A New Foundation For Control-Dependence and Slicing for Modern Program Structures*

Technical Report #8

Venkatesh Prasad Ranganath, Torben Amtoft,
Anindya Banerjee, John Hatcliff
Department of Computing and Information Sciences,
Kansas State University †

Matthew B. Dwyer
Department of Computer Science and Engineering,
University of Nebraska ‡

October 29, 2004

Abstract

The notion of control dependence underlies many program analysis and transformation techniques used in numerous applications. Despite wide application, existing definitions and approaches to calculating control dependence are difficult to apply seamlessly to modern program structures. Such programs structures 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 control dependence that can be directly applied to modern program structures. In the context of slicing reactive systems, the paper proposes a notion of slicing correctness based on weak bisimulation and proves that the definition of control dependence generates slices that conform to this notion of correctness. Finally, a variety of properties show that the new definitions conservatively extend classic definitions. These new definitions and algorithms for control dependence form the basis of 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 used in numerous applications including program slicing applied for program understanding [18], debugging [7], and optimizations, partial evaluation [2], compiler optimizations [6] such as global scheduling, loop fusion, code motion etc. 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 and have been used in the current form for well over 20 years, there are several aspects of

*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) by Lockheed Martin, and by Intel Corporation.

†Manhattan KS, 66506, USA. {rvprasad,tamtoft,ab,hatcliff}@cis.ksu.edu

‡Lincoln NE, 68588-0115, USA. dwyer@cse.unl.edu

these definitions that prevent them from being applied smoothly to modern program structures which rely significantly on exception processing and increasingly support reactive systems which 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. Restriction (a) means 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. Restriction (b) means 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 [8, 18] 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. 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 nothing in the thread's control flow to indicate an exit point).

(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, distributed 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 entire 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 [9] 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 dependence that overcome the problematic aspects of the classic definitions described above. The specific contributions of this paper are as follows.

- We propose new definitions of control dependence that are simple to state and easy to calculate and that work directly on control-flow graphs that may have no end nodes or non-unique end nodes, thus avoiding troublesome pre-processing CFG transformations (Section 4).
- We prove that these definitions applied to reducible 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 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 discuss the intuitions behind algorithms for computing control dependence (according to the new definitions) to justify that control dependence is computable in polynomial time (Section 6).

Expanded discussions, definitions and full proofs appear in the companion technical report [19] which can be found on the project web site [21].

The proposed notions of control dependence described in this paper have been implemented in Indus-Kaveri [13, 21] – our publicly available open-source Eclipse-based Java slicer that works on full Java 1.4 and has been applied to code bases of up to 10,000 lines of Java application code ($<$ 80K bytecodes) excluding library code. Besides its application as a stand-alone program visualization, debugging, and code transformation tool, our slicer is being used in the next generation of our Bandera tool set for model-checking concurrent Java systems.

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 commands in program,
- the set of 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 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 where each $n_p \in N^P$ has two outgoing edges labeled T and F respectively, and each $n_s \in (N^S - N^E)$ has an outgoing edge labeled A (representing Always taken),
- the start node n_0 has no incoming edges and all nodes in N are reachable from n_0. □

We will display the labels on CFG edges only when necessary for the current exposition.

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 sometimes consider CFGs with the unique end node property in our comparisons 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 labels on the edges for n_p allow one to determine the nodes for the true and false branches of the conditional). 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* π from n_i to n_k is a sequence of nodes $n_i, n_{i+1}, \ldots, n_k$ such for every consecutive pair of nodes (n_j, n_{j+1}) in the path there is an edge from n_j to n_{j+1}. A path between nodes n_i and n_k can also be denoted as $[n_i..n_k]$. When the meaning is clear from the context, we will use π to denote the set of nodes contained in π and we write $n \in \pi$ when n occurs in the sequence π. Path π

is *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 [17]. 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 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 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 [18]. 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 e (and thus n post-dominates m), it may be the case that there is a loop in the CFG between m and n that admits an infinite path beginning at m that never encounters n. Strong post-domination rules out the possibility of such loops between m and n – thus, it is sensitive to the possibility of non-termination along paths from m to n. Note that domination relations are well-defined but post-domination relationships are not well-defined for graphs that do not have the unique end node property.

A CFG G 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 give rise to reducible control-flow graphs, including Java programs. Our definitions and most of our correctness results apply to irreducible CFGs as well, but our bi-simulation-based correctness of slicing result only holds for reducible graphs since bi-simulation requires ordering properties that can only be guaranteed on reducible graphs.

2.2 Program Execution

The execution semantics of program CFGs is phrased in terms of transitions on program states (n, σ) where n is a CFG node and σ is a store mapping the corresponding program's variables to values. A series of transitions gives an *execution trace* through p's statement-level control flow graph. It is important to note that when execution is in state (n_i, σ_i), the code at node n_i has not yet been executed. Intuitively, the code at n_i is executed on the transition from (n_i, σ_i) to successor state (n_{i+1}, σ_{i+1}). Execution begins at the state node n_0, and the execution of each node possibly updates the store and transfers control to an appropriate successor node. Execution of a node $n_e \in N^E$ produces a final state (halt, σ) where the control point is indicated by a special label halt – this indicates a normal termination of program execution. The presentation of slicing in the next section involves arbitrary finite and infinite non-empty sequences of states written $\Pi = s_1, s_2, \ldots$. For a set of variables V, we write $\sigma_1 =_V \sigma_2$ when for all $x \in V$, $\sigma_1(x) = \sigma_2(x)$.

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 [22]. 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 recall the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [22].

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

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

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 π 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. □

For a node n to be control-dependent on predicate m, there must be two paths that connect m with the unique end node 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. At present, we simply note that the above definition is standard and widely used (e.g., see [17]).

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

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

□

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 also be applied we computing forward slices.

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* [23]. Informally, given an arbitrary trace Π of p and an analogous trace Π_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 Π and Π_s yields identical state sequences. We will consider slicing correctness requirements in greater detail in Section 5.1.

3 Assessment of Existing Definitions

3.1 Variations in Existing Control Dependence Definitions

Although the definition of control dependence that we stated in Section 2 is widely used, there are a number of (sometimes subtle) variations appearing in the literature. One dimension of variation is whether the particular definition captures only *direct* control dependence or also admits *indirect* control dependences. For example, using the definition of control dependence in Definition 3, for Figure 1 (a), we can conclude that $a \overset{cd}{\rightarrow} f$ and $f \overset{cd}{\rightarrow} g$ however $a \overset{cd}{\rightarrow} g$ 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, some definitions of control dependence [18] incorporate this notion of transitivity directly into the definition itself as we will illustrate later.

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, $a \stackrel{cd}{\rightarrow} d$ but $c \stackrel{cd}{\rightarrow} d$ 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 [18] and used in a number of works including the study of control dependence by Bilardi and Pingali [4].

Definition 5 (Podgurski-Clarke Control Dependence)

- n_2 is *strongly control dependent* on n_1 ($n_1 \stackrel{PC-scd}{\rightarrow} n_2$) if there is a path from n_1 to n_2 that does not contain the immediate post dominator of n_1.
- n_2 is *weakly control dependent* on n_1 ($n_1 \stackrel{PC-wcd}{\rightarrow} n_2$) 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 strong control dependence above roughly corresponds to Definition 3, but it captures indirect control dependence whereas Definition 3 captures only direct control dependence. For example, in Figure 1, in contrast to Definition 3 we have $a \stackrel{PC-scd}{\rightarrow} g$ because there is a path afg which does not contain the immediate post-dominator of a. However, one can show 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.

The notion of weak control dependence above subsumes the notion of strong control dependence ($n_1 \stackrel{PC-scd}{\rightarrow} n_2$ implies $n_1 \stackrel{PC-wcd}{\rightarrow^*} n_2$) and it captures weaker dependences between nodes induced by non-termination, that is, it is non-termination sensitive. Note that for Figure 1 (a), $c \stackrel{PC-wcd}{\rightarrow} d$ because 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.

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 $C = \{d\}$ would include a but not b and c (we assume no data dependence between d and b or c) if the slicing is based on strong control dependence. 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 seem to largely technical stylistic decision in how the definitions are stated.

Very few works 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 stronger relation (relating more nodes) and will thus include more nodes in the slice. 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 [9].

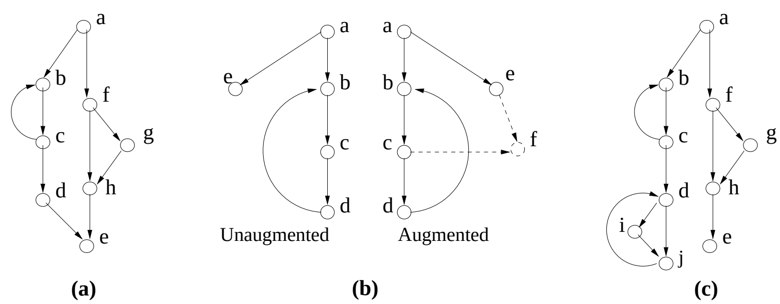

Figure 1: (a) is a simple CFG. (b) illustrates how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have unique exit node reachable from all nodes. (c) is a CFG with multiple control sinks of different sorts.

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. If the augmentation is non-destructive, a new CFG is generated which costs time and memory. If the augmentation is destructive, this may clash with the requirements of other clients of the CFG, thus necessitating the reversal of the augmentation before subsequent analyses can proceed. In addition, having multiple end nodes (e.g., an exceptional exit and a regular return) flow into a single new end node causes semantically different information to flow together.

Many systems have threads where the main control loop has no exit – the loop is "exited" by simply killing the thread. For example, in 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 once it starts, and hence, not terminate except when it is 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. However, 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, $a \overset{PC-wcd}{\rightarrow} \{e, b, c\}$ and $c \overset{PC-wcd}{\rightarrow} \{b, c, d, f\}$ in the augmented CFG. It is trivial to prune dependences involving f. However, there are new dependences $c \overset{PC-wcd}{\rightarrow} \{b, c, d\}$ 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, $a \overset{PC-wcd}{\rightarrow} d$ exists in the unaugmented CFG but not in the augmented CFG, and it is not obvious how to recover this information.

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

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 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 6 ($n_i \stackrel{ntscd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) non-termination sensitive control dependent** on node n_i if n_i has at least two successors, n_k and n_l,

(1) for all maximal paths from n_k, n_j always occurs and it occurs before any occurrence of n_i.
(2) 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. □

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, E and A, which define that a path from a given node with a given structure exists or that 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 ϕ as a ϕ-node): $\mathsf{X}\phi$ states that the successor node is a ϕ-node, $\mathsf{F}\phi$ states the existence of a ϕ-node, $\mathsf{G}\phi$ states that a path consists entirely of ϕ-nodes, $\phi\mathsf{U}\psi$ states the existence of a ψ-node and that the path leading up to that node consists of ϕ-nodes, finally, the $\phi\mathsf{W}\psi$ operator is a variation on U that relaxes the requirement that a ψ-node exist. In a CTL formula path quantifiers and modal operators occur in pairs, e.g., $\mathsf{AF}\phi$ says on all paths from a node a ϕ node occurs. A formal definition of CTL can be found in [5].

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

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

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

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

See [19] for the proof that these two definitions are equivalent on CFGs.

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

We now turn to constructing a non-termination insensitive version of control dependence. The definition above 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, a *control sink* κ is a set of CFG nodes that form a strongly connected component such that for each $n \in \kappa$ each successor of n is also in κ. It is trivial to see 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). *c-sink* denotes a set-valued function on nodes such that $\textit{c-sink}(n) = S$ where if n belongs to a control sink then S is set of nodes representing that sink, otherwise $S = \emptyset$.

For a control flow graph, its strongly connected components form a DAG, which the control sinks being the leaves. This shows:

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

Existing definitions 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. The set of *sink-bounded paths from* n_k (denoted $\textit{SinkPaths}(n_k)$) contains all π such that π is a path from n_k to a node n_s such that n_s belongs to a control sink.

Definition 7 ($n_i \overset{nticd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) non-termination insensitively control dependent** on n_i if n_i has at least 2 successors, n_k and n_l,

(1) for all paths $\pi \in \textit{SinkPaths}(n_k)$, $n_j \in \pi$.
(2) there exists a path $\pi \in \textit{SinkPaths}(n_l)$ such that $n_j \notin \pi$ and if π leads to a control sink κ, $n_j \notin \kappa$. □

This definition is expressed in CTL as

$$n_i \overset{nticd}{\rightarrow} n_j \quad = \quad (G, n_i) \models \mathsf{EX}(\hat{\mathsf{A}}\mathsf{F}(n_j)) \wedge \mathsf{EX}(\hat{\mathsf{E}}[\neg n_j \mathsf{U}(\textit{c-sink?} \wedge n_j \notin \textit{c-sink})])$$

where $\hat{\mathsf{A}}$ and $\hat{\mathsf{E}}$ represent quantification over sink-bounded paths only. *c-sink?* evaluates to *true* only if the current node belongs to a control sink and *c-sink* returns the sink set associated with the current node.

To see that this definition is non-termination insensitive, note that $c \overset{nticd}{\not\rightarrow} d$ in Figure 1 (a) since there does 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 there is a trivial path from d to the control sink $\{d, i, j\}$, i belongs to that control sink. This is because our definition inherently captures a form of fairness – since the backedge from j guarantees that d will be executed an infinite number of times, the only way to avoid executing i would be to branch to d on every cycle. 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 would apply the definition to control sinks in isolation with back edges removed.

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 by pass d. However, in case of $c \overset{cd}{\rightarrow} d$, c does *decisively* decide if control can by pass d. The decisiveness stems from the fact that there is a choice at the control point such that it prevents the control from reaching the given program point before reaching the control point. Hence, the relation can be defined as follows.

Definition 8 ($n_i \overset{dcd}{\rightarrow} n_j$) In a CFG, n_j is **(directly) decisively control dependent** on node n_i if n_i has at least two successors, n_k and n_l,

(1) for all maximal paths from n_k, n_j always occurs and 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. □

Observe that the above definition implies Definition 6.

This stronger form of control dependence is useful to answer the questions - *"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.

4.1 Examples

Consider Figure 1 (c). According to Definition 6, $a \overset{ntscd}{\rightarrow} b$ 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, $a \overset{ntscd}{\rightarrow} e$. b can be executed $n+1$ times and value of n depends on the choice at c. Hence, $c \overset{ntscd}{\rightarrow} b$. 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 by pass i in an iteration while it reaches i in a subsequent iteration depending on the choice made at d. Hence, $d \overset{ntscd}{\rightarrow} i$.

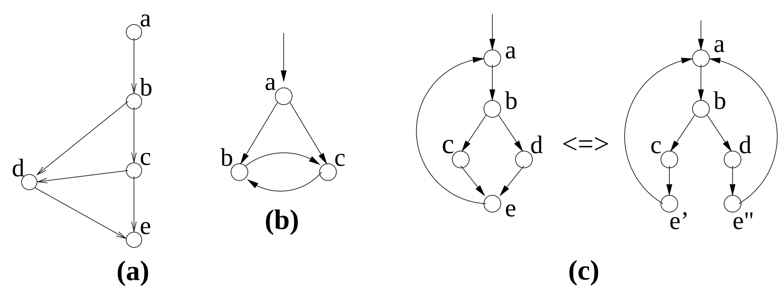

Figure 2: More control flow graphs.

In a non-termination insensitive setting, loops are assumed to be terminal if structurally valid (i.e., if the loop has an exit node). Hence, in Figure 1 (c), the loop $b \rightarrow c \rightarrow b$ is terminal 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 dependences steming in a non-termination sensitive setting for the same graph, most of them hold except $d \overset{ntscd}{\rightarrow} i$. To understand why, observe that the loop starting at d can be split into 2 loops as done in Figure 2 (c). Upon loop splitting, each loop is terminal (but both loops together are not terminal). Hence, there can be no control dependence in the loop starting at d (or in a control sink) in a non-termination insensitive setting.

4.2 Properties of the Dependence Relations

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. In addition, direct non-termination insensitive control dependence (Definition 7) implies the *transitive closure* of direct non-termination sensitive control dependence.

Theorem 1 (Coincidence Properties) *For all CFGs with the unique end node property, and for all nodes $n_i, n_j \in N$,*

(1) $n_i \neq n_j$ and $n_i \overset{cd}{\rightarrow} n_j$ implies $n_i \overset{nticd}{\rightarrow} n_j$

(2) $n_i \overset{nticd}{\rightarrow} n_j$ *implies* $n_i \overset{cd}{\rightarrow} n_j$

(3) $n_i \overset{PC-wcd}{\rightarrow} n_j$ *iff* $n_i \overset{ntscd}{\rightarrow} n_j$

(4) For all CFGs, for all nodes $n_i, n_j \in N : n_i \overset{nticd}{\rightarrow} n_j$ *implies* $n_i \overset{ntscd^*}{\rightarrow} n_j$ □

First, we rephrase condition (2) of Definition 3 by expanding the definition of strictly postdominates:

- Either $n_i = n_j$, or there exists a non-trivial path π from n_i to the end node, e, such that n_j does not occur on this path.

Clearly, from Definition 3, $n_i \overset{cd}{\rightarrow} n_i$. Next, we have the following fact.

Fact 1 $n_i \overset{cd}{\rightarrow} n_j$ *and* $n_i \neq n_j$ *implies* n_i *has at least two successors.*

To see this fact, suppose, towards a contradiction, that $n_i \overset{cd}{\rightarrow} n_j$ and $n_i \neq n_j$ and n_i has a single successor n_k. Consider path π from n_i to n_j. By condition (1) of Definition 3, any path from n_k to the end node must pass through n_j. Thus any path from n_i to the end node must pass through n_j. This contradicts condition (2) of Definition 3, as then n_i is strictly postdominated by n_j.

Thus we can restate the two conditions of Definition 3 as follows:

cd(i) n_i has at least two successors n_k and n_l.

cd(ii) there exists a path $\pi = n_i\, n_k \ldots n_j$ such that every node $m' \in \pi - \{n_i, n_j\}$ is post-dominated by n_j, and

cd(iii) there exists a path $\pi = n_i\, n_l \ldots e$ such that n_j does not occur on this path.

We now begin the proof of Theorem 1.

PROOF *Proof of (1)* Assume $n_i \overset{cd}{\rightarrow} n_j$ and $n_i \neq n_j$. Then n_i has at least two successors, n_k and n_l. Let π be a sink-bounded path from n_k. The unique end node is the only control sink in the CFG. We have two cases: (a) π is finite and ends with the end node: then as n_k is postdominated by n_j, n_j always occurs on π. (b) π is infinite: then π is not sink-bounded. Thus this case does not occur.

Let π' be a path from n_l to the end node such that n_j does not occur on this path. Clearly, π' is sink-bounded. And, $n_j \notin \pi'$.

Hence $n_i \overset{nticd}{\rightarrow} n_j$.

Proof of (2). We have $n_i \overset{cd}{\rightarrow} n_j$ when $n_i = n_j$. So, consider the case $n_i \neq n_j$. Assume $n_i \overset{nticd}{\rightarrow} n_j$.

Assume that on all sink-bounded paths from n_k, n_j always occurs. Towards a contradiction, assume that for any path π from n_i to n_j, there exists a node $m' \in \pi - \{n_i, n_j\}$ such that there exists a path from m' to the end node, e, not containing n_j. Consider the path $n_i\, n_k \ldots m' \ldots e$. This path is a sink-bounded path from n_k not containing n_j. Contradiction.

Assume there exists a sink-bounded path π from n_l such that $n_j \notin \pi$ and if π leads to a control sink κ, $n_j \notin \kappa$. Towards a contradiction, assume that for any path from n_i to the end node, e, n_j occurs on this path. Since n_l is a successor of n_i, every path from n_l to e is sink-bounded and contains n_j. Thus there does not exist a sink-bounded path from n_l such that $n_j \notin \pi$. Contradiction.

Hence $n_i \overset{nticd}{\rightarrow} n_j$.

Proof of (3). For readability, we restate Podgurski-Clarke's definition of weak control dependence from Definition 5 and directly non-termination sensitive control dependence from Definition 6. We have $n_i \overset{PC-wcd}{\rightarrow} n_j$ 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.

Next, $n_i \overset{ntscd}{\to} n_j$ 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 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.

We will prove the equivalence by showing that both Definition 6 and Podgurski-Clarke's weak control dependence are equivalent to the following simplified definition:

Definition 9 In a CFG, n_j is 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.

First, let us argue that non-termination sensitive control dependence (Definition 6) and the simplified definition (Definition 9) are equivalent.

Since clearly **ntscd(ii)** implies **(b)** and **(c)** implies **ntscd(iii)**, we are left with showing that:
(b) implies **ntscd(ii)**: Let π be a maximal path from n_k. By **(b)**, n_j occurs there. Now assume, towards a contradiction, that in π, n_i occurs strictly before any occurrence of n_j. Since there is an edge from n_i to n_k, this means that the graph has a cycle containing n_k but not containing n_j. But then we can find a maximal path from n_k where n_j does not occur, contradicting **(b)**.

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

Now we show that Podgurski-Clarke's direct weak control dependence and the simplified definition, Definition 9, are equivalent. Then we can conclude that $n_i \overset{PC-wcd}{\to} n_j$ iff $n_i \overset{ntscd}{\to} n_j$. There are four steps.

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

4. **(c)** implies **pcwcd(iii)**: Our assumption is that there exists a maximal path π from n_l with n_j not occurring in π. Now there are two cases:
   π is finite, with the last node being an end node: But then n_j does not postdominate n_l, in particular n_j does not strongly postdominate n_l.
   π is infinite: But then for any k, π will be a path from n_l of length k not containing n_j, again showing that n_j does not strongly postdominate n_l.

Proof of (4). 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 $\pi = n_i, n_k, \ldots, n_j$ (there exists such a path, by Lemma 1). We can write $\pi = [u_0, u_1, \ldots, u_m]$ where $m \geq 1$, $u_0 = n_i$, $u_1 = n_k$, $u_m = n_j$. Observe that for all $i = 1 \ldots m$, n_j occurs on all sink-bounded paths from u_i to n_j (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 $i \leq p$ 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 $u_i \overset{ntscd}{\rightarrow} u_p$. Also observe that for all $i = 1 \ldots m$ we have $Q_p(p)$ holds, but $Q_p(0)$ does not hold. Now we are ready for the construction: 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 $j_1 = 0$, we are done. Otherwise, since $Q_{j_1}(j_1)$ holds but $Q_{j_1}(j_1+1)$ 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. ∎

For the correctness (bisimulation-based) proof in Section 5.1, we shall need a few results about slice sets (members of which are "observable"). A crucial property is that the first observable on any path will be encountered sooner or later on all other paths:

Lemma 2 *Assume the node set Ξ is closed under termination sensitive control dependency, and that $n_0 \notin \Xi$. Assume that there is a path π from n_0 to n_1, with $n_1 \in \Xi$ but for all $n \in \pi$ with $n \neq n_1$, $n \notin \Xi$. Then all maximal paths from n_0 will contain n_1.* □

Proof Assume, in order to arrive at a contradiction, that there exists a maximal path from n_0 that does not contain n_1. We define a predicate Q, such that $Q(n)$ holds iff there exists a maximal path from n that does not contain n_1. By our assumption, $Q(n_0)$ holds; clearly, $Q(n_1)$ does not hold. Therefore, π can be written as $[n_0..n_2n_3..n_1]$ where $Q(n_2)$ holds but $Q(n_3)$ does not hold (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 $n_1 \in \Xi$ we from Ξ being closed under $\overset{ntscd}{\rightarrow}$ get $n_2 \in \Xi$ which contradicts n_1 being the only node in π which is also in Ξ.

Note that since $Q(n_2)$ holds, 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 $Q(n_3)$ does not hold). 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 is to prove that all maximal paths from n_3 contain n_1, which follows since $Q(n_3)$ does not hold. ∎

In a similar way we can show:

Lemma 3 *Assume Ξ is closed under $\overset{nticd}{\rightarrow}$, and that $n_0 \notin \Xi$. Assume that there is a path π from n_0 to n_1, with $n_1 \in \Xi$ but for all $n \in \pi$ with $n \neq n_1$, $n \notin \Xi$. 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 4 *Assume that $n_0 \notin \Xi$, but that there is a path π starting at n_0 which contains a node in Ξ.*

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

We are now ready for the main result, stating that from a given node there is a unique first observable. (For this, we need the CFG to be reducible; the irreducible graph in Fig. 2,(b) provides a counterexample.)

Theorem 2 *Assume that $n_0 \notin \Xi$, that $n_1, n_2 \in \Xi$, and that there are paths $\pi_1 = [n_0..n_1]$ and $\pi_2 = [n_0..n_2]$ such that on both paths, all nodes except the last do not belong to Ξ.*

If Ξ is closed under termination insensitive control dependency (a weaker requirement than being closed under termination sensitive control dependency), and if the CFG is reducible, then $n_1 = n_2$. □

Proof By Lemma 1, we can extend π_1 and π_2 into sink-bounded paths π_1' and π_2'. By Lemma 3, we infer that π_2' contains n_1, and that π_1' contains n_2. If $n_1 \neq n_2$, this implies that n_1 is reachable from n_2, and vice versa, while both being reachable from n_0, something which cannot happen in a reducible graph. ■

5 Slicing

We now describe how to slice a (reducible) CFG G wrt. a slice set S_C, the smallest set containing C which is closed under data dependence $\stackrel{dd}{\rightarrow}$ and also under some kind of control dependence: at least we must require it is closed under $\stackrel{nticd}{\rightarrow}$, but a stronger correctness property (Sect. 5.1) holds if it is also closed under $\stackrel{ntscd}{\rightarrow}$.

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 $code_2(n) = code_1(n)$ for $n \in S_C$; for $n \notin S_C$ then

- 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 criteria C represent "observations" that one is making about a CFG G under consideration. Specifically, for a $n \in C$, 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 τ-moves [16]. In our setting, we shall consider transitions that do one or more steps before arriving at a node in the slice set.

Definition 10 For $i = 1, 2$ we write $s \overset{i}{\longmapsto} s'$ to denote that wrt. code map $code_i$, the program state s rewrites in one step to s'.

For $i = 1, 2$ we write $s_0 \overset{i}{\Longrightarrow} s$ if there exists $s_1 \dots s_k$ ($k \geq 1$) with $s_k = s$ such that (with each $s_j = (n_j, \sigma_j)$)

- for all $j \in \{1 \dots k\}$ we have $s_{j-1} \overset{i}{\longmapsto} s_j$;
- $n_k \in S_C$ but for all $j \in \{1 \dots k-1\}$, $n_j \notin S_C$. □

Definition 11 A binary relation $\mathcal{S}$ on program states is a bisimulation if whenever $(s_1, s_2) \in \mathcal{S}$ then

(a) if $s_1 \overset{1}{\Longrightarrow} s_1'$ then there exists a s_2' such that $s_2 \overset{2}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in \mathcal{S}$, and
(b) if $s_2 \overset{2}{\Longrightarrow} s_2'$ then there exists a s_1' such that $s_1 \overset{1}{\Longrightarrow} s_1'$ and $(s_1', s_2') \in \mathcal{S}$.

If instead of (b) we only have (c) below, we say that $\mathcal{S}$ is a quasi-bisimulation.

(c) if $s_2 \overset{2}{\Longrightarrow} s_2'$ then either $s_1 \overset{1}{\not\Longrightarrow}$ or there exists a s_1' such that $s_1 \overset{1}{\Longrightarrow} s_1'$ and $(s_1', s_2') \in \mathcal{S}$. □

For each node n in G, we define $relv(n)$, the set of relevant variables at n, by stipulating that $x \in relv(n)$ if there exists a node $n_k \in S_C$ and a path π from n to n_k such that $x \in refs(n_k)$, but $x \notin defs(n_j)$ for all nodes n_j occurring before n_k in π.

The above is well-defined in that it does not matter whether we use $code_1$ or $code_2$, as it is easy to see that the value of $relv(n)$ is not influenced by the content of nodes not in S_C, since that set is closed under $\overset{dd}{\rightarrow}$. (Also, the closedness properties of S_C are not affected by using $code_2$ rather than $code_1$.)

We are now ready to state the correctness theorem:

Theorem 3 *Let the relation $\mathcal{S}_0$ be given by $(n_1, \sigma_1)\, \mathcal{S}_0\, (n_2, \sigma_2)$ iff $n_1 = n_2$ and $\sigma_1 =_{relv(n_1)} \sigma_2$. Then (if G is reducible)*

- *$\mathcal{S}_0$ is a quasi-bisimulation;*
- *$\mathcal{S}_0$ is even a bisimulation if S_C is closed under $\overset{ntscd}{\rightarrow}$.* □

Proof (Sketch.) We must consider transitions of the form $(n, \sigma_i) \overset{i}{\Longrightarrow} (n', \sigma_i')$; that is we have $(n, \sigma_i) \overset{i}{\longmapsto} (n'', \sigma_i'')$ and either $n'' = n'$ or $(n'', \sigma_i'') \overset{i}{\Longrightarrow} (n', \sigma_i')$.

With $j = 3 - i$, our general goal is to simulate the above transition wrt. $code_j$. For three cases, listed below, we find σ_j'' such that $(n, \sigma_j) \overset{j}{\longmapsto} (n'', \sigma_j'')$ with $\sigma_i'' =_{relv(n'')} \sigma_j''$: then we are done if $n'' = n$; otherwise we apply inductive reasoning.

$n \in S_C$ Here $\sigma_i =_{ref(n)} \sigma_j$. Therefore, if n is a predicate, the same branch will be taken; if n is a statement, the stores will be updated with the same value.

$n \notin S_C$ is a statement Here $code_2(n) =$ `skip`, and the claim follows since the value stored by $code_1(n)$ will not belong to $relv(n'')$ (as S_C is closed under $\overset{dd}{\rightarrow}$).

$n \notin S_C$ is a predicate, $i = 1$ Then $code_2(n) =$ `cskip`, and the claim is trivial.

We are left with the interesting case where $n \notin S_C$ is a predicate, $i = 2$. Two subcases:

- S_C is closed under $\stackrel{ntscd}{\to}$; we must show (b) of Definition 11. But Lemma 4 tells us that there exists n_1, σ'_1 such that $(n, \sigma_1) \stackrel{1}{\Longrightarrow} (n_1, \sigma'_1)$, where $n_1 = n'$ by Theorem 2. For $x \in relv(n')$, we have to show that $\sigma'_1(x) = \sigma'_2(x)$, which follows since such variables cannot be modified along the way (again since S_C is closed under $\stackrel{dd}{\to}$).
- otherwise, we only have to show (c) of Definition 11, so assume that there exists n_1, σ'_1 such that $(n, \sigma_1) \stackrel{1}{\Longrightarrow} (n_1, \sigma'_1)$. By Theorem 2 we infer that $n_1 = n'$, and we proceed as in the previous case. ■

6 Algorithms

In this section we provide the intuition behind the algorithms to calculate various forms of control dependences that were presented earlier. This includes a an overview of the main processing steps of the algorithm to calculate non-termination sensitive control dependence and its adaptation to computing other forms of dependences.

A complete description of the algorithms along their correctness and complexity analysis are given in Section 6. In this section we provide an overview of its main processing steps and its adaptation to computing other forms of dependence.

6.1 Non-Termination Sensitive Control Dependence

Control dependences are calculated using a symbolic data-flow analysis. Fundamentally, 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_1 with more than one successor in G, the set of paths starting at n_1 that begin with $n_1 \to n_2$ is represented by $t_{n_1 n_2}$. The algorithm propagates these symbolic values to collect the effects of particular control flow choices at program points in the CFG. For each node n_3 in the CFG a set of symbolic values, $S_{n_3 n_1}$, is stored for each node n_1 in the CFG that has more than one successor; these sets record the set of paths that originate from n_1. The algorithm preserves the invariant that $t_{n_1 n_2} \in S_{n_3 n_1}$ if and only if all non-trivial non-terminal paths or terminal paths starting from n_1 with $n_1 \to n_2$ contain node n_3.

Let T_{n_1} denote the outdegree of n_1 and $condNodes(G)$ denote the set of nodes with outdegree greater than one.

The algorithm is initialized such that, for each node $n_1 \in condNodes(G)$, $t_{n_1 n_2}$ is inserted into $S_{n_2 n_1}$ for each successor n_2 of n_1 and n_2 is marked for processing. The algorithm then proceeds by executing the following three steps for each marked node n_3; it terminates when there are no longer any marked nodes.

1. For each node $n_1 \in condNodes(G) \backslash n_3$, if $|S_{n_3 n_1}| = T_{n_1}$ then, for each node $n_4 \in condNodes(G) \backslash n_3$, all symbols from $S_{n_1 n_4}$ are inserted into $S_{n_3 n_4}$. This captures the property that if all non-terminal paths or terminal paths that end in exit nodes from every successor of n_4 contains n_1, then these paths will also contain n_3.
2. Depending on the number of successors of n_3, one of the following actions is performed if any $S_{n_3 n_5}$ was changed.

   $|\boldsymbol{succs}(\boldsymbol{n}_3)| = 1$ Let n_5 be the successor of n_3. For each node $n_4 \in condNodes(G)$ such that $S_{n_5 n_4} \backslash S_{n_3 n_4} \neq \emptyset$, insert $S_{n_3 n_4}$ into $S_{n_5 n_4}$ and add n_5 into the worklist. This captures the property that all non-terminal paths or terminal paths that end in exit nodes that contain n_3 will also contain n_5.

$|\boldsymbol{succs(n_3)}| > 1$ For each node n_4, if $|S_{n_4 n_3}| = T_{n_3}$ then n_4 is marked for processing. This captures the requirement that any path information change at n_3 needs to be considered at each node n_4 that will occur on all non-terminal paths or terminal paths that end in exit nodes starting from n_3.

3. Unmark n_3.

When there are no more marked nodes, all-path reachability information for every pair of nodes, n_3 and n_1 (with outdegree greater than one), in the graph is available in $S_{n_3 n_1}$. The presence of a token $t_{n_1 n_2}$ in $S_{n_3 n_1}$ indicates that all non-terminal paths or terminal paths that end in exit nodes starting with the edge $n_1 \rightarrow n_2$ contain n_3. So, if $|S_{n_3 n_1}| > 0 \wedge |S_{n_3 n_1}| \neq T_{n_1}$ then, by Definition 6, it can be inferred that n_3 *is directly control dependent on* n_1. On the other hand, if $|S_{n_3 n_1}| > 0$ and $|S_{n_3 n_1}| = T_{n_1}$ then, by Definition 6, it can be inferred that n_3 *is not directly control dependent on* n_1.

6.1.1 A Walk-through

Consider the CFG in Figure 1 (c). In phase (1), t_{ab} and t_{af} are injected into S_{ba} and S_{fa}, respectively. Likewise, t_{cb}, t_{cd}, t_{fg}, t_{fh}, t_{di}, and t_{dj} are injected into S_{ba}, S_{fa}, S_{bc}, S_{dc}, S_{id}, and S_{jd}, respectively, and *b, d, i, j, g,* and *h* are marked for processing. As *b* has only one successor and $S_{cc} \backslash S_{bc} = \{t_{cb}\}$, t_{cb} is injected into S_{cc} and *c* is marked for processing.

Similarly, as *g* has only one successor and $S_{gf} \backslash S_{hf} = \{t_{fg}\}$, t_{fg} is injected into S_{hf} and *h* is marked for processing. As $|S_{hf}| = T_f$ ($S_{hf} = \{t_{fg}, t_{fh}\}$) , t_{af} is injected into S_{ha} as $S_{fa} \backslash S_{ha} = \{t_{af}\}$. Similarly, $t_{cd} \in S_{jc}$.

After the propagation stops, the algorithm decidse *h* is directly non-termination sensitively control dependent on *a* as $|S_{ha}| = T_a = 2$. Similarly, the algorithm decides $a \stackrel{ntscd}{\rightarrow} b, c \stackrel{ntscd}{\rightarrow} b, c \stackrel{ntscd}{\rightarrow} c, c \stackrel{ntscd}{\rightarrow} d, d \stackrel{ntscd}{\rightarrow} i, c \stackrel{ntscd}{\rightarrow} j, a \stackrel{ntscd}{\rightarrow} f, f \stackrel{ntscd}{\rightarrow} g, a \stackrel{ntscd}{\rightarrow} h$ and $a \stackrel{ntscd}{\rightarrow} e$. As $|S_{jd}| = T_d = 2$, the algorithm decides *j* is not directly non-termination sensitively control dependent on *d*. Similarly, the algorithm decides $f \stackrel{ntscd}{\not\rightarrow} h$. For all other combinations, as $|S_{xy}| = 0$, the algorithm decides $y \stackrel{ntscd}{\not\rightarrow} x$.

6.2 Non-Termination Insensitive Control Dependence

As Definition 7 implies indirect variant of Definition 6, we can prune indirect non-termination sensitive control dependence to arrive at non-termination insensitive control dependence. For each pair of node n_1 and n_2 such that $n_2 \stackrel{ntscd}{\rightarrow} n_1$, the following steps decide if the $n_2 \stackrel{nticd}{\rightarrow} n_1$.

1. As observed, no node can be non-termination insensitively control dependent on a node in the control sink. Hence, if n_2 belongs to a control sink then $n_2 \stackrel{nticd}{\not\rightarrow} n_1$. Also, if n_1 belongs to a control sink and there is only one control sink in the CFG then $n_2 \stackrel{nticd}{\not\rightarrow} n_1$. If not, proceed to the next step.

2. Explore the graph (using DFS or BFS) starting from n_2 without traversing any edges incident on n_1. The exploration is terminated when the graph is exhausted or when a node in a control sink that does not contain n_1 is encountered. In the former case, there are no paths from n_2 to a control sink not containing n_1, hence, $n_2 \stackrel{nticd}{\not\rightarrow} n_1$. The opposite is true in the latter case.

6.2.1 A Walk-through

Again consider the CFG in Figure 1 (c). Clearly, $c \stackrel{ntscd^*}{\rightarrow} i$. Also, note that *e* is the only control sink node that does not belong to the control sink containing *i*. Upon exploring the CFG along all outgoing edges from *c* without exploring edges emanating from *i*, $\{b, d\}$ are reachable. Of these, *b* is not a control sink node and *d* is a control sink node. Upon reaching *d*, as *d* belongs to the control sink containing i, the algorithm will not consider d. Hence, as there is no path from c to a control sink not containing i, $c \stackrel{nticd}{\not\rightarrow} i$.

On the other hand, $a \stackrel{ntscd^*}{\rightarrow} i$. Upon exploring the CFG along all outgoing edges from a without exploring edges emanating from i, $\{b, c, d, f, g, h, e\}$ are reachable. Upon reaching e, as e is a control sink node that does not belong to the control sink containing i, the algorithm will conclude $a \stackrel{nticd}{\rightarrow} i$.

6.3 Decisive Control Dependence

As Definition 8 implies Definition 6, we can prune non-termination sensitive control dependence to arrive at decisive control dependence. The pruning condition is the negative form of the third clause in Definition 8 – for each successor n_l of n_i, there exists a maximal path such that n_j occurs before any occurrence of n_i.

We use an algorithm similar to Figure 3 to calculate if there is a path from a successor of a conditional node to a given node with no occurrences of the conditional node. The basic idea is to represent each edge emanating from a conditional node n_1 (to n_2) by a token, $t_{n_1 n_2}$, and then to propagate the token to nodes reachable from n_2. However, no token $t_{n_1 n_i}$ will be propagated from n_1 to it's successors except in the initialization phase. Hence, if a $t_{n_1 n_2}$ is present in $S_{n_3 n_1}$ then it should be the case that there exists a path from n_2 to n_3 that does not contain n_1.

6.3.1 A Walk-through

Consider the CFG in Figure 2 (a). Clearly, $b \stackrel{ntscd}{\rightarrow} d$. In the above algorithm, t_{bc} and t_{bd} is injected into S_{cb} and S_{db}. In contrast to the algorithm for non-termination sensitive control dependence, t_{bc} will be injected into S_{db} as there is a path from c to d. Hence, as $|S_{db}| = T_b = 2$, the algorithm decides $b \stackrel{dcd}{\not\rightarrow} d$.

On the other hand, similar situation occurs for d and c - $c \stackrel{ntscd}{\rightarrow} d$ and t_{cd} and t_{ce} is injected into S_{dc} and S_{ec}. However, t_{ce} can never reach S_{dc} as there is no path from e to d. Hence, as $|S_{dc}| \neq T_c = 2$, the algorithm decides $c \stackrel{dcd}{\rightarrow} d$. However, note that if there was a back edge from e to b, then t_{ce} can reach S_{dc}, in which case $c \stackrel{dcd}{\not\rightarrow} d$.

6.4 Complexity

The proposed algorithms have a worst-case asymptotic complexity of $O(|N|^3 \times K)$ where K is the sum of the outdegree of all nodes with more than one successor in the CFG. Linear time algorithms to calculate control dependence have been proposed in the literature [18]. These algorithms, however, rely on augmentation of the CFG. The practical cost of this augmentation varies with the specific algorithm and control dependence being calculated. Our experience with an implementation of our general algorithms in a program slicer for full Java suggests that, despite its complexity bound, it can be scaled to programs with tens-of-thousands of lines of code and still return results in a matter of seconds. We suspect that this is due in part to the elimination of the need for augmenting CFGs in our approach.

In the proposed approach, none of the above mentioned issues arise. In fact, the proposed algorithms merely justifies that it is possible to calculate control dependence based on the proposed definitions in polynomial time, but does not claim optimality. Hence, it may not possible to calculate the same information more efficiently.

7 Related Work

Fifteen years ago, control dependence was rigorously explored by Podgurski and Clarke in [18]. 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, 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 [14], Johnson proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges. In comparison, in this paper we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz, Reps, and Binkley [11] 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 [11] 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 [20] 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, [10], Horwitz et.al. use a semantics based multi-layered approach to reason about the correctness of slicing in the realm of data dependence. In [3], Ball et.al used 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 with out 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. Even though our definitions apply to irreducible graphs, we need to extra structure of reducible graphs to achieve the stronger correctness property. We are currently investigating if we can establish their correctness property using our control dependence definitions on irreducible graphs.

In [8], Hatcliff et.al. presented notions of dependence for concurrent CFGs, and proposed a notion of bi-simulation as the correctness property. Millett and Teitelbaum [12] study static slicing of Promela (the model description language for the model-checker SPIN) and its application to model checking, simulation, and protocol understanding, but they do not formalize a notion of correct slice nor do they discuss issues related to preserving non-termination and liveness properties. Krinke [15] considers static slicing of multi-threaded programs with shared variables, and focuses on issues associated with inter-thread data dependence but does not consider non-termination sensitive forms of control dependence.

8 Conclusion

The notion of control dependence is used in myriad of applications, and researchers and tool builders increasing seek to apply it to modern software systems and high-assurance applications – even though the control flow structure and semantic behavior of these systems does not mesh well with the requirements of existing control dependence dependences. 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 of static 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 a justify a comprehensive set of dependence notions for the rich features found in concurrent Java programs.

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. Anderson. *Program Analysis and Specialization for the C Programming Languages*. PhD thesis, DIKU, University of Copenhagen, DIKU, University of Copenhagen, Universit et sparken 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, ACM Press New York, NY, USA.

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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. Technical Report 8, Kansas State University, 2004. Available at http://projects.cis.ksu.edu/docman/admin/index.php?group_id=12.

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

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

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

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

A Details

A.1 Algorithm to calculate Non-Termination Sensitive Control Dependence

Proof of correctness We show that phase (1) and (2) of the algorithm are correct by proving that the following property holds at the end of phase (2).

$t_{n_1 n_2} \in S_{n_3 n_1}$ *if and only if each path* $\pi \in [n_2..n_1?]$ *contains* n_3.

```
NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE(G)
G(N, E, n0, N^E) :  a control flow graph.
S[|N|, |N|] :  a matrix of sets where S[n1, n2] represents S_{n1 n2}.
T[|N|] :  a sequence of integers where T[n1] denotes T_{n1}.
CD[|N|] :  a sequence of sets.
workbag :  a set of nodes.

 # (1) Initialize
workbag ← ∅
for  each n1 in condNodes(G)
do succs = succs(n1, G)
    for  each n2 in succs
    do workbag ← workbag ∪ {n2}
        S[n2, n1] ← {t_{n1 n2}}

 # (2) Calculate all-path reachability
while workbag ≠ ∅
do flag ← false
    n3 ← remove(workbag)
    for  each n1 in condNodes(G)\n3
    do if |S[n3, n1]| = T[n1]
          then for  each n4 in condNodes(G)\n3
                do if S[n1, n4]\S[n3, n4] ≠ ∅
                      then S[n3, n4] ← S[n3, n4] ∪ S[n1, n4]
                            flag = true

    if flag and |succs(n3, G)| = 1
      then n5 ← select(succs(n3, G))
            for n4 in condNodes(G)
            do if S[n5, n4]\S[n3, n4] ≠ ∅
                  then S[n5, n4] ← S[n5, n4] ∪ S[n3, n4]
                        workbag ← workbag ∪ {n5}
      else  if flag and |succs(n3, G)| > 1
              then for  each n4 in N
                    do if |S[n4, n3]| = T[n3]
                          then workbag ← workbag ∪ {n4}

 # (3) Calculate non-termination sensitive control dependence
for  each n3 in N
do for  each n1 in condNodes(G)
    do if |S[n4, n3]| > 0 and |S[n3, n1]| ≠ T[n1]
          then CD[n3] ← CD[n3] ∪ {n1}

return CD
```

Figure 3: The algorithm to calculate non-termination sensitive control dependence.

We shall use the *only if* direction as the loop invariant for the outer loops of phase (1) and (2).

At the beginning of phase (1), each token set $T_{n_3 n_1} = \emptyset$. Hence, the invariant is trivially established. In the loops at line 9 and 11, for each immediate successor node n_2 of each conditional node n_1, $t_{n_1 n_2}$ is injected into $T_{n_2 n_1}$. This trivially preserves the invariant at the end of the loop as $n_3 (= n_2)$ occurs on all segments starting n_2. The loop will terminate as the number of nodes in the graph is finite.

Now the reasoning about phase (2).

Initialization At the beginning of phase (2), the invariant is estalished as it is preserved at the termination of phase (1).

Maintenance $|S_{n_3 n_4}| = T_{n_4} > 0 \implies (\forall \pi \in [n_4..n_4?].(|\pi| > 1 \implies n_3 \in \pi))$. In other words, any path ending at n_4 can be extended to contain n_3. This is captured in line 23 and the invariant is established.

Termination Note that, even in the worst case, there can be $|N|^2$ tokens and $|N|^2$token sets. In each iteration, either the size of a token set increases at least by one or remains the same. Eventually the size of the token sets will stabilize (not increase) preventing additions of elements to the workbag at lines 31 and 35 (by not setting *flag* to *true* in the conditional at 22). Hence, the loop at line 16 will terminate while maintaining the invariant.

As for the *if* direction, the conditional at line 26 ensures that any change at a node n_3 is propagated to it's lone successor n_5 (lines 27-31) or to any node n_4 that occurs on all non-trivial paths $\pi \in [n_3..n_3?]$ (line 35 combined with subsequent execution of loop at line 19). In other words, the conditional captures the path extension mentioned in *maintenance*. This combined with the termination of the phases proves the *if* direction of the property.

In phase (3), direct control dependence is calculated based on the available reachability information. The termination of this phase is obvious by the finiteness of the nodes and edges of the graph.

Complexity analysis In phase (1), for every node with multiple successors in the CFG, each of its successors is processed. Hence, it leads to a worst-case asymptotic complexity of $O(|E|)$ for phase (1). In phase (3), for each node, every node in the CFG is processed leading to a worst-case asymptotic complexity of $O(|N|^2)$ for this phase.

In phase (2), the loop at line 16 iterates till the size of the token sets represented by S stabilizes. The maximum size of a token set $S[n_1, n_2]$ is given by $T[n_2]$ which is equal to the outdegree of n_2. In each iteration, either the size of a token set increases at least by one or remains the same. In the former case, it contributes an iteration. As the size of the token sets $S[n_1, n_2]$ is bound, all token sets of $S[n_1]$ will stabilize in a total of $\sum T[i]$ or less iterations. The loops in line 19 and 21 contribute $O(|\mathit{condNodes}(G)|^2) \approx O(|N|^2)$ to each such iteration. Hence, the worst-case complexity of phase (2) will be $O(|N|^3 \times \sum T[i] \times \lg(|N|))$ by factoring in the complexity $O(\lg |N|)$ of set operations.

By combining the above information, the worst-case complexity due to phase 1, 2, and 3 will be $O(|E| + |N|^3 \times \sum T[i] \times \lg |N| + |N|^2)$. However, as $O(|N|^3 \times \sum T[i] \times \lg |N|)$ dominates $O(|N|^2)$ and $O(|E|)$, the complexity will be $O(|N|^3 \times \sum T[i] \times \lg |N|$ when $\sum T[i] \times \lg |N| > 1$. It will be $O(|N|^2 + |E|))$ when $\sum T[i] = 0$.

As in practice $|\mathit{condNodes}(G)|^2 \approx |N|$, the complexity in the case where $\sum T[i] \times \lg |N| > 1$ will reduce to $O(|N|^2 \times \sum T[i] \times \lg |N|)$.

A.2 Algorithm to calculate Non-Termination Insensitive Control Dependence

Proof of correctness The control dependence between n_1 and n_2 as calculated by algorithm in Figure 3 is pruned if *notcd* is *true*. This happens only if a node $n_3 \neq n_1$ such that it belongs

```
NON-TERMINATION-INSENSITIVE-CONTROL-DEPENDENCE(G)
 1  G(N, E, n_0, N^E) :  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  K[|N|] :  a sequence of sets where K[n_1] is the set of nodes in the control sink of n_1.
 4  workbag :  a set of nodes.
 5
 6   # Calculate non-termination insensitive control dependence
 7  CD = NON-TERMINATION-SENSITIVE-INDIRECT-CONTROL-DEPENDENCE(G)
 8  for  each n_1 ∈ N
 9  do sinks ← GET-NODES-OF-SINKS-NOT-CONTAINING-NODE(n_1, G)
10      for  each n_2 ∈ CD[n_1]
11      do visited ← {n_1}
12          notcd ← true
13          workbag ← workbag ∪ {n_2}
14          while workbag ≠ ∅ and notcd
15          do n_3 ← remove(workbag)
16              visited ← visited ∪ {n_3}
17              if n_3 ∈ sinks
18                 then notcd ← false
19                 else  workbag ← workbag ∪ succs(n_3, G)\visited
20          if notcd
21             then CD[n_1] ← CD[n_1]\n_2
22
23  return CD
```

Figure 4: The algorithm to calculate non-termination insensitive control dependence.

to a control sink not containing n_1 is encountered on the graph exploration starting at n_2. The exploration visits each node only once by remembering the visited nodes in *visited*. Also, as *visited* contains n_1 at the beginning of the loop at line 15, immediate successors of n_1 will not be visited unless they are reachable by edges from other nodes reachable via nodes other than n_1. It is trivial to see that all nodes reachable from n_2 (not via outgoing edges of n_1) will be visited. Hence, the loop at line 15 will correctly decide if n_1 is non-termination insensitively control dependent on n_2. As the loop at line 15 is repeated for each control dependence of each node, the algorithm correctly calculates non-termination insensitively control dependence for the given CFG.

Complexity analysis The worst-case complexity of the call at line 8 is $O(|N|^3 \times \sum T[i] \times \lg |N| + |N|^3)$ by factoring in the cost to calculate indirect variant of non-termination sensitive control dependence from it's direct variant. It is possible to calculate the SCCs in a graph in $O(|N|+|E|)$ time, check if an SCC is control sink in $O(|E|)$ time, and check if a control sink contains a given node in $O(|N|)$. Hence, the call in line 10 contributes $O(|N| \times |E|)$. The loop at line 15 may explore each edge, hence, contributing a complexity of $O(|E|)$. The cumulative complexity of the loops at line 9 and 11 is $O(|N|)^2 \times |E|)$ as there can be $O(|N|^2)$ non-termination sensitive control dependences in the worst case. Hence, the worst-case complexity of the algorithm will be $O(|N|^3 \times \sum T[i] \times \lg |N| + |N|^2 \times |E|)$.

A.3 Algorithm to calculate Decisive Control Dependence

Proof of correctness We shall prove that the following property holds at the end of phase (2).

$t_{n_1 n_2} \in S_{n_3 n_1}$ if and only if there is a path from n_2 to n_3 not containing n_1.

We shall use the *only-if* direction as a loop invariant for the loop at lines 15-21.

Initialization After phase (1), for each successor n_2 of a conditional node n_1, $t_{n_1 n_2} \in S_{n_2 n_1}$. The invariant holds as there is an empty path from n_2 to n_2 that does not contain n_1.

```
Decisive-Control-Dependence(G)
 1  G(N, E, n0, N^E) :  a control flow graph.
 2  S[|N|, |N|] :  a matrix of sets where S[n1, n2] represents S_{n1 n2}.
 3  T[|N|] :  a sequence of integers where T[n1] denotes T_{n1}.
 4  workbag :  a set of nodes.
 5
 6   # (1) Initialize
 7  workbag ← ∅
 8  for  each n1 in condNodes(G)
 9  do succs ← succs(n1, G)
10      for  each n2 in succs
11      do workbag ← workbag ∪ {n2}
12          S[n2, n1] ← {t_{n1 n2}}
13
14   # (2) Calculate exists-a-path reachability
15  while workbag ≠ ∅
16  do n3 ← remove(workbag)
17      for  each n4 ∈ succs(n3, G)
18      do for  each n5 ∈ condNodes(G)\n3
19          do if S[n4, n5]\S[n3, n5] ≠ ∅
20                 then S[n4, n5] ← S[n4, n5] ∪ S[n3, n5]
21                      workbag ← workbag ∪ n4
22
23   # (3) Calculate decisive control dependence
24  CD ← Non-Termination-Sensitive-Control-Dependence(G)
25  for  each n1 ∈ N
26  do for  each n2 ∈ CD[n1]
27      do if |S[n1, n2]| = T[n2]
28             then CD[n1] ← CD[n1]\n2
29
30  return CD
```

Figure 5: The algorithm to calculate decisive control dependence.

Maintenance If there is a path from n_6, a successor of a conditional node n_5, to n_3 with no occurrence of n_5 then there is a similar path from n_6 to n_4, a successor of n_3. However, this is not true if $n_5 = n_3$. The logic in line 19 ensures that $t_{n_5 n_6} \in S_{n_3 n_5} \implies t_{n_5 n_6} \in S_{n_4 n_5}$ to maintain the invariant while line 18 avoids the case where $n_5 = n_3$.

Termination By an argument similar to that for algorithm in Figure 3, we can conclude that the loop at line 15 will terminate.

Upon termination, for each pair of node, n_3 and n_4, such that $n_3 \rightarrow n_4$ exists, for all nodes $n_5 (\neq n_3)$, $S_{n_3 n_5} \subseteq S_{n_4 n_5}$.

From the loop invariant we know that, if $t_{n_5 n_6} \in S_{n_3 n_5}$ then there is path from n_6, a successor of a conditional node n_5, to n_3 with no occurrences of n_5. If n_4 is a successor of n_3 then there is a path from n_6 to n_4 as well. Also, $S_{n_3 n_5} \subseteq S_{n_4 n_5} \implies t_{n_5 n_6} \in S_{n_4 n_5}$. Hence, the *if* direction of the property holds at the end of phase (2).

As for the correctness of phase (3), it is trivial to see that $S_{n_1 n_2} = T[n_2]$ implies that there is a path from each successor of n_2 that contains n_1 before any occurrence of n_2. Likewise, $S_{n_1 n_2} \neq T[n_2]$ implies that there is a successor of n_2 such that all paths from it does not contain n_1 before any occurrence of n_2. Hence, phase (3) correctly calculates decisive control dependence.

Complexity analysis In phase (1) each successor of each conditional node is processed. Hence, the worst-case complexity for phase (1) will be $O(|E|)$.

The complexity of phase (3) is $O(|N|^3 \times \sum T[i] \times \lg |N|)$. For phase (2), the complexity analysis would be the same as that for phase (3).

In phase (4), each control dependence is explored. In the worst case, there can be $|N|^2$ control dependences in a CFG. Hence, the worst-case complexity of phase (3) is $O(|N|^2)$.

Hence, the worst-case complexity of the algorithm will be $O(|E| + |N|^3 \times \sum T[i] \times \lg |N| + |N|^2)$. Special cases of worst-case complexity of algorithm in Figure 3 can be applied here as well.