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

Venkatesh Prasad Ranganath[1], Torben Amtoft[1], Anindya Banerjee[1], Matthew B. Dwyer[2], and John Hatcliff[1]

[1] Department of Computing and Information Sciences, Kansas State University[**]
{rvprasad, tamtoft, ab, hatcliff}@cis.ksu.edu
[2] Department of Computer Science and Engineering, University of Nebraska, Lincoln[***]
dwyer@cse.unl.edu

Abstract. The notion of control dependence underlies many program analysis and transformation techniques. Despite wide applications, existing definitions and approaches for calculating control dependence are difficult to apply seamlessly to modern program structures. Such program 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 slicing. It develops definitions and algorithms for computing control dependence that can be directly applied to modern program structures. A variety of properties show that the new definitions conservatively extend classic definitions. 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. The 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 [2], debugging [3], partial evaluation [4], compiler optimizations [5] 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 for calculating control dependence and slicing are widely applied and have been used in the current form for well over 20

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

[**] Manhattan KS, 66506, USA.

[***] Lincoln NE 68588-0115, USA.

years, there are several aspects of 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 [6, 2]. 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.) A deeper problem is that 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 [7] 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 4.1).
– 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 [8]. The proposed notions of control dependence described in this paper have been implemented in Indus [9] – 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 Bandera, a tool set for model-checking concurrent Java systems.[1]

2 Basic Definitions

2.1 Control Flow Graphs

In the sequel, we follow tradition and represent a program as a control-flow graph, whose definition we adapt from Ball and Horwitz [10].

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* A*lways 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 above definition *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 [11]. 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. 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. 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 [2]. 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.

A CFG G of the form (N, E, n_0) is *reducible* if E can be partitioned into disjoint sets E_f (the *forward* edge set) and E_b (the *back* edge set) such that (N, E_f) forms a DAG in which each node can be reached from the entry node n_0 and for all edges $e \in E_b$, the target of e dominates the source of e. All "well-structured" programs, including Java programs, give rise to reducible control-flow graphs. Our definitions and most of our correctness results apply to irreducible CFGs as well, but our correctness result of slicing based on bisimulation holds for reducible graphs since bisimulation requires ordering properties that can only be guaranteed on reducible graphs – (see example in Section 4 preceding Theorem 2.)

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. For state (n_i,σ_i), the code at n_i is executed on the transition from (n_i,σ_i) to successor state (n_{i+1},σ_{i+1}). Execution begins at the start node n_0, and the execution of each node possibly updates the store and transfers control to an appropriate successor node. Execution of a node $n_e \in N^E$ produces a final state (halt,σ) where the control point is indicated by a special label halt – this indicates a normal termination of program execution. The presentation of slicing in Section 5 involves arbitrary finite and infinite 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 referenced at some program points of interest; such program points are traditionally called the *slicing criterion* [12]. A slicing criterion C for a program p is a non-empty set of nodes $\{n_1,\ldots,n_k\}$ where each n_i is a node in p's CFG.

The definitions below are the classic ones of the two basic notions of dependence that appear in slicing of sequential programs: *data dependence* and *control dependence* [12].

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

Definition 2 (data dependence). *Node n is* data-dependent *on m (written $m \overset{dd}{\rightarrow} n$ – the arrow pointing in the direction of data flow) if there is a variable v such that: (1) there exists a non-trivial path 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 [11]).

We write $m \xrightarrow{d} n$ when either $m \xrightarrow{dd} n$ or $m \xrightarrow{cd} n$. The algorithm for constructing a program slice proceeds by finding the set of CFG nodes S_C (called the *slice set* or *backward static slice*) from which the nodes in C are reachable via $\xrightarrow{d}$. The term "backward" signifies that 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. Our definitions of control dependence can be applied in computing forward slices as well.

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:*

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

We will consider slicing correctness requirements in greater detail in Section 5.1. For now we note that commonly in the slicing literature the desired correspondence between the source program and the slice is not formalized; 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* [13]. 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.

3 Assessment of Existing Definitions

3.1 Variations in Existing Control Dependence Definitions

Although Definition 3 of control dependence is widely used, there are a number of (often subtle) variations appearing in the literature. Here are some:

Admissibility of indirect control dependences. For example, using the definition of control dependence in Definition 3, for Fig. 1 (a), we can conclude that $a \xrightarrow{cd} f$ and $f \xrightarrow{cd} g$ however $a \xrightarrow{cd} g$ 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 [2] incorporate this notion of transitivity directly into the definition itself as we will illustrate later.

Sensitivity to non-termination. Consider Fig. 1 (a) again, where node c represents a post-test that controls a potentially infinite loop. According to Definition 3, $a \xrightarrow{cd} d$ holds but $c \xrightarrow{cd} d$ 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 the variations by recalling definitions of strong and weak control dependence given by Podgurski and Clarke [2] and used in a number of works, e.g., the study of control dependence by Bilardi and Pingali [14].

Definition 5 (Podgurski-Clarke Control Dependence).

- *n_2 is strongly control dependent on n_1 ($n_1 \overset{PC-scd}{\rightarrow} n_2$) 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 ($n_1 \overset{PC-wcd}{\rightarrow} n_2$) 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.*

Whereas Definition 3 captures direct control dependence only, strong control dependence as defined above captures indirect control dependence. For example, in Fig. 1, in contrast to Definition 3, we have $a \overset{PC-scd}{\rightarrow} g$ because there is a path *afg* which does not contain e, 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.

Weak control dependence subsumes the notion of strong control dependence ($n_1 \overset{PC-scd}{\rightarrow} n_2$ implies $n_1 \overset{PC-wcd}{\rightarrow^*} n_2$) and it captures weaker dependences between nodes induced by non-termination: it is non-termination sensitive. For Fig. 1 (a), $c \overset{PC-wcd}{\rightarrow} d$ because *d* does not strongly post-dominate *b*: the presence of the loop controlled by *c* guarantees that there exists no k such that every path from node *b* of length $\geq k$ passes through *d*.

The impact of the variations on slicing. 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 Fig. 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 seems to be largely a 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, weak control dependence is actually a larger 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 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 [7].

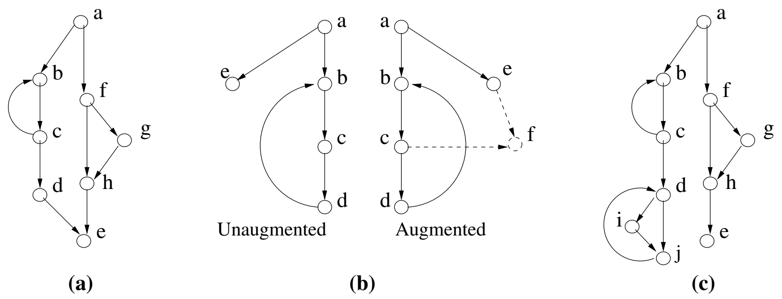

Fig. 1. (a) is a simple CFG. (b) illustrates how a CFG that does not have a unique exit node reachable from all nodes can be augmented to have unique exit node reachable from all nodes. (c) is a CFG with multiple control sinks of different sorts

3.2 Unique End Node Restriction on CFGs

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. If the augmentation is not reversed, the graph algorithms and analyses algorithms should be made intelligent to operate on the actual CFG embedded in the augmented CFG.

Many systems have threads where the main control loop has no exit – the loop is "exited" by simply killing the thread. For example, in 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. But this can disrupt the control dependence calculations. In Fig. 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*. But now 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, the control dependence of n_j 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 **($n_i \overset{ntscd}{\rightarrow} n_j$).** *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, and (1) for all maximal paths from n_k, n_j always occurs and, either $n_j = n_i$, or n_j occurs before any occurrence of n_i; and, (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) [15]. 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.

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

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

Here, $(G, n_i) \models$ expresses the fact that the CTL formula is checked against the graph G at node n_i. The two conjuncts are essentially a direct transliteration of the two conditions in Definition 6.

We have formulated Definition 6 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*. A CTL formula for this simplified definition is

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

See [8] for the proof that the simplified definition and Definition 6 are equivalent on CFGs.

Illustrating non-termination sensitivity of Definition 6: Note that $c \overset{ntscd}{\rightarrow} d$ in Fig. 1 (a) since there exists a maximal path (an infinite loop between b and c) where d never occurs. In Fig. 1 (c), note that $d \overset{ntscd}{\rightarrow} i$ 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 Fig. 1 (b unaugmented), and $\{e\}$ and $\{d, i, j\}$ are control sinks in Fig. 1 (c). Let the set of *sink-bounded paths from n_k* (denoted $SinkPaths(n_k)$) contain all paths π 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 two successors, n_k and n_l, and (1) for all paths $\pi \in SinkPaths(n_k)$, $n_j \in \pi$; and, (2) there exists a path $\pi \in SinkPaths(n_l)$ such that $n_j \notin \pi$ and if π leads to a control sink κ, $n_j \notin \kappa$.*

In CTL:

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

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; *c-sink* returns the sink set associated with the current node.

Illustrating non-termination insensitivity of Definition 7: Note that $c \overset{nticd}{\not\rightarrow} d$ in Fig. 1 (a) since all paths from c to the control sink, $\{e\}$, contain d. In Fig. 1 (b unaugmented) $a \overset{nticd}{\rightarrow} e$ because there exists 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. Interestingly, for Fig. 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 the definition inherently captures a form of fairness – since the back edge 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. Consequently, 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.

4.1 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 $n_i, n_j \in N$ we have: (1) $n_i \overset{cd}{\rightarrow} n_j$ 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 $n_i \overset{ntscd^*}{\rightarrow} n_j$.*

Part(4) of the above theorem is illustrated as follows: in Fig. 1 (a), $a \overset{nticd}{\rightarrow} d$ holds but $a \overset{ntscd}{\rightarrow} d$ does not. But $a \overset{ntscd^*}{\rightarrow} d$ holds as both $a \overset{ntscd}{\rightarrow} c$ and $c \overset{ntscd}{\rightarrow} d$ hold.

For the (bisimulation-based) correctness proof in Section 5.1, we shall need a few results about slice sets (members of which are termed "observable"). The main intuition is that the nodes in a slicing criteria C represent "observations" that one is making about a CFG G under consideration. Specifically, for an $n \in C$, one can observe that n has been executed and also observe the values of any variables referenced at n. A crucial property is that the first observable node on any path (n_1 in the lemmas below) will be encountered sooner or later on all other paths. Letting Ξ be the set of nodes, we have:

Lemma 1. *Assume Ξ is closed under $\overset{ntscd}{\rightarrow}$, 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 maximal paths from n_0 will contain n_1.*

The notion of "closed" Ξ is this: if $n_i \in \Xi$ and $n_i \overset{ntscd}{\rightarrow} n_j$ then $n_j \in \Xi$.

Lemma 2. *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 3. *Assume that $n_0 \notin \Xi$, but that there is a path π starting at n_0 which contains a node in Ξ. (1) If Ξ is closed under $\overset{nticd}{\rightarrow}$, then all sink bounded paths starting at n_0 will reach Ξ. (2) If Ξ is also closed under $\overset{ntscd}{\rightarrow}$, then all maximal paths starting at n_0 will reach Ξ.*

We are now ready for the section's main result: from a given node there is a unique first observable. For this, we need the CFG to be reducible, as can be seen by the counterexample where from n_0 there are edges to n_1 and n_2 between which there is a cycle.

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 $\overset{ntscd}{\rightarrow}$ and if the CFG is reducible, then $n_1 = n_2$.*

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 $\overset{dd}{\rightarrow}$ and also closed under $\overset{ntscd}{\rightarrow}$.

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 $n \notin S_C$:

- 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

For a slicing criterion C, 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 8. *For $i = 1,2$, wrt. code map $code_i$: $s \overset{i}{\longmapsto} s'$ denotes that program state s rewrites in one step to s'. And, $s_0 \overset{i}{\Longrightarrow} s$ denotes that there exists $s_1 \dots s_k$ ($k \geq 1$) with $s_k = s$ such that*
(1) for all $j \in \{1 \dots k\}$ we have $s_{j-1} \overset{i}{\longmapsto} s_j$;
(2) $n_k \in S_C$ but for all $j \in \{1 \dots k-1\}$, $n_j \notin S_C$, where $s_j = (n_j, \sigma_j)$ for each j.

Definition 9. *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 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 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 have now arrived at the correctness theorem:

Theorem 3. *Let 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 (given reducible G) if S_C is closed under $\overset{ntscd}{\rightarrow}$ then $\mathcal{S}_0$ is a bisimulation.*

6 Non-termination Sensitive Control Dependence Algorithm

Control dependences are calculated using a symbolic data-flow analysis. Each outgoing edge $n \rightarrow p$ of a predicate node n is represented by a token t_{np}. At each node m, a summary set S_{mn} is maintained for each predicate node n. Tokens are injected into the summary sets of the successors of each predicate node. The tokens are then propagated according to the following rules until no propagatian can occur.

- If q is a non-predicate node in $q \rightarrow r$ then the tokens in the summary sets at q are copied into the corresponding summary sets at r. This records that all maximal paths containing q also contain r.
- Only if all tokens corresponding to a predicate node n have arrived at node q then the tokens in the summary sets at n are copied into corresponding summary sets at q. This records that all maximal paths containing n also contain q.

Upon termination, $t_{np} \in S_{mn}$ indicates that all maximal paths from n starting with $n \rightarrow p$ contain m. Based on this observation, if $|S_{mn}| > 0 \wedge |S_{mn}| \neq T_n$ then, by Definition 6, it can be inferred that m *is directly control dependent on* n. On the other hand, if $|S_{mn}| > 0$ and $|S_{mn}| = T_n$ then, by Definition 6, it can be inferred that m *is not directly control dependent on* n.

The above algorithm has a worst-case asymptotic complexity of $O(|N|^3 \times K)$ where K is the sum of the outdegree of all predicate nodes in the CFG. Linear time algorithms to calculate control dependence based on augmented CFGs have been proposed in the literature [2]. The practical cost of this augmentation varies with the specific algorithm and the nature of control dependence being calculated. Our experience with an implementation of our algorithm in a program slicer for full Java [17] suggests that, despite its complexity bound, it elegantly scales to programs with tens-of-thousands of lines of code. We suspect that this is due in part to the elimination of the processing overhead involved in dealing with augmented CFGs.

```
NON-TERMINATION-SENSITIVE-CONTROL-DEPENDENCE($G$)
$G(N, E, n_0, N^E)$ : a control flow graph.
$S[|N|, |N|]$ : a matrix of sets where $S[n_1, n_2]$ represents $S_{n_1 n_2}$.
$T[|N|]$ : a sequence of integers where $T[n_1]$ denotes $T_{n_1}$.
$CD[|N|]$ : a sequence of sets.
$workbag$ : a set of nodes.

# (1) Initialize
$workbag \leftarrow \emptyset$
for each $n_1$ in $condNodes(G)$ and $n_2$ in $succs(n_1, G)$
do $workbag \leftarrow workbag \cup \{n_2\}$
   $S_{n_2 n_1} \leftarrow \{t_{n_1 n_2}\}$

# (2) Calculate all-path reachability
while $workbag \neq \emptyset$
do $flag \leftarrow false$
   $n_3 \leftarrow remove(workbag)$
   for each $n_1$ in $condNodes(G) \setminus n_3$
   do if $|S_{n_3 n_1}| = T_{n_1}$
         then for each $n_4$ in $condNodes(G) \setminus n_3$
              do if $S_{n_1 n_4} \setminus S_{n_3 n_4} \neq \emptyset$
                    then $S_{n_3 n_4} \leftarrow S_{n_3 n_4} \cup S_{n_1 n_4}$
                         $flag = true$

   if $flag$ and $|succs(n_3, G)| = 1$
     then $n_5 \leftarrow$ the successor of $n_3$ in $G$
          for $n_4$ in $condNodes(G)$
          do if $S_{n_5 n_4} \setminus S_{n_3 n_4} \neq \emptyset$
                then $S_{n_5 n_4} \leftarrow S_{n_5 n_4} \cup S_{n_3 n_4}$
                     $workbag \leftarrow workbag \cup \{n_5\}$
     else if $flag$ and $|succs(n_3, G)| > 1$
            then for each $n_4$ in $N$
                 do if $|S_{n_4 n_3}| = T_{n_3}$
                       then $workbag \leftarrow workbag \cup \{n_4\}$

# (3) Calculate non-termination sensitive control dependence
for each $n_3$ in $N$
do for each $n_1$ in $condNodes(G)$
   do if $|S_{n_4 n_3}| > 0$ and $|S_{n_3 n_1}| \neq T_{n_1}$
         then $CD[n_3] \leftarrow CD[n_3] \cup \{n_1\}$

return $CD$
```

Fig. 2. The algorithm to calculate non-termination sensitive control dependence

A complete description of the algorithm, its correctness, and its complexity analysis is given in [8].

7 Related Work

Fifteen years ago, control dependence was rigorously explored by Podgurski and Clarke [2]. 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 [14] proposed new concepts related to control dependence along with algorithms based on these concepts to efficiently calculate weak control dependence. Johnson proposed an algorithm that could be used to calculate control dependence in time linear in the number of edges [18]. In comparison, in this paper we sketch a feasible algorithm in a more general setting.

In the context of slicing, Horwitz, Reps, and Binkley [19] presented what has now become the standard approach to inter-procedural slicing via dependence graphs. Recently, Allen and Horwitz [20] extended previous work on slicing to handle exception-based inter-procedural control flow. In this work, they handle CFG's with two end nodes (one for normal return and one for exceptional return) but it is unclear how this affects the control dependence captured by the dependence graph. In comparison, we have shown that program slicing is feasible with unaugmented CFGs.

For relevant work on slicing correctness, Horwitz et.al. use a semantics based multi-layered approach to reason about the correctness of slicing in the realm of data dependence [21]. Ball et.al use a program point specific history based approach to prove the correctness of slicing for arbitrary control flow [10]. We extend that work to consider arbitrary control flow without the unique end-node restriction. Their correctness property is a weaker property than bisimulation – 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 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.

Hatcliff et.al. present notions of dependence for concurrent CFGs, and propose a notion of bisimulation as the correctness property [6]. Millett and Teitelbaum [22] study static slicing of Promela (the model description language for the model-checker SPIN) and its application to model checking, simulation, and protocol understanding. They reuse existing notions of slicing, however, they neither discuss issues related to preservation of non-termination and liveness properties nor formalize a notion of correct slice for their applications. Krinke [23] 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 control dependence.

8 Conclusion

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

We have rigorously justified these definitions by providing detailed proofs of correctness (see the companion technical report [8]), 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 statically and dynamically calculating dependences for concurrent Java programs for slicing, program verification, and security applications. In particular, we are exploring the relationship between dependences extracted from execution traces and dependences extracted from control-flow graphs in an effort to systematically justify a comprehensive set of dependence notions for the rich features found in concurrent Java programs. This effort will yield a more direct *semantic* connection between notions of dependence and execution traces instead of working indirectly through syntactic-oriented CFG definitions. With the translated, temporal logic-based dependence definitions, we are investigating how certain temporal properties of the unsliced version of the program are preserved in the sliced version.

References

1. Corbett, J., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting Finite-state Models from Java source code. In: 22^{nd} International Conference on Software Engineering (ICSE'00). (2000) 439–448.
2. Podgurski, A., Clarke, L.: A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Soft. Engg. **16** (1990) 965–979.
3. Francel, M.A., Rugaber, S.: The relationship of slicing and debugging to program understanding. In: Seventh IEEE International Workshop on Program Comprehension (IWPC'99). (1999) 106–113.
4. Anderson, L.O.: Program Analysis and Specialization for the C Programming Languages. PhD thesis, DIKU, University of Copenhagen (1999).
5. Ferrante, J., Ottenstein, K.J., Warren, J.O.: The program dependence graph and its use in optimization. ACM TOPLAS **9** (1987) 319–349.
6. Hatcliff, J., Corbett, J.C., Dwyer, M.B., Sokolowski, S., Zheng, H.: A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In: International Static Analysis Symposium (SAS'99) (1999), 1–18.
7. Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing software for model construction. Journal of Higher-order and Symbolic Computation **13** (2000) 315–353.

8. Ranganath, V.P., Amtoft, T., Banerjee, A., Dwyer, M.B., Hatcliff, J.: A new foundation for control-dependence and slicing for modern program structures. Technical Report 8, SAnToS Lab., Kansas State University (2004). Available at http://projects.cis.ksu.edu/docman/admin/index.php?editdoc=1&docid=95&group_id=12.
9. SAnToS Laboratory, Kansas State University: Indus, a toolkit to customize and adapt Java programs. Available at http://indus.projects.cis.ksu.edu.
10. Ball, T., Horwitz, S.: Slicing programs with arbitrary control-flow. In: First International Workshop on Automated and Algorithmic Debugging (AADEBUG). Volume 749 of Lecture Notes in Computer Science, Springer-Verlag (1993) 206–222.
11. Muchnick, S.S.: Advanced Compiler Design & Implementation. Morgan Kaufmann Publishers. Inc., San Francisco, California, USA (1997).
12. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages **3** (1995) 121–189.
13. Weiser, M.: Program slicing. IEEE Trans. Soft. Engg. **10** (1984) 352–357.
14. Bilardi, G., Pingali, K.: A framework for generalized control dependences. In: PLDI'96, 1996, 291–300.
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999).
16. Milner, R.: Communication and Concurrency. Prentice Hall (1989).
17. Jayaraman, G., Ranganath, V.P., Hatcliff, J.: Kaveri: Delivering Indus Java Program Slicer to Eclipse. In: Fundamental Approaches to Software Engineering (FASE'05), 2005. To appear.
18. Johnson, R., Pingali, K.: Dependence-based program analysis. In: PLDI'93, 1993, 78–89.
19. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. ACM TOPLAS, 1990, 35–46.
20. Allen, M., Horwitz, S.: Slicing Java programs that throw and catch exceptions. In: PEPM'03, 2003, 44–54.
21. Horwitz, S., Pfeiffer, P., Reps, T.W.: Dependence analysis for pointer variables. In: PLDI'89, 1989, 28–40.
22. Millett, L., Teitelbaum, T.: Slicing Promela and its applications to model checking, simulation, and protocol understanding. In: Fourth International SPIN Workshop. (1998)
23. Krinke, J.: Static slicing of threaded programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE'98). (1998) 35–42.