2003. Slicing Java programs that throw and catch exceptions. In *Procedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*. ACM, 44–54.

ANDERSEN, L. O. 1994. Program analysis and specialization for the C programming languages. Ph.D. thesis, DIKU, University of Copenhagen, DIKU, Universitetsparken 1, DK-2100, Copenhagen Ø, Denmark.

BALL, T. AND HORWITZ, S. 1993. Slicing programs with arbitrary control-flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging (AADEBUG)*. Lecture Notes in Computer Science, vol. 749. Springer, 206–222.

BILARDI, G. AND PINGALI, K. 1996. A framework for generalized control dependences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Pennsylvania, PA). ACM Press, New York, 291–300.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. 1999. *Model Checking*. MIT Press, Cambridge, MA.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PĂSĂREANU, C. S., ROBBY, AND ZHENG, H. 2000. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*. 439–448.

FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. O. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (Jul.), 319–349.

FRANCEL, M. A. AND RUGABER, S. 1999. The relationship of slicing and debugging to program understanding. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC)*. 106–113.

HATCLIFF, J., DWYER, M. B., AND ZHENG, H. 2000. Slicing software for model construction. *J. Higher-Order Symb. Comput. 13*, 4, 315–353. A special issue containing selected papers from the 1999 *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*.

HATCLIFF, J., CORBETT, J. C., DWYER, M. B., SOKOLOWSKI, S., AND ZHENG, H. 1999. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings on the International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 1694. Springer, 1–18.

HECHT, M. S. AND ULLMAN, J. D. 1974. Characterizations of reducible flow graphs. *J. ACM 21*, 3, 367–375.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1, 26–60.

HORWITZ, S., PFEIFFER, P., AND REPS, T. W. 1989. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 28–40.

I. MILLETT, L. AND TEITELBAUM, T. 1998. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*.

JAYARAMAN, G., RANGANATH, V. P., AND HATCLIFF, J. 2004. Kaveri: Delivering Indus Java program slicer to Eclipse. http://projects.cis.ksu.edu/docman/?group_id=12.

JOHNSON, R. AND PINGALI, K. 1993. Dependence-Based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 78–89.

KRINKE, J. 1998. Static slicing of threaded programs. In *Proceedings of the ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 35–42.

MILNER, R. 1989. *Communication and Concurrency*. Prentice-Hall. ISBN: 0-13-115007-3.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

PODGURSKI, A. AND CLARKE, L. 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng. 16*, 9, 965–979.

RANGANATH, V. P., AMTOFT, T., BANERJEE, A., B. DWYER, M., AND HATCLIFF, J. 2005. A new foundation for control-dependence and slicing for modern program structures. In *Proceedings of the European Symposium on Programming Languages and Systems (ESOP)*. Lecture Notes in Computer Science, vol. 3444. Springer, 77–93. Extended version available at http://projects.cis.ksu.edu/docman/?group_id=12.

SANTOS LABORATORY. 2007 Indus, a toolkit to customize and adapt Java programs. http://indus.projects.cis.ksu.edu.

STAFFORD, J. 2000. A formal, language-independent, and compositional approach to interprocedural control dependence analysis. Ph.D. thesis, University of Colorado.

TIP, F. 1995. A survey of program slicing techniques. *J. Programm. Lang. 3*, 121–189.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4, 352–357.

Received September 2005; accepted January 2007