# A New Foundation for Control Dependence and Slicing for Modern Program Structures [*][†] (Technical Report #2004-8)

Venkatesh Prasad Ranganath  Torben Amtoft  Anindya Banerjee
John Hatcliff
Department of Computing and Information Sciences
Kansas State University [‡]
{rvprasad,tamtoft,ab,hatcliff}@cis.ksu.edu

Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska [§]
dwyer@cse.unl.edu

October 29, 2004
revised: March 11, 2006

**Abstract**

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

This paper 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 paper proposes a notion of slicing correctness based on weak bisimulation and proves that the new definitions of control dependence generate slices that conform to this notion of correctness. This new framework of control dependence definitions and correctness results is able to support programs with either reducible or 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 Indus Java Slicer – a publicly available program slicer that has been implemented for full Java.

## 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 [20], debugging [8], partial evaluation [2], compiler optimizations such as global scheduling, loop

[*]This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607,CCR-0296182,CCR-0209205,ITR-0326577,CCR-0444167), by Lockheed Martin, and by Intel Corporation.

[†]Part of this work was published in the Proceedings of European Symposium On Programming (ESOP) 2005.

[‡]234, Nichols Hall, Manhattan KS, 66506, USA.

[§]256, Avery Hall, Lincoln NE, 68588-0115, USA.

fusion, and code motion [7]. Intuitively, a program statement $n_1$ is control-dependent on a statement $n_2$, if $n_2$ (typically, a conditional statement) controls whether or not $n_1$ 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 above) 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.

**(I.)** Classic *definitions of control dependence* 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 with (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, restriction (b) implies that existing definitions cannot be applied directly to reactive programs or system models with control loops that are designed to run indefinitely.

Restriction (a) is usually addressed by performing a pre-processing 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 [9, 20]. Restriction (b) can also be addressed in a similar fashion by, e.g., selecting a single node within the CFG to represent the end node. This case is more problematic than the pre-processing 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).

**(II.)** Existing *definitions of slicing correctness* either apply to programs with terminating execution traces, or they often fail to state whether or not 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, real-time systems with autonomous components, e.g. data sensors, etc.

Despite the difficulties, it appears that researchers and practitioners do continue to apply slicing transformations to programs that fail to satisfy the restrictions above. However, in reality the pre-processing transformations related to issue **(I)** 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 down-stream tools/transformations must interpret and build on in a consistent manner. Moreover, regarding issue **(II)** it will be infeasible to continue to ignore issues of termination as slicing is increasingly applied in high-assurance applications such as reducing models for verification [10] and for reasoning about security issues where it is crucial that liveness/non-termination properties be preserved.

Working on a larger project on 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 paper, 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 above. The specific contributions of this paper are as follows.

- After reviewing and assessing classic definitions of control dependence in Section 2 and 3, we propose new definitions of control dependence that are simple to state and easy to calculate and that apply directly to control-flow graphs that may have no end nodes or non-unique end nodes, thus avoiding troublesome pre-processing CFG transformations. We formalize these definitions and also supplement the formalization by providing equivalent definitions in Computation Tree Logic (CTL) (Section 4).

- To enable slicing of irreducible control flow graphs, we propose a new control-based dependence to capture control-flow-imposed ordering relationships between nodes of control flow graphs (Section 4).
- We clarify the relationship between our new definitions and classic definitions by showing that our new definitions represent a form of "conservative extension" of classic definitions: when our new definitions are applied to CFGs that conform to the restriction of a single end node, our definitions correspond to classic definitions – they do not introduce any additional dependences nor do they omit any dependences (Section 4.2).
- We prove that the proposed definitions 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 that require definitions of control dependence.

The notions of control dependence proposed in this paper have been implemented in our Java slicer that is publicly available as part of project Indus [23]. 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[15], an plugin that contributes Java program slicing feature to 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[6] tool set for model-checking concurrent Java systems.

**Extensions to the Conference version** This document extends its ESOP predecessor [21] with a new notion of control-based dependence to handle irreducible CFGs. A detailed correctness proof for slicing both reducible and irreducible CFGs that relies on the proposed notion of dependences is also presented. We also provide algorithms to calculate four forms of control-based dependences along with their proof of correctness.

## 2 Basic 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 control-flow graph adapted from Ball and Horwitz [3].

**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 $N^S$, $N^P$, where $N^S$ are *statement nodes* with each $n_s \in N^S$ having at most one successor, where $N^P$ are *predicate nodes* with each $n_p \in N^P$ having two distinct successors, and $N^E \subseteq N^S$ contains all nodes of $N^S$ that have no successors, *i.e.*, $N^E$ contains all end nodes of $G$,
- $E$ is a set of labeled edges that represent the control flow between graph nodes, and

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

- the start node $n_0$ has no incoming edges and all nodes in $N$ are reachable from $n_0$. $\square$

As stated earlier, existing presentations of slicing require that each CFG $G$ satisfies the *unique end node property*: there is exactly one element in $N^E = \{n_e\}$ and $n_e$ is reachable from all other nodes of $G$. The definition above *does not* require this property of CFGs, but we will consider CFGs with the unique end node property when comparing our work to previous work.

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. Specifically, for $n_s \in N^S$, $code(n_s)$ yields the code for an assignment statement, and for $n_p \in N^P$, $code(n_p)$ the code for the test of a conditional statement. 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.

A CFG *path* $\pi$ 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 $\pi$ to denote the set of nodes contained in $\pi$ and we write $n \in \pi$ when $n$ occurs in the sequence $\pi$. Path $\pi$ is *non-trivial* if it contains at least two nodes. A path is *maximal* if it is infinite or if it terminates in an end node.

The following definitions describe relationships between graph nodes and the distinguished start and end nodes [19]. 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). Node $n$ *post-dominates* node $m$ in $G$ (written $post\text{-}dom(n, m)$) if every path from node $m$ to the end node $n_e$ passes through $n$. Node $n$ *strictly post-dominates* node $m$ in $G$ if $post\text{-}dom(n, m)$ and $n \neq m$. Node $n$ is the *immediate post-dominator* of node $m$ if $n \neq m$ and $n$ is the first post-dominator on every path from $m$ to the end node $n_e$. Node $n$ *strongly post-dominates* node $m$ in $G$ if $n$ post-dominates $m$ and there is an integer $k \geq 1$ such that every path from node $m$ of length $\geq k$ passes through $n$ [20]. The difference between strong post-domination and the simple definition of post-domination above is that even though node $n$ occurs on every path from $m$ to $n_e$ (and thus $n$ post-dominates $m$), it may be the case that $n$ does not strongly post-dominate $m$ 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 post-domination is sensitive to the possibility of non-termination 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 entry 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 Java programs, 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. Our definitions and correctness results 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 $(n, \sigma)$ where $n$ is a CFG node and $\sigma$ 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 $(n_i, \sigma_i)$, the code at node $n_i$ has not yet been executed. Intuitively, the code at $n_i$ is executed on the transition from $(n_i, \sigma_i)$ to successor state $(n_{i+1}, \sigma_{i+1})$. Execution begins at the state node $(n_0, \sigma_o)$, 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 $(\mathsf{halt}, \sigma)$ where the control point is indicated by a special label $\mathsf{halt}$ – this indicates a normal termination of program execution. The presentation of slicing in Section 5 involves arbitrary finite and infinite non-empty 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 [24]. Traditionally, the "program points of interest" are called the *slicing criterion.* A slicing criterion $C$ for a program $p$ is a non-empty set of nodes $\{n_1, \ldots, n_k\}$ where each $n_i$ is a node in $p$'s CFG.

The definitions below are the classic ones of the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [24].

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 $m$ (written $m \overset{dd}{\rightarrow} n$ – the arrow pointing in the direction of data flow) if there is a variable $v$ such that

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

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 (control dependence)** Node $n$ is *control-dependent* on $m$ in program $p$ (written $m \overset{cd}{\rightarrow} n$) if

1. there exists a non-trivial path $\pi$ from $m$ to $n$ in $p$'s CFG such that every node $m' \in \pi - \{m, n\}$ is post-dominated by $n$, and
2. $m$ is not strictly post-dominated by $n$. $\square$

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 several slightly different notions of control-dependence appearing in the literature, and we will consider several of these variants and relations between them in the rest of the paper. Here we simply note that the above definition is standard and widely used (e.g., see [19]).

We write $m \overset{d}{\rightarrow} n$ when either $m \overset{dd}{\rightarrow} n$ or $m \overset{cd}{\rightarrow} n$. 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 $\overset{d}{\rightarrow}$.

**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:

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

The notion of slicing described above 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 paper 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 the source program and the 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* [25]. Informally, given an arbitrary trace $\Pi$ of $p$ and an analogous trace $\Pi_s$ of $p_s$, $p_s$ is a correct slice of $p$ if projecting out the nodes in criterion $C$ (and the variables referenced at those nodes) for both $\Pi$ and $\Pi_s$ 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, using the definition of control dependence in Definition 3, for Figure 1 (a), we can conclude that $a \overset{cd}{\rightarrow} f$ and $f \overset{cd}{\rightarrow} g$, but $a \overset{cd}{\rightarrow} g$ does not hold because $g$ does not post-dominate $f$. The fact that $a$ and $g$ are indirectly related ($a$ does play a role in determining if $g$ 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 [20] incorporate this notion of transitivity directly into the definition of control dependence.

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

We now further illustrate these dimensions by recalling definitions of strong and weak control dependence given by Podgurski and Clarke [20] and used in numerous efforts, including the study of control dependence by Bilardi and Pingali [4].

**Definition 5 (Podgurski-Clarke Strong Control Dependence)** *$n_2$ is strongly control dependent on $n_1$ ($n_1 \overset{scd}{\rightarrow} n_2$) if there is a path from $n_1$ to $n_2$ that does not contain the immediate post dominator 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 control dependence in Definition 3 is direct. For example, in Figure 1 (a), in contrast to Definition 3, we have $a \overset{scd}{\rightarrow} g$ because there is a path $afg$ which does not contain $e$, the immediate post-dominator of $a$. However, given the difference between these variants based on directness, 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.

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

The notion of weak control dependence captures dependences between nodes induced by non-termination, hence, it is non-termination sensitive. Note that for Figure 1 (a), $c \overset{wcd}{\rightarrow} d$ because $d$ is a successor of $c$ and strongly post dominates itself, and $d$ does not strongly post-dominate $b$: 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 dependence, the notion of weak control dependence is direct. Hence, $n_1 \overset{scd}{\rightarrow} n_2$ does not imply $n_1 \overset{wcd}{\rightarrow} n_2$ but $n_1 \overset{scd}{\rightarrow} n_2$ does imply $n_1 \overset{wcd^*}{\rightarrow} n_2$.

In assessing the above variants of control dependence in the context of program slicing, it is important to note that slicing based on Definition 3 or the strong control dependence above can transform a non-terminating program into a terminating one (i.e., non-termination is not preserved in the slice). In Figure 1 (a), assume that the loop controlled by $c$ is an infinite loop. Using the slice criterion $C = \{d\}$, slicing using strong control dependence would generate a slice that includes

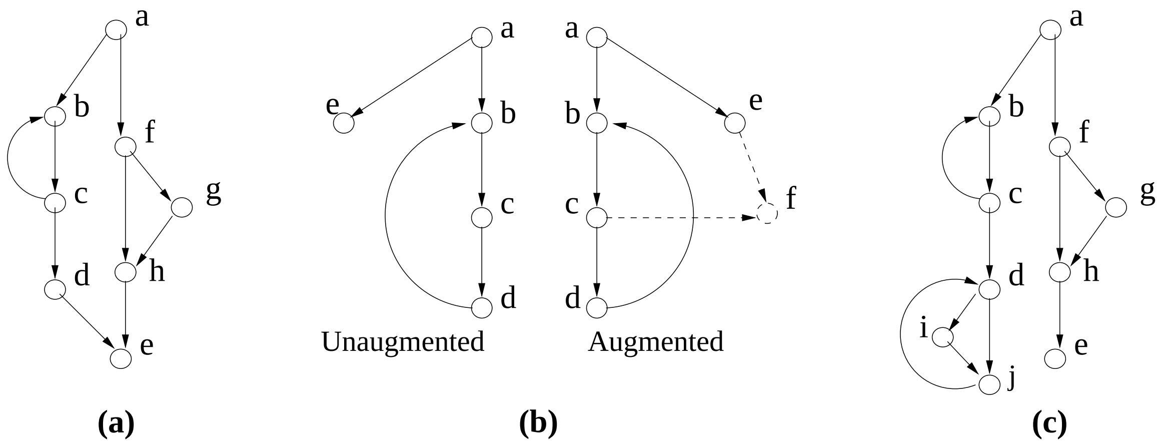

Figure 1: (a) is a simple CFG. (b) illustrates how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have unique exit node reachable from all nodes. (c) is a CFG with multiple control sinks of different sorts.

$a$ but not $b$ and $c$ (we assume no data dependence between $d$ and $b$ 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. 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 1 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 non-termination sensitive notion of weak control dependence above. We conjecture that there are at least two reasons for this. First, although it bears the qualifier "weak", weak control dependence is actually a larger relation[2] and will thus include more nodes in the slice[3]. Second, many applications of slicing focus on debugging and program visualization and understanding, and in these applications having slices that preserve non-termination 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 non-termination properties, since failure to do so could allow inferences to be made that compromise security policies, for instance invalidate checks of liveness properties [10]. This motivates our careful consideration of non-terminating program behaviors in the definitions of control dependence and slicing that we provide later in the paper.

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

Table 1: Various control dependences existing in the graph in Figure 1 (a). Control dependences denoted by $\overset{ntscd}{\rightarrow}$ and $\overset{nticd}{\rightarrow}$ will be introduced in the following pages.

[2]In Figure 1 (a), the size of $\overset{wcd}{\rightarrow}$ relation is 9 whereas the size of $\overset{scd}{\rightarrow}$ relation is 8.

[3]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 CFG

All definitions of control dependences that we are aware of 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 they suggest that CFGs can be augmented to achieve this property, e.g., 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 non-terminating loop and add an edge from $n$ to $e$. In our experience, such augmentations complicate the system being analyzed in several ways. Non-destructive 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 the reversal of the augmentation before subsequent clients use the CFG. If augmentation is not reversed, the graph algorithms and analyses algorithms should be made intelligent to operate on the actual CFG embedded in the augmented CFG.

Many systems have threads where the main control loop has no exit – the loop is "exited" by simply 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 such that they 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. Although a suggestion to delete any dependence on $c$ may work for the given CFG, it fails 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 CFG, 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 end node, we aim to instead 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 instead of 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, e.g., 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 that are added (e.g., $n_j$ always occurs before any occurrence of $n_i$) limits 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* $n_j$. The definition below considers maximal paths (which includes infinite paths) and thus is sensitive to non-termination.

**Definition 7 ($n_i \overset{ntscd}{\rightarrow} n_j$)** In a CFG, $n_j$ is **(directly) non-termination 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 from $n_k$, $n_j$ always occurs and either $n_i = n_j$ or $n_j$ strictly ($n_j \neq n_i$) precedes any occurrence of $n_i$;
2. there exists a maximal path from $n_l$ on which either $n_j$ does not occur, or $n_i$ strictly precedes any occurrence of $n_j$. □

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

We supplement a traditional presentation of dependence definitions with definitions given as formulae in computation tree logic (CTL) [5]. 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, $\mathsf{E}$ and $\mathsf{A}$, which indicate if 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 $\phi$ as a $\phi$-node): $\mathsf{X}\phi$ states that the successor node is a $\phi$-node, $\mathsf{F}\phi$ states the existence of a $\phi$-node in the path, $\mathsf{G}\phi$ states that a path consists entirely of $\phi$-nodes, $\phi\mathsf{U}\psi$ states the existence of a $\psi$-node and that the subpath leading up to that node consists of $\phi$-nodes; finally, the $\phi\mathsf{W}\psi$ operator is a variation on $\mathsf{U}$ that relaxes the requirement that a $\psi$-node exists (if not, all nodes in the path must be $\phi$-nodes). In a CTL formula, path quantifiers and modal operators occur in pairs, e.g., $\mathsf{AF}\phi$ says that on all paths from a node, a $\phi$ node occurs. A formal definition of CTL can be found in [5].

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

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

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 natural language above.

We have formulated the definition above to apply to *execution traces* instead of CFG paths. In this setting, one needs to bound relevant segments by $n_i$, as discussed above. 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

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

See Section 4.2 for the proof that Definition 7 and its simplification are equivalent on CFGs.

To see that Definition 7 is non-termination sensitive, note that $c \overset{ntscd}{\rightarrow} d$ in Figure 1 (a) since there exists a maximal path (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 Figure 1 (b unaugmented) $a \overset{ntscd}{\rightarrow} e$ because there is an infinite loop through $b, c, d$ and $a \overset{ntscd}{\rightarrow} \{b, c, d\}$ because there is maximal path ending in $e$ that does not contain $b$, $c$, or $d$. In Figure 1 (c), note that $d \overset{ntscd}{\rightarrow} i$ because there is an infinite path from $j$ (cycle on $jdj$) on which $i$ does not occur.

We now turn to constructing a non-termination insensitive version of control dependence. The above non-termination 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 node in the set is reachable from every other node in the set and there is no path leading out of the set. More precisely:

**Definition 8 (Control sink)** A *control sink* $\kappa$ 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 $\kappa$. □

Observe that each end node forms a control sink and each loop without any exit edges in the graph forms a control sink. 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 $\mathit{SinkPaths}(n_k)$) contains all *maximal* paths $\pi$ from $n_k$ with the property that there exists a control sink $\kappa$ such that

- $\pi$ contains a node $n_s$ from $\kappa$ (hence, all nodes following $n_s$ in $\pi$ will also belong to $\kappa$);
- if $\pi$ is infinite, then all nodes in $\kappa$ will occur in $\pi$ infinitely often. □

The latter requirement expresses "fairness". Note that if $\pi_1$ is a suffix of $\pi_2$, then $\pi_1$ is sink-bounded iff $\pi_2$ is sink-bounded. Also observe that in a CFG with a unique end node $n_e$, a path is sink-bounded iff it ends in $n_e$.

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 being contracted into leaf nodes. This shows:

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

Existing definitions [3, 20, 4] of non-termination 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 ($n_i \overset{nticd}{\rightarrow} n_j$)** In a CFG, $n_j$ is **(directly) non-termination 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 $\pi \in \mathit{SinkPaths}(n_k)$, $n_j \in \pi$;
2. there exists a path $\pi \in \mathit{SinkPaths}(n_l)$ such that $n_j \notin \pi$. □

This definition is expressed in CTL as

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

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

To see that this definition is non-termination insensitive, note that $c \overset{nticd}{\not\rightarrow} d$ 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) $a \overset{nticd}{\rightarrow} e$ because there is a path from $b$ to the control sink $\{b, c, d\}$ and neither the path nor the sink contain $e$, and $a \overset{nticd}{\rightarrow} \{b, c, d\}$ because there is a path ending in control sink $\{e\}$ that does not contain $b$, $c$, or $d$. It is interesting to note that in Figure 1 (c), our definition concludes that $d \overset{nticd}{\not\rightarrow} i$ because although $\{d, i, j\}$ is a control sink and there is a maximal path from $d$ that avoids $i$ (by choosing $j$ over $i$ each time), this path is not sink-bounded thanks to the "fairness" requirement. The consequence of this property is 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 below in Definition 14).

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, $b \overset{cd}{\rightarrow} c, b \overset{cd}{\rightarrow} d$, and $c \overset{cd}{\rightarrow} d$. In case of $b \overset{cd}{\rightarrow} d$, it is possible for the control to reach $d$ even if the control flows along $b \rightarrow c$. Hence, $b$ does not *decisively* decide if control can bypass $d$. However, in case of $c \overset{cd}{\rightarrow} d$, $c$ does *decisively* decide if control can bypass $d$. The decisiveness stems from the fact that the choice at the control point ($c$) that prevents the control from reaching the given program point ($d$) is *final.* Hence, the decisive control dependence relation can be defined as follows.

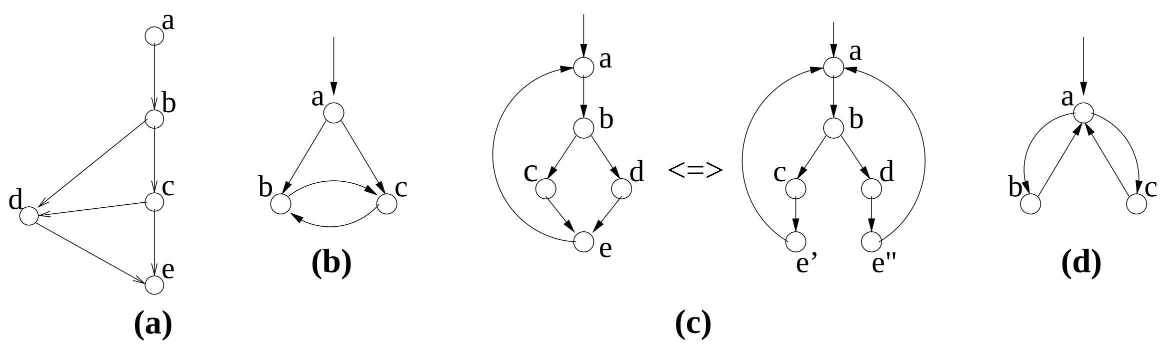

Figure 2: More control flow graphs.

**Definition 11 ($\boldsymbol{n_i \overset{dcd}{\rightarrow} n_j}$)** 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 $n_i$;
2. for all maximal paths from $n_l$, $n_j$ does not occur, or $n_j$ is strictly preceded by $n_i$. □

Although the above definition and Definition 7 are almost identical, they differ in the quantification in the second clause. Hence, the above 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 nested control structure.

Programs written in unstructured languages such as JVM bytecodes 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 above 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 if $b$ or $c$ will be executed first when the control does enter the control sink $\{b, c\}$. 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 ($\boldsymbol{n_i \overset{dod}{\rightarrow} n_j \leftrightharpoons n_k}$)** Let $n_1$, $n_2$, $n_3$ be distinct nodes. $n_2$ and $n_3$ are decisively order-dependent on $n_1$, written $n_1 \overset{dod}{\rightarrow} n_2 \leftrightharpoons n_3$, if

1. all maximal paths from $n_1$ contain both $n_2$ and $n_3$, and
2. $n_1$ has a successor from which all maximal paths[4] 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 associated correctness proofs.

Observe that the above 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. We refer to this relation as *strong order dependence.* As given in the following definition, the universal quantification on the maximal paths is required for one of $n_2$ and $n_3$, successor nodes of $n_1$.

**Definition 13 ($\boldsymbol{n_i \overset{sod}{\rightarrow} n_j \leftrightharpoons n_k}$)** Let $n_1$, $n_2$, $n_3$ be distinct nodes. $n_2$ and $n_3$ are strongly order-dependent on $n_1$, written $n_1 \overset{sod}{\rightarrow} n_2 \leftrightharpoons n_3$, if

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

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 $n_2$,
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 $n_2$. □

Strong order dependence definition can be further generalized to capture control dependence, hence, be 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 ($\boldsymbol{n_i \overset{wod}{\rightarrow} n_j \leftrightharpoons n_k}$)** In a CFG, nodes $n_j$ and $n_k$ ($n_j \neq n_k$) are weakly order dependent on $n_i$ iff

- there exists a maximal path from $n_i$ where $n_j$ strictly precedes any occurrence of $n_k$,
- there exists a maximal path from $n_i$ where $n_k$ strictly precedes any occurrence of $n_j$, and
- $n_i$ has a successor $n_l$ such that either
  - on all maximal paths from $n_l$, $n_j$ strictly precedes any occurrence of $n_k$, or
  - on all maximal paths from $n_l$, $n_k$ strictly precedes any occurrence of $n_j$. □

Given the definition of various forms of order dependences and the property[5] of reducible CFG – every cycle in a reducible CFG has one node that dominates other nodes of the cycle – it is possible to naively conclude that there can be no order dependences of any form between $n_i, n_j$, and $n_k$ provided they are distinct and occur in a reducible CFG. This is true in case of decisive (as proved in Lemma 3) and strong variants of order dependence. However, this is not true in case of weak order dependence. As an example, observe that $b$ and $c$ are weakly order dependent on $a$ ($a \overset{wod}{\rightarrow} b \leftrightharpoons c$) in the reducible graph in Figure 2 (d) while $b$ and $c$ are neither strongly nor decisively order dependent on $a$. Further, one can observe and prove (although not done in this effort) that, in a reducible CFG, $a \overset{wod}{\rightarrow} b \leftrightharpoons c \implies a \overset{ntscd}{\rightarrow} b \vee a \overset{ntscd}{\rightarrow} c$.

Although order dependence captures the ordering on nodes imposed by control flow, it is overly conservative in cases where such an 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 the order is observed only by the changes to variables used in $a$ and $b$ 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 ($\boldsymbol{n_i \overset{dsod}{\rightarrow} n_j \leftrightharpoons n_k}$)** In a CFG, nodes $n_j$ and $n_k$ ($n_j \neq n_k$) are **data-sensitive order dependent** on $n_i$ iff

1. $n_i \overset{sod}{\rightarrow} n_j \leftrightharpoons n_k$;
2. either $n_j \overset{dd}{\rightarrow} n_k$ or $n_k \overset{dd}{\rightarrow} n_j$. □

[5]Definition (f) in the abstract of [11]

## 4.1 Examples

Consider Figure 1 (c). According to Definition 7, $a \overset{ntscd}{\rightarrow} b$ as the first execution of $b$ depends on the choice made at $a$. Likewise, $a \overset{ntscd}{\rightarrow} c$ and $a \overset{ntscd}{\rightarrow} f$. Similarly, $f \overset{ntscd}{\rightarrow} g$. Independent of the choice made at $f$, the control will always reach $h$. Hence, $f \overset{ntscd}{\not\rightarrow} h$ but $a \overset{ntscd}{\rightarrow} h$. Similarly, $a \overset{ntscd}{\rightarrow} e$, and $c \overset{ntscd}{\rightarrow} b$. If $b \rightarrow c \rightarrow b$ is an infinite loop, control will never reach $d$; the length of the loop is dependent on the choice made at $c$. Hence, $c \overset{ntscd}{\rightarrow} d$. In the loop starting at $d$, it is possible that the control will bypass $i$ in an iteration while it reaches $i$ in a subsequent iteration depending on the choice made at $d$.[6] Hence, $d \overset{ntscd}{\rightarrow} i$.

In a non-termination insensitive setting, loops are assumed to be terminating (provided the loop has an exit node). Hence, in Figure 1 (c), the loop $b \rightarrow c \rightarrow b$ is assumed terminating as it has an exit edge $c \rightarrow d$. This implies that the loop cannot indefinitely delay the control from reaching $d$. Hence, $c \overset{nticd}{\not\rightarrow} d$. As for other non-termination sensitive control dependences for the same graph, most of them hold also in the non-termination *in*sensitive case, except we have $d \overset{nticd}{\not\rightarrow} i$, and also $c \overset{nticd}{\not\rightarrow} j$ as $j$ belongs to the control sink that terminates all sink-bounded paths from $c$. As for decisive control dependence, in Figure 2 (a), $c \overset{dcd}{\rightarrow} d$ and $b \overset{dcd}{\not\rightarrow} d$.

| *Nodes* | $\overset{ntscd}{\rightarrow}$ | $\overset{nticd}{\rightarrow}$ |
|---|---|---|
| $a$ | $b, c, f, h, e$ | $b, c, d, i, j, f, h, e$ |
| $c$ | $b, d, j$ | $b$ |
| $d$ | $i$ | $-$ |
| $f$ | $g$ | $g$ |

Table 2: Various control dependences (based on new definitions) existing in the graph in Figure 1 (c).

In Figure 2 (b), $b$ and $c$ are decisively order dependent on $a$ ($a \overset{dod}{\rightarrow} b \leftrightharpoons c$), hence, $b \overset{dd}{\rightarrow} c$ or $c \overset{dd}{\rightarrow} b$ implies $a \overset{dsod}{\rightarrow} b \leftrightharpoons c$. This result also holds for *strong* and *weak* order dependence. On the other hand, $b$ and $c$ are weakly order dependent on $a$ in Figure 2 (d) but they are neither related by decisive nor strong order dependence. Also, $i$ and $j$ are weakly order dependent on $d$ ($d \overset{sod}{\rightarrow} i \leftrightharpoons j$) in Figure 1 (c).

Focusing only on order dependences, in Figure 3 (a), $c$ and $d$ are strongly and weakly order dependent on both $b$ and $a$. However, $c$ and $d$ are decisively order dependence only on $b$.

Unlike in Figure 3 (a), $c$ and $d$ are neither strongly nor decisively order dependent on *b, b',* and $a$ in Figure 3 (b). The reason for this is the absence of edges $c \rightarrow d$ and $d \rightarrow c$. However, $c$ and $d$ are weakly order dependent on $b$ and *b'* in Figure 3 (b).

## 4.2 Properties of the Dependence Relations

**Conservative extension.** We begin by showing that the new definitions of control dependence conservatively extend classic definitions: when we consider our definitions in the original setting with CFGs with unique end nodes, the definitions coincide with the classic definitions (as already suggested by the examples in Table 1).

**Theorem 1 (Coincidence Properties, I)** *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$, $n_i \overset{cd}{\rightarrow} n_j$ if and only if $n_i \overset{nticd}{\rightarrow} n_j$.* □

Proof First notice that for any $n$ and $m$, $m$ post-dominates $n$ if and only if every sink-bounded path from $n$ contains $m$.

[6]Observe that the loop starting at $d$ can be split into two loops as done in Figure 2 (c).

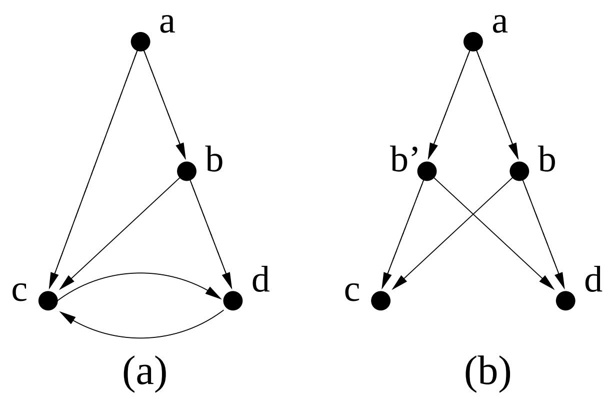

Figure 3: Control flow graphs specific to order dependence.

We shall first prove "only if": so let $n_i \overset{cd}{\rightarrow} n_j$. There thus exists a non-trivial path $\pi$ from $n_i$ to $n_j$ such that every node in $\pi$ except $n_i$ is post-dominated by $n_j$. Let $\pi$ 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 post-dominated by $n_j$.

Also, we know from $n_i \overset{cd}{\rightarrow} n_j$ that $n_i$ is not strictly post-dominated by $n_j$. Therefore either $n_i = n_j$, or $n_i$ is not post-dominated 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 post-dominated by $n_j$.

We have thus found $n_k$ and $n_l$, 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 $n_i \overset{nticd}{\rightarrow} n_j$.

Next we prove "if": so let $n_i \overset{nticd}{\rightarrow} n_j$. 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 post-dominated by $n_j$; in either case, $n_i$ is not strictly post-dominated 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 $\pi$ be a shortest such path. In order to show that $n_i \overset{cd}{\rightarrow} n_j$, it suffices to show that all nodes in $\pi$ are post-dominated by $n_j$. But this clearly follows from (i). ■

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

**Definition 16** In a CFG, $n_j$ is non-termination sensitive control dependent on $n_i$ iff

**(a)** $n_i$ has two successors $n_k$ and $n_l$;

**(b)** on all maximal paths from $n_k$, $n_j$ occurs;

**(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 non-termination sensitive control dependence (Definition 7).* □

Proof First, we restate the definition of directly non-termination sensitive control dependence (Definition 7); we have $n_i \overset{ntscd}{\rightarrow} n_j$ 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 it equals $n_i$ or it occurs before any occurrence of $n_i$.

**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 with showing two implications:

- First, we show that **(b)** implies **ntscd(ii)**: Let $\pi$ be a maximal path from $n_k$. By **(b)**, $n_j$ occurs there. Now assume, towards a contradiction, that in $\pi$, $n_i$ 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 **ntscd(iii)** implies **(c)**: Let $\pi$ be a maximal path from $n_l$ on which $n_i$ occurs strictly before the first (if any) occurrence of $n_j$. If $\pi$ does not contain $n_j$, we are done. So assume that $\pi$ 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 could be expected, we have a similar result relating the corresponding CTL formulae:

**Theorem 2 (Simplified NTSCD CTL Equivalence)** *The expression of NTSCD as a CTL formula over CFG paths ($\phi_{CFG}$):*

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

*is equivalent to the CTL formula over execution traces ($\phi_{trace}$):*

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

Proof It suffices to prove that the pairs of sub-formulae under the $\mathsf{EX}$ operators in the two formulae are equivalent.

We prove that $\phi_{\text{CFG}}$ implies $\phi_{\text{trace}}$ 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 $\mathsf{EW}$ 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 $\mathsf{AU}$ 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.

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

   Every path from a CFG node $n_i$ either has a prefix that is cyclic in $n_i$, $n_i(\neg n_i)^* n_i$, or is a path that is acyclic in $n_i$, $n_i(\neg n_i)^*$. 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 $\phi_{\text{CFG}}$ implies $\phi_{\text{trace}}$.

We prove that $\phi_{\text{trace}}$ implies $\phi_{\text{CFG}}$ in two steps:

1. $\mathsf{A}[\neg n_i \mathsf{U} n_j])$ implies $\mathsf{AF}(n_j)$:
   The $\mathsf{AU}$ 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 $\mathsf{EW}$ 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 $\mathsf{EW}$ 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 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 above $\mathsf{EW}$. 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 $\phi_{\text{trace}}$ implies $\phi_{\text{CFG}}$. ■

**Theorem 3 (Coincidence Properties, II)** *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$, $n_i \overset{wcd}{\rightarrow} n_j$ if and only if $n_i \overset{ntscd}{\rightarrow} n_j$.* □

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 $n_i \overset{wcd}{\rightarrow} n_j$ iff:

**pcwcd(i)** $n_i$ has at least two successors, $n_k$ and $n_l$.

**pcwcd(ii)** $n_j$ strongly postdominates $n_k$.

**pcwcd(iii)** $n_j$ does not strongly postdominate $n_l$.

There are four steps.

1. **pcwcd(ii)** implies **(b)**: Let $\pi$ be a maximal path from $n_k$. We must show that $n_j$ occurs in $\pi$. There are two possibilities:
   *$\pi$ is finite*: The last node of $\pi$ must be an end node. Since $n_j$ postdominates $n_k$, this shows that $n_j$ occurs in $\pi$.
   *$\pi$ is infinite*: We know that there exists $q$ such that all paths from $n_k$ longer than $q$ contain $n_j$; in particular, $\pi$ will contain $n_j$ since $\pi$ is infinite, hence longer than $q$.

2. **(b)** implies **pcwcd(ii)**: First let us show that $n_j$ postdominates $n_k$; so let $\pi$ be a path from $n_k$ to an end node. We must show that $\pi$ contains $n_j$, but this follows from **(b)** since $\pi$ 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. For let $\pi$ be a path from $n_k$ longer than $q$: it contains a repetition, so if $n_j$ does not occur in $\pi$ 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 $\pi$ from $n_l$ to an end node such that $n_j$ does not occur in $\pi$. The claim now follows since $\pi$ is maximal.
   *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 $\pi$ from $n_l$ with $n_j$ not occurring in $\pi$. Now there are two cases:
*$\pi$ is finite, with the last node being an end node*: But then $n_j$ does not postdominate $n_l$, in particular $n_j$ does not strongly postdominate $n_l$.
*$\pi$ is infinite*: But then for any $q$, $\pi$ 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 conclude the proof of Theorem 3. ■

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

**Theorem 4** *For all nodes $n_i, n_j \in N$, $n_i \overset{nticd}{\rightarrow} n_j$ implies $n_i \overset{ntscd^*}{\rightarrow} n_j$.* □

Note that this result is supported by the examples in Tables 1 and 2. For example, in Figure 1 (a), $a \overset{nticd}{\rightarrow} d$ holds but $a \overset{ntscd}{\rightarrow} d$ does not. But $a \overset{ntscd^*}{\rightarrow} d$ holds as both $a \overset{ntscd}{\rightarrow} c$ and $c \overset{ntscd}{\rightarrow} d$ 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 $\pi$ 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 $i = 1 \ldots m$, $n_j$ occurs on all sink-bounded paths from $u_i$ (otherwise **(i)** would be contradicted). So, if all sink-bounded paths from $n_l$ would contain $u_i$, all sink-bounded paths from $n_l$ would contain $n_j$, contradicting **(ii)**. Thus for all $i = 1 \ldots m$, there exists a sink-bounded path from $n_l$ not containing $u_i$.

Now define predicates $Q_p$ such that $Q_p(i)$ holds iff $0 \leq i \leq p \leq m$ and all maximal paths from $u_i$ contain $u_p$. Observe that if $Q_p(i)$ does not hold but $Q_p(i+1)$ holds, then $u_i \overset{ntscd}{\rightarrow} u_p$ (cf. 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 $n_l$ contains $u_p$, contradicting the above).

Now we are ready for the construction: if $u_m = u_0$, we are done. Otherwise, we can find $j_1$ such that $Q_m(j_1)$ does not hold but $Q_m(j_1+1)$ holds, showing that $u_{j_1} \overset{ntscd}{\rightarrow} u_m$. If $u_{j_1} = u_0$, we are done. Otherwise, since $Q_{j_1}(j_1)$ holds but $Q_{j_1}(0)$ does not hold, we can find $j_2$ such that $Q_{j_1}(j_2)$ does not hold but $Q_{j_1}(j_2+1)$ holds, showing that $u_{j_2} \overset{ntscd}{\rightarrow} u_{j_1}$. Now we can repeat as desired. ■

**Order dependency is relevant for irreducible graphs only.**

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

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

**od(i)** all maximal paths from $n_1$ contain both $n_2$ and $n_3$, and

**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

$n_1$ is reachable from neither $n_2$ nor $n_3$. (*1)

For otherwise, we could wlog. assume 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 wlog. 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

in $E_f$, a path $[n_3..n_4]$ not containing $n_2$ (*2)

and a back edge

$n_4 \rightarrow n_5$ where $n_5$ dominates $n_4$. (*3)

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

a path $[n_0..n_1]$ not containing $n_2$. (*4)

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

a path $[n_1..n_3]$ not containing $n_2$. (*5)

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

a path $[n_0..n_4]$ containing $n_1$ but not containing $n_2$.

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, members of which are termed "observable". Typically, these results require slice sets $\Xi$ to be closed under non-termination sensitive control dependency, i.e., if $n_1 \overset{ntscd}{\rightarrow} n_2$ and $n_2 \in \Xi$ then also $n_1 \in \Xi$. For certain weaker results, it is sufficient to demand that $\Xi$ is closed under non-termination insensitive control dependency, i.e., if $n_1 \overset{nticd}{\rightarrow} n_2$ 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 (Theorem 5), we shall also demand $\Xi$ to be closed under (decisive) order dependency, i.e., if $n_i \overset{dod}{\rightarrow} n_j \leftrightharpoons n_k$ 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 $\Xi$. □

Clearly, if $n \in \Xi$ then $obs^1_{may}(n) = \{n\}$. Next, 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", i.e., the first observable on any path will be encountered sooner or later on all other paths:

**Lemma 4** *Assume the node set $\Xi$ is closed under non-termination sensitive control dependency. 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 $\pi$ from $n_0$ to $n_1$ where all nodes except $n_1$ do not belong to $\Xi$. 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

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

By our assumption, $Q(n_0)$ does not hold; clearly, $Q(n_1)$ holds. Therefore, $\pi$ can be written as $[n_0..n_2n_3..n_1]$ where $Q(n_2)$ does not hold but $Q(n_3)$ holds (that is, 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 $n_2 \overset{ntscd}{\rightarrow} n_1$; then from $n_1 \in \Xi$ and from $\Xi$ being closed under $\overset{ntscd}{\rightarrow}$ we get $n_2 \in \Xi$ which contradicts $n_1$ being the only node in $\pi$ which is also in $\Xi$.

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 (cf. 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 $\Xi$ is closed under $\overset{nticd}{\rightarrow}$. 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 un-observable paths:

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

- *If $\Xi$ is closed under non-termination insensitive control dependency, then all sink-bounded paths starting at $n_0$ will reach $\Xi$.*
- *If $\Xi$ is also closed under non-termination sensitive control dependency, then all maximal paths starting at $n_0$ will reach $\Xi$.* □

Our main result, Theorem 5 given below, 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 non-termination sensitive control dependency (since $a \overset{ntscd}{\not\rightarrow} b$ and $a \overset{ntscd}{\not\rightarrow} c$) and provides $a$ with *two* possible first observables. Our remedy is to demand that the slice set $\Xi$ is 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 $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$ and $\overset{dod}{\rightarrow}$, 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 $\pi$ from $n_0$ such that $|obs^1_{may}(n)| > 1$ holds for all $n$ occurring in $\pi$, for then $\pi$ would contain no nodes in $\Xi$, contradicting $obs^*_{must}(n_0)$ being non-empty. Thus there exists a node $n_1$ such that $|obs^1_{may}(n_1)| > 1$, and thus $n_1 \notin \Xi$, 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 $n_1 \overset{dod}{\rightarrow} n_2 \leftrightarrows n_3$, which since $\Xi$ is closed under $\overset{dod}{\rightarrow}$ and since $n_2, n_3 \in \Xi$ will imply $n_1 \in \Xi$, yielding 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 $n_2 \in obs^1_{may}(n_1) \subseteq obs^*_{must}(n_1)$.

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

Note: Theorem 5 will *not* hold if we assume only that $\Xi$ is closed under non-termination *in*sensitive control dependency. To see this, consider the following reducible graph: in $E_f$, there is an edge from $n_2$ to $n_1$ and an edge from $n_1$ to $n_3$ and also a direct edge from $n_2$ to $n_3$; in $E_b$, there is an edge from $n_1$ to $n_2$ and an edge from $n_3$ to $n_2$. The only control sink is thus the whole graph, so the relation $\overset{nticd}{\rightarrow}$ is empty; also the relation $\overset{dod}{\rightarrow}$ is empty (by Lemma 3). All node sets are thus vacuously closed under $\overset{nticd}{\rightarrow}$ and $\overset{dod}{\rightarrow}$. Assume $\Xi$ is such that $n_2, n_3 \in \Xi$ but $n_1 \notin \Xi$. Then $obs^1_{may}(n_1)$ contains *two* elements: $n_2$ and $n_3$. (On the other hand, $n_3$ is non-termination sensitive dependent on $n_1$, so $\Xi$ is not closed under non-termination sensitive control dependency, and the scenario is thus *not* a counterexample to the actual Theorem 5.)

## 5 Slicing

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

**Definition 19 (Slicing Transformation)** The result of slicing is a program with the same CFG as the original one, 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`;
- if $n$ is a predicate node then $code_2(n)$ is `cskip`, the semantics of which is that it non-deterministically chooses one of its successors. □

The above definition is conceptually simple, so as to facilitate the correctness proofs. Of course, one would want to do some post-processing, like eliminating `skip` commands and eliminating `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 $C$ correspond to *silent moves* or non-observable 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 wrt. the original code map $code_1$ as wrt. the sliced code map $code_2$, and this will imply that they have the same $C$ observations.

The discussion above 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 $\tau$-moves [18]. Recall from Sect. 2.2 that a state $s$ is a pair $(n, \sigma)$ where $\sigma$ is a store mapping variables into values.

**Definition 20** For $i = 1, 2$ we write

- $i \vdash s \rightarrow s'$ to denote that wrt. 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** A binary relation $\phi$ 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 $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$ and $\overset{dod}{\rightarrow}$, 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** For each node $n$ in $G$, we define $relv(n)$, the set of relevant variables at $n$, by stipulating that $x$ in $relv(n)$ iff there exists a node $n_k \in \Xi$ and a path $\pi$ from $n$ to $n_k$ such that $x \in ref(n_k)$, but for all nodes $n_j$ occurring before $n_k \in \pi$, $x \notin def(n_j)$. □

Strictly speaking, we should have defined (for i = 1,2) functions $ref_i(n)$ to return the variables referenced at node $n$ wrt. code map $code_i$, functions $def_i(n)$ to return the variables defined at node $n$ wrt. code map $code_i$, and functions $relv_i(n)$ and relation $\overset{dd}{\rightarrow}_i$ parametrized wrt. $ref_i$ 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 has the same set of relevant variables for each node.

**Lemma 7** *Assume, with $\overset{dd}{\rightarrow}_i$ etc. as defined just above, that $\Xi$ is closed under $\overset{dd}{\rightarrow}_1$. Then*

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

PROOF To show (1), assume the contrary; then there exists a path $\pi$ 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 $\pi$: $x \notin def_2(n')$. 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 $\Xi$ is closed under $\overset{dd}{\rightarrow}_1$, we can infer that there exists a node $n'$ interior in $\pi$ with $x \in def_1(n')$; let $n_1$ be the last such $n'$. Then $n_1 \in \Xi$ and $x \in def_1(n_1)$ and (cf. above) $x \notin def_2(n_1)$. But since $n_1 \in \Xi$ we have $code_1(n_1) = code_2(n_1)$ which yields the desired contradiction.

To show (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 $\pi$ 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 $\pi$, $x \notin def_i(n')$. 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 $\pi$. 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 $\pi$ with $x \in def_j(n')$. Then $n_1 \overset{dd}{\rightarrow}_j n_k$; since $n_k \in \Xi$ which by (1) is closed under $\overset{dd}{\rightarrow}_j$, 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 $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$. 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 $\Xi$ 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 $\pi$ 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 $\pi$. Since $n_3 = obs^1_{may}(n_1)$, we can split $\pi$ 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 with showing that $x \notin def(n_j)$ for all nodes $n_j$ occurring before $n_k$ in $\pi'$. Assume the contrary, and let $n'$ be the last node in $\pi'$ serving as a counterexample. Since $\Xi$ is closed under $\overset{dd}{\rightarrow}$, we infer that $n' \in \Xi$; also, due to the properties of $\pi$, we infer that $n'$ does not occur in $\pi_0$, and therefore, $n'$ occurs before $n_3$ in $\pi_2$. But this contradicts $n_3 = obs^1_{may}(n_2)$. ■

We need one more auxiliary result.

**Lemma 9** *Assume that $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$. If $i \vdash s_1 \overset{\tau}{\longmapsto} s_2$ 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 $V$ 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 (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, 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 $obs^1_{may}(n_2) = \emptyset$.

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 program.

**Definition 23** For $\Xi$ closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$, 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 $s_1 = (n_1, \sigma_1)$ and $s_2 = (n_2, \sigma_2)$ 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 $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$, then the relation $R$ from Definition 23 is a weak bisimulation (cf. Def. 21).* □

Proof For $i \in \{1, 2\}$ and $j = 3 - i$, we must show

1. If $s_1\,R\,s_2$ 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 $s_1'\,R\,s_2'$.

2. If $s_1\,R\,s_2$ 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 $s_1'\,R\,s_2'$.

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

For (2), assume that $i \vdash s_i \overset{n}{\longmapsto} s_i'$. Thus $s_i$ is of the form $(n, \sigma_i)$; 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 $\Xi$. All this shows that there exists $s_j'' = (n, \sigma_j'')$ such that $j \vdash s_j \overset{\tau}{\Longrightarrow} s_j''$. By repeated application of Lemma 9 we infer $\sigma_j'' =_V \sigma_j$ and since $\sigma_i =_V \sigma_j$ thus also $\sigma_i =_V \sigma_j''$. In particular,

$\sigma_i$ and $\sigma_j''$ agree on $ref(n)$. (*)

Therefore, $s_j''$ will choose the same branch as $s_i$ (if $n$ is a predicate node, otherwise vacuously). That is, there exists $s_j'$ of the form $(n', \sigma_j')$ such that $j \vdash s_j'' \overset{n}{\longmapsto} s_j'$ and thus $j \vdash s_j \overset{n}{\Longrightarrow} s_j'$. We are left with showing 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 bisimular to the initial state 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 [25] to infinite traces, this demonstrates that

**Theorem 7** *If $\Xi$ is closed under $\overset{ntscd}{\rightarrow}$, $\overset{dod}{\rightarrow}$, and $\overset{dd}{\rightarrow}$, then the sliced program has the same "observable behavior" as the original 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 Non-Termination 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.

```
NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE($G$)
$G(N, E, n_0)$ :  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
$workbag \leftarrow \emptyset$
for  each $n$ in $condNodes(G)$
do for  each $m$ in $succs(n, G)$
    do $S[m, n] \leftarrow \{t_{nm}\}$
        $workbag \leftarrow workbag \cup \{m\}$

 # (2) Calculate all-path reachability
while $workbag \neq \emptyset$
do $n \leftarrow remove(workbag)$
     # (2.1) One successor case
    if $T_n = 1$ and $n \notin succs(n, G)$
      then $m \leftarrow select(succs(n, G))$
            for $p$ in $condNodes(G)$
            do if $S[n, p] \backslash S[m, p] \neq \emptyset$
                  then $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
                        $workbag \leftarrow workbag \cup \{m\}$
     # (2.2) Multiple successors case
    if $|succs(n, G)| > 1$
      then for $m$ in $N$
            do if $|S[m, n]| = T_n$
                  then for $p \in condNodes(G) \backslash \{n\}$
                        do if $S[n, p] \backslash S[m, p] \neq \emptyset$
                              then $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
                                    $workbag \leftarrow workbag \cup \{m\}$
 # (3) Calculate non-termination sensitive control dependence
for  each $n$ in $N$
do for  each $m$ in $condNodes(G)$
    do if $0 < |S[n, m]| < T_m$
          then $CD[m] \leftarrow CD[m] \cup \{n\}$

return $CD$
```

Figure 4: Algorithm to calculate non-termination sensitive control dependence

### 6.1.1 Proof of correctness

The correctness of the algorithm (Figure 4) is presented as the following theorem.

**Theorem 8** *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 "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 the completion of phase (1), the invariant is trivially established at the beginning of phase (2). If $n$ has only one successor $m$ then all maximal paths containing $n$ will contain $m$. Hence, the assignement at line 21 establishes the invariant at the end of the loop at line 19 (and conditional at line 17). If $n$ has multiple succesors and all maximal paths through the successors contain $m$ then all maximal paths containing $n$ will also contain $m$. This is captured by the assigment at line 29 and the invariant is established at the end of the loops at line 25 and 27.

As the graph has finite number of nodes, the number of successors of a node is finite. Hence, the total number of symbols ($t_{nm}$) in the $G$ 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 (line 22 and line 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 node that is the first common node to occur on all maximal paths originating from the successors of $q$. Let $r$ be this common node. Also assume there are no conditional nodes in the paths from $q$ to $r$. From the previous results and non-branching property of the paths between $q$ and $r$, $|S_{rq}| = T_q$. This implies $S_{qn} \subseteq S_{rn}$, hence, the supposition is falsified. Similar reasoning can be applied inside-out when conditional nodes occur on the paths from $q$ to $r$.

The above 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 8, it is trivial to see that phase (3) correctly calculates non-termination sensitive control dependence.

### 6.1.2 Complexity analysis

Phases (1) and (3) of the algorithm (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 Non-Termination Insensitive Control Dependence (NTICD)

The proposed algorithm (Figure 5) to calculate non-termination insensitive control dependence is very similar to the NTSCD algorithm. The only differences being 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 non-termination sensitivity. Phase (2.3) in NTICD algorithm induces non-termination insensitivity by undoing this preservation. Also, $t_{nm}$ represents all extensible finite paths starting with $n \rightarrow m$ in NTICD algorithm.

### 6.2.1 Proof of correctness

Given the similarity of NTSCD and NTICD algorithms, we prove the correctness of NTICD algorithm by proving that phase (3) along with the interpretation attached to $t_{nm}$ calculates non-termination insensitive control dependence.

The key observation being that phase (2.3) induces non-termination insensitivity. Succinctly, if $t_{nm} \in S_{nn}$ then $t_{nm}$ is added to $S_{pn}$ where $p$ is a successor of $n$. Thus establishing that all finite paths that start with $n \rightarrow m$ and that reach $n$ can be finitely extended to reach $p$, hence, inducing non-termination insensitivity.

**Lemma 10** *If $t_{nm} \in S_{pn}$ and $p$ belongs to a control sink then for all nodes $q \in$ c-sink$(p).t_{nm} \in S_{qn}$* □

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, all finite paths from $p$ can be extended to $q$. Hence, $S_{pn} \subseteq S_{qn}$. Similarly, we can prove $S_{qn} \subseteq S_{pn}$. Hence, $S_{pn} = S_{qn}$. ■

**Theorem 9** *Phase (3) of NTICD calculates non-termination 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 $|S_{pn}| = T_n$, it 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 non-termination insensitive control dependence. ■

### 6.2.2 Complexity analysis

Phase (2.3) of NTICD algorithm contributes $O(\sum T_n \times |N| \times \lg(|N|))$ to the overall complexity of phase (2) of 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 NTICD algorithm is identical as that of NTSCD algorithm.

## 6.3 Decisive Control Dependence (DCD)

As Definition 11 implies Definition 7, we calculate decisive control dependence by pruning non-termination sensitive control dependence. It is evident that clause (2) in Definition 11 is a stronger than that in Definition 7. Hence, we use the negative form of clause (2) in Definition 11 – 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 non-termination sensitive control dependence to calculate decisive control dependence.

In the algorithm (Figure 6), $t_{nm}$ represents a path $\pi$ that starts with $n \rightarrow m$ and is maximal or terminates with $n$ while $t_{nm} \in S_{pn}$ represents that 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 nodes and other nodes of the $G$. This information is later used in phase (3) to calculate decisive control dependence.

```
Non-Termination-Insensitive-Control-Dependence($G$)
$G(N, E, n_0)$ :  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
$workbag \leftarrow \emptyset$
for  each $n$ in $condNodes(G)$
do for  each $m$ in $succs(n, G)$
    do $S[m, n] \leftarrow \{t_{nm}\}$
        $workbag \leftarrow workbag \cup \{m\}$

 # (2) Calculate all-path reachability
while $workbag \neq \emptyset$
do $n \leftarrow remove(workbag)$
     # (2.1) One successor case
    if $T_n = 1$ and $n \notin succs(n, G)$
      then $m \leftarrow select(succs(n, G))$
            for $p$ in $condNodes(G)$
            do if $S[n, p] \backslash S[m, p] \neq \emptyset$
                 then $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
                      $workbag \leftarrow workbag \cup \{m\}$
     # (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 $condNodes(G) \backslash \{n\}$
                      do if $S[n, p] \backslash S[m, p] \neq \emptyset$
                           then $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
                                $workbag \leftarrow workbag \cup \{m\}$
     # (2.3) Erase non-termination sensitivity
    if $|S[n, n]| > 0$
      then for $m$ in $succs(n, G) \backslash n$
            do if $S[n, n] \backslash S[m, n] \neq \emptyset$
                 then $S[m, n] \leftarrow S[m, n] \cup S[n, n]$
                      $workbag \leftarrow workbag \cup \{m\}$

 # (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 $CD[m] \leftarrow CD[m] \cup \{n\}$

return $CD$
```

Figure 5: Algorithm to calculate non-termination insensitive control dependence

```
DECISIVE-CONTROL-DEPENDENCE($G$)
 1  $G(N, E, n_0)$ :  a control flow graph
 2  $S[|N|, |N|]$ :  a matrix of sets where $S[n_1, n_2]$ represents $S_{n_1 n_2}$
 3  $T[|N|]$ :  a sequence of integers where $T[n_1]$ denotes $T_{n_1}$
 4  $CD[|N|]$ :  a sequence of sets
 5  $workbag$ :  a set of nodes
 6
 7   # (1) Initialize
 8  $workbag \leftarrow \emptyset$
 9  for each $n$ in $condNodes(G)$
10  do $succs \leftarrow succs(n, G)$
11      for each $m$ in $succs$
12      do $workbag \leftarrow workbag \cup \{m\}$
13          $S[m, n] \leftarrow \{t_{nm}\}$
14
15   # (2) Calculate exists-a-path reachability
16  while $workbag \neq \emptyset$
17  do $n \leftarrow remove(workbag)$
18      for each $m$ in $succs(n, G)$
19      do for each $p$ in $condNodes(G)$
20          do if $S[n, p] \backslash S[m, p] \neq \emptyset$
21                then $S[m, p] \leftarrow S[m, p] \cup S[n, p]$
22                     $workbag \leftarrow workbag \cup m$
23
24   # (3) Calculate decisive control dependence
25  $CD \leftarrow$ 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 $CD[n] \leftarrow CD[n] \backslash \{m\}$
30
31  return $CD$
```

Figure 6: Algorithm 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 the conditional nodes and the other nodes of $G$.

**Theorem 10** *At the end of phase(2) in the DCD algorihtm, $t_{nm} \in S_{pn}$ iff there exists a path starting with $n \rightarrow m$ that can be extended to $p$.* $\square$

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, phases (1) will terminate. The invariant is trivially established at the beginning of phase (2). The loops at line 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 the 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 the $G$ 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 the $G$. 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 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 if the three clauses in the definition of decisive order dependence[7] are satisfied. We can use information from graph reachability algorithm to check if $m$ and $p$ satisfy 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 colored bound directed acyclic graph (DAG). The bounding condition is that out-going 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 only if 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 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 depenendent* on any node $n$ that has at least one *black* child and at least one *white* child.

In case of a cyclic graph, the source $q$ of a back egde 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 above description/intuition, we need to prove that the coloring and bounding of the DAG does indeed capture the information required to decide if $n \overset{dod}{\rightarrow} m \leftrightharpoons p$. We shall prove the correctness of the algorithm by proving the following theorems.

**Theorem 11** *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,*
3. *a node is uncolored if (1) all its immediate successors are uncolored, it has at least two children of different colors, or it is the source of a back edge in G.* □

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

```
ORDER-DEPENDENCE()
1  $OD[|N|][|N|]$ :  a matrix that captures order dependence
2  $G(N, E, n_0)$ :  a control flow graph
3  for  each $n$ in $condNodes(G)$
4  do for  each $m$ in $N$
5      do for  each $p$ in $N \backslash \{m\}$
6          do if REACHABLE$(m, p, G) \wedge$ REACHABLE$(p, m, G) \wedge$ DEPENDENCE$(n, p, m, G)$
7                then $OD[m][p] = OD[m][p] \cup \{n\}$
8  return $OD$

DEPENDENCE$(n, m, p, G)$
 1  $color[|N|]$ :  a sequence of values ranging over $\{unknown,\ white,\ black\}$
 2  for  each $q$ in $N$
 3  do $color[q] \leftarrow uncolored$
 4  $color[m] = white$
 5  $color[p] = black$
 6  $visited \leftarrow \{m, p\}$
 7  COLORED-DAG$(G, n, color, visited)$
 8  $whiteChild \leftarrow false$
 9  $blackChild \leftarrow false$
10  for  each $q$ in $succs(n, G)$
11  do if $color[q] = white$
12         then $whiteChild \leftarrow true$
13      if $color[q] = black$
14         then $blackChild \leftarrow true$
15  return $whiteChild \wedge blackChild$

COLORED-DAG$(G, n, color, visited)$
 1  if $n \notin visited$
 2     then $visited \leftarrow visited \cup \{n\}$
 3            for  each $q$ in $succs(n, G)$
 4            do COLORED-DAG$(G, q, color, visited)$
 5            $c \leftarrow color[select(succs(n, G))]$
 6            for  each $q$ in $succs(n, G)$
 7            do if $color[q] \neq c$
 8                  then $c \leftarrow uncolored$
 9                       break
10            $color[n] \leftarrow c$
11  return
```

Figure 7: Algorithm to calculate decisively strong order dependence

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 line 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 atleast 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. Hence, either the loop at line 6 will terminate abnormally or terminate 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) iff all nodes reachable from $n$ in the DAG are white(black).* □

**Theorem 12** *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 the *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 $r$ be *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 out-going 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 backedges 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 above. Hence, `dependene()` 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 atleast 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 *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 graph reachability algorithm is $O(|N|^3)$. The loops at line 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 in [20]. 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 et.al [16] proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges. Later, Bilardi et.al [4] proposed new concepts related to control dependence along with algorithms based on these concepts to efficiently calculate weak control dependence In comparison, in this paper we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz, Reps, and Binkley [13] presented what has now become the standard approach to inter-procedural slicing via dependence graphs. However, in the last decade, C++, Java, and other languages that support semantically different exit points (exceptional and normal) to a procedure have become prominent. Hence, the work of Horwitz et.al cannot be applied directly as data dependence changes due to the semantic differences between the exit points/statements. This issue was recently addressed by Allen and Horwitz [1]. In their effort, they extend the previous work [13] to handle exception-based inter-procedural control flow. In this 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, the naive approach of considering the invocation site as a predicate (Soot [22] and [1]) and relating the `catch` statement with the corresponding `throw` statement via data dependence would suffice. If extra precision is required, then our definitions can be trivially applied to a collection of CFGs by tweaking the proposed algorithms to utilize the information about the connectivity between the nodes of different CFGs being considered.

For relevant work on slicing correctness, Horwitz et.al [12] use a semantics based multi-layered approach to reason about the correctness of slicing in the realm of data dependence. Alternatively, Ball et.al [3] used a program point specific history based approach to prove the correctness of slicing for arbitrary control flow. We build off of that work to consider arbitrary control flow without the unique end-node restriction. Their correctness property is a weaker property than bi-simulation – it does not require ordering to be maintained between observable nodes if there is no dependence between these nodes – and it holds for irreducible CFGs. For irreducible graphs, we need the extra notion of "order-dependency" to achieve the stronger correctness property.

In terms of handling dependences in a concurrent setting, Krinke [17] considered static slicing of multi-threaded programs with shared variables, and focused on issues associated with inter-thread data dependence but did not consider non-termination sensitive forms of control dependence. Millett and Teitelbaum [14] 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 that, as we argue in this paper, do not account for the subtleties of multi-threaded execution. They did not discuss the appropriateness of those notions for an inherently multi-threaded language like Promela, nor did they formalize a notion of correct slice for their applications. Hatcliff et.al [9] presented notions of dependence for concurrent CFGs to capture Java-like synchronization primitives. They proposed a notion of bi-simulation as the correctness property, but they did not provide a detailed definition or proof of correctness as has been done in this paper.

## 8 Conclusion

The notion of control dependence is used in myriads of applications, and researchers and 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 paper, 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 non-terminating.

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, and argued that any additional cost in computing these relations is negligible when one considers the cost and ill-effects of preprocessing steps required for previous definitions. Thus, 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 dependences extracted from execution traces and dependences extracted from control-flow graphs in an effort to systematically justify a comprehensive set of dependence notions for the rich features found in concurrent Java programs.

Also, the relation between order dependence and control dependence indicates that control reachability-based control dependence is a general case of control ordering-based order dependence. 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 high-level structure (source language) of programs from their intermediate forms (machine instructions) based on control flow orderings (like patterns).

## References

[1] M. Allen and S. Horwitz. Slicing Java programs that throw and catch exceptions. In *Procedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*, pages 44–54. ACM, June 2003.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Languages.* PhD thesis, DIKU, University of Copenhagen, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100, Copenhagen ∅, Denmark, May 1994.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging (AADEBUG'93)*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer-Verlag, 1993.

[4] G. Bilardi and K. Pingali. A framework for generalized control dependences. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96)*, pages 291–300, Philadelphia, Pennsylvania, United States, 1996. ACM Press New York, NY, USA.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 1999.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-state Models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448, June 2000.

[7] J. Ferrante, K. J. Ottenstein, and J. O. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[8] M. A. Francel and S. Rugaber. The relationship of slicing and debugging to program understanding. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 106–113, 1999.

[9] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings on the 1999 International Symposium on Static Analysis (SAS'99)*, Lecture Notes in Computer Science, Sept 1999.

[10] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Journal of Higher-order and Symbolic Computation*, 13(4):315–353, 2000. A special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.

[11] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journale of ACM*, 21(3):367–375, 1974.

[12] S. Horwitz, P. Pfeiffer, and T. W. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI'89)*, pages 28–40. ACM, 1989.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 1990.

[14] L. I.Millett and T. Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, 1998.

[15] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering Indus Java Program Slicer to Eclipse. Available at http://projects.cis.ksu.edu/docman/admin/index.php?group_id=12., October 2004.

[16] R. Johnson and K. Pingali. Dependence-based program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*, pages 78–89, 1993.

[17] J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, 1998.

[18] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989. ISBN: 0-13-115007-3.

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

[20] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(8):965–979, 1990.

[21] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B.Dwyer, and J. Hatcliff. A New Foundation For Control-Dependence and Slicing for Modern Program Structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005.* Springer-Verlag, April 2005. Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12.

[22] Sable Group. Soot, a Java Optimization Framework. This software is available at http://www.sable.mcgill.ca/soot/.

[23] SAnToS Laboratory. Indus, a toolkit to customize and adapt Java programs. This software is available at http://indus.projects.cis.ksu.edu.

[24] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

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