SAS'06, LNCS 4134, pp 387-404. 2006.

Parametric and Termination-Sensitive Control Dependence - Extended Abstract

Feng Chen and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana - Champaign, USA
{fengchen,grosu}@uiuc.edu

Abstract. A parametric approach to control dependence is presented, where the parameter is any prefix-invariant property on paths in the control-flow graph (CFG). Existing control dependencies, both direct and indirect, can be obtained as instances of the parametric framework for particular properties on paths. A novel control dependence relation, called termination-sensitive control dependence, is obtained also as an instance of the parametric framework. This control dependence is sensitive to the termination information of loops, which can be given via annotations. If all loops are annotated as terminating then it becomes the classic control dependence, while if all loops are annotated as non-terminating then it becomes the weak control dependence; since in practice some loops are terminating and others are not, termination-sensitive control dependence is expected to improve the precision of analysis tools using it. The unifying formal framework for direct and indirect control dependence suggests also, in a natural way, a unifying terminology for the various notions of control dependence, which is also proposed in this paper. Finally, a worst-case $O(n^2)$ algorithm to compute the indirect termination-sensitive control dependence for languages that allow only "structured" jumps (i.e., ones that do not jump into the middle of a different block), such as Java and *C#*, is given, avoiding the $O(n^3)$ complexity of the trivial algorithm calculating the transitive closure of the direct dependence.

1 Introduction

Control dependence plays a fundamental role in program analysis: in program slicing [13, 18], in compiler optimization [12, 1], in total program correctness [15], and in security (of information flows) [11]. Intuitively, a statement S control-depends on a choice statement C iff the choice made at C determines whether S is executed or not. Because of the significance and broad range of applications of control dependence, related definitions and algorithms have been extensively investigated: [12] gives an efficient algorithm to compute (direct) control dependence; [15] introduces strong control dependence (also called the range of the control statement in [19]) as well as weak control dependence; [4] defines a generalized control dependence to capture both classic and weak direct control dependencies, together with their corresponding algorithms.

Although all these notions of control dependence are related, there is no adequate unifying framework for all of them, not even a uniform or consistent terminology. This often results in confusion and difficulty in understanding existing work, and may slow future developments, in particular defining new, or domain-specific control dependence relations. For example, the strong control dependence in [15] is the transitive closure of the control dependence in [12], contradicting common practice in formal terminology, since the former is actually weaker than the latter as a binary relation; the generalized control dependence in [4] addresses only the *direct* control dependencies (classic and weak), but omits the word "direct" in definitions and proofs, and also proposes the terminology "loop control dependence" for (direct) weak control dependence; [15] claims that strong control dependence is included in weak control dependence, which appears quite intuitive, but it is non-trivial to prove rigorously. A rigorous development of a unifying framework for the various control dependences, like the one proposed in this paper, would enhance understanding and clarify terminology in this area.

A first important step in this direction is made by [4], which defines a generalized control dependence that is parametrized by a property on paths and captures both classic and weak direct control dependences. A linear time algorithm [4] detects all statements that directly depend on a choice statement. However, the parametric approach in [4] covers only *direct* control dependence. The first contribution of our work, *parametric control dependence* (Section 3), consists of an extension of the work in [4] that also includes *indirect* control dependencies, as well as *comparisons* of different concrete instances of it. Our compact prefix-invariance property of the parameter is equivalent to the intersection of all the constraints on the parameter required by the results in [4], modulo the fact that we do not add a self-looping edge to the terminal node of the CFG to capture weak control dependence; in fact, we need to apply no transformations on CFGs in order to capture particular control dependencies as special cases. We also develop a rigorous mathematical theory in Section 3, capturing formally many results about different control dependence relations (Corollaries 1, 2 and 3).

The second contribution of this paper consists of defining a new control dependence relation that we call *termination-sensitive control dependence*, because it is sensitive to the termination information of loops, which can be given as annotations. If all loops are annotated as terminating then the termination-sensitive control dependence becomes the classic control dependence, while if all loops are annotated as non-terminating then it becomes the weak control dependence. If some loops are annotated as terminating while others not, then the termination-sensitive dependence strictly includes the classic control dependence and is strictly included in the weak one. Thus, one can regard it as a "knob" allowing one to tune the precision anywhere in between the two most widely accepted, but rather extreme control dependence relations. Since in practice some loops are terminating and others are not, termination-sensitive control dependence is expected to improve the precision of analysis tools using it. We introduce this termination-sensitive control dependence and derive all its properties as a formal instance of the parametric control dependence in the first part of the paper; it is in fact this new control dependence together with the lack of foundational and algorithmic support for *indirect* variants of control dependence of the generic control dependence in [4] that motivated our parametric approach to control dependence presented in Section 3.

The third contribution of our paper, Section 5, consists of an $O(n^2)$ algorithm to compute all *control scopes* for all the (branch) statements in a program of size n, in the context of higher level programming languages, such as Java and C#; statement S is in the control scope of C if and only if S termination-sensitive *indirectly* control-depends on C (control scope will be defined in Section 5). Since our control scopes become precisely the transitive closures of the classic and weak *direct* control dependencies when the loops are all annotated as terminating and as non-terminating, respectively, this generic algorithm seamlessly yields special instance algorithms to calculate the *indirect* versions of these dependencies, namely the complete strong and weak control dependencies, in $O(n^2)$ complexity. These results appear to be new even in the widely accepted, but in our view restricted, framework of strong and weak control dependence.

Section 2 revisits control dependence notions and presents them in a uniform light, as instances of the forthcoming parametric control dependence. Section 3 presents our parametric version of control dependence; a result relating the control dependence relations associated to different path properties allows us to compare the various instances of control dependence, in particular to show that the termination-sensitive (indirect) control dependence, discussed in Section 4, includes the standard control dependence but is included by the weak control dependence. Section 5 discusses the $O(|V|^2)$ algorithm to compute the entire termination-sensitive indirect control dependence. Due to space limitations, the interested reader is referred to [8] for detailed proofs.

Motivation. Even though *direct* variants of control dependence tend to suffice in program slicing efforts, there are many applications that need *indirect* control dependence. For example, in [19], the (indirect) control dependence is used to define and reason about information flow in security, and in [15], (indirect) weak control dependence is used to prove total correctness of programs. A less standard application domain is that of runtime analysis or multithreaded systems, described in more detail below.

Our main motivation for the termination-sensitive control dependence came from efforts in debugging multithreaded systems, namely in improving the accuracy and the coverage of predictive runtime analysis [7]. Since we refer back to it later in the paper, we explain this runtime analysis on a very simple example. Assume the running threads and events in Figure 1, where e_1 causally precedes, or "happens-before", e_2 (e.g., e_1 writes a shared variable and e_2 reads it right afterwards), and the statement generating e'_3 is in the control scope of the statement generating e_2, while the statement generating e_3 is not in the control scope of e_2. Then we say that e'_3 *is dependent upon* e_1, but that e_3 is *not* dependent upon e_1, despite the fact that e_1 obviously happened before e_3.

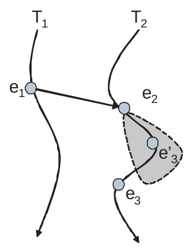


Fig. 1. Predictive Analysis

The intuition here is that e_3 would happen anyway, with or without e_1 happening. Because of its combined static/dynamic flavor, we call this new dependence relation on events the *hybrid dependence*. Interestingly, if the events in the scope of e_2 are not relevant for the property to check, then any permutation/linearization of relevant events consistent with the intra-thread total order and the hybrid dependence corresponds to some valid execution of the multithreaded system. Therefore, if any of these permutations violate the property, then the system can do so in a different execution. In particular, without any other dependencies but those in Figure 1, the property "e_1 must happen before e_3" can be violated by the program generating this execution, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e_1 should precede e_3, since e_3 happens anyway.

The control scope of a statement is determined statically, as the set of statements that control depend on it. Unfortunately, classic control dependence does not consider non-terminating loops, thus leading to false positives in the runtime analysis, while weak control dependence makes the worst case assumption (all loops are not terminating), resulting in over-restrictive dependence among events and thus false negatives. Termination-sensitive control dependence takes the termination information of loops into account in order to build a more precise control dependence relation.

2 Control Dependence Revisited

Here we discuss some of the major known results on control dependence, introducing at the same time a uniform notation and terminology. Some of the results in this section are mentioned in other works as "folklore"; however, we were not able to find them proved formally in the literature. We will show that all these results follow as corollaries of the general results in the next section. The structure of the results in this section anticipates the structure of those for parametric control dependence in the next section.

Preliminaries. A *directed graph* G is a pair $\langle V, E\rangle$, where $E \subseteq V \times V$. The elements of V are called *nodes* and those of E are called *edges*. A *finite path* of G is a finite sequence of nodes $u_1u_2...u_{m+1}$ such that $(u_i, u_{i+1}) \in E$ for all $0 < i \leq m$, where $m > 0$ is its *length*. If $u = u_1$ and $v = u_{m+1}$ then we call this path a *$u - v$ path*. For any node u, we let λ_u be the empty path from u to itself; its length is 0. An *infinite path* is an infinite sequence $u_1u_2...$ such that $(u_i, u_{i+1}) \in E$ for all $i > 0$. A *u–path* is a (finite or infinite) path starting with u. We let $Paths(G)$ be the set of all paths of G, finite or infinite. For a path π either infinite or finite in length greater than or equal to $k \geq 0$, we let $\pi|_k$ be the path containing the first k edges of π, i.e., $u_1u_2...u_{k+1}$. We also define the concatenation of paths: if $\alpha = u_1u_2...u_m$ finite and $\pi = u_mu_{m+1}...$ finite or infinite, then $\alpha\pi$ is the finite or infinite path $u_1u_2...u_mu_{m+1}....$ A *property* of paths in a graph G is a set $\mathcal{P} \subseteq Paths(G)$. For any $\pi \in Paths(G)$, we say that $\mathcal{P}(\pi)$ holds, or simply $\mathcal{P}(\pi)$, iff $\pi \in \mathcal{P}$.

Definition 1. *[12] A* ***control flow graph*** *$CFG = \langle V, E, START, END\rangle$ is a directed graph $\langle V, E\rangle$ together with an* ***entry node****, $START$, from which every other node is reachable, and an* ***exit node****, END, reachable from any other node. We make the standard assumption that nodes in V except END can have either one or two successors. Let $V_C \subseteq V$ denote the set of nodes with two successors and call them* ***choice nodes****.*

Nodes in V correspond to statements in the program, edges in E indicate possible flows of control in the program, and choice nodes correspond to choice statements, such as conditionals, e.g., C_1 in Figure 2 (A). Conditionals can also be parts of loops, e.g., C_1 and C_2 in Figure 2 (C). Due to the assumption on the number of successors, $|E| = O(|V|)$. In this paper, we tend to use letters at the beginning of the Greek alphabet, such as α, β, γ, etc., for $u - v$ paths, and letters π, π' and so on, for infinite or $u - END$ paths, though this convention is not strictly obeyed. From here on we fix a CFG.

2.1 Classic Control Dependence

Definition 2. *([12, 11]) Node u is* ***post-dominated*** *by node v, written $u \diamond\!\!\rightarrow v$, iff all $u - END$ paths contain v. Let* ***PostDom****(u) be the set of post-dominators of u except u.*

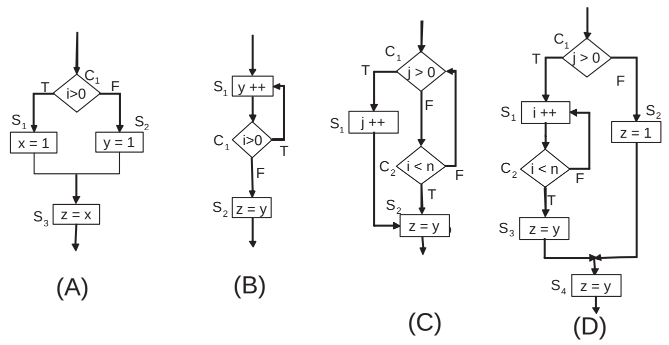


Fig. 2. Some control flow graphs

The notation $u \mathrel{\diamond\!\!\rightarrow} v$ symbolizes that no matter how we leave u (the first two edges of the diamond), we eventually converge (the other two edges of the diamond) and reach (the arrow) v. In Figure 2 (A), $C_1 \mathrel{\diamond\!\!\rightarrow} S_3$, while S_1 and S_2 do *not* post-dominate C_1; in Figure 2 (B), $C_1 \mathrel{\diamond\!\!\rightarrow} S_2$, while S_1 is *not* a post-dominator of C_1 – however, there is no guarantee that S_2 will be reached once C_1 is reached, because of the potentially infinite loop through C_1. In our context of predictive runtime analysis, this reflects a serious limitation of the classic notion of control dependence; we will discuss this issue shortly.

Lemma 1. *The post-dominance relation, $\mathrel{\diamond\!\!\rightarrow}$, is a partial order on the nodes of CFG.*

The following properties of post-dominance are immediate corollaries of our parametric control dependence framework in Section 3:

Corollary 1. *The following hold: (1) If $v_1 \neq v_2 \in PostDom(u)$ then either $v_1 \mathrel{\diamond\!\!\rightarrow} v_2$ or $v_2 \mathrel{\diamond\!\!\rightarrow} v_1$, i.e., $\langle PostDom(u), \mathrel{\diamond\!\!\rightarrow}\rangle$ is a total order; and (2) For any u, if $PostDom(u) \neq \emptyset$ then $PostDom(u)$ has a unique first element w.r.t. $\mathrel{\diamond\!\!\rightarrow}$.*

Definition 3. *Let $ipd(u)$ be the first element of $\langle PostDom(u), \mathrel{\diamond\!\!\rightarrow}\rangle$, called the* ***immediate post-dominator*** *of u; let $u \mathrel{\dot{\diamond}\!\!\rightarrow} v$ iff $v = ipd(u)$.*

The immediate post-dominator is the post-dominator that appears first on *any* $u - END$ path. For example, in Figure 2 (A), $C_1 \mathrel{\dot{\diamond}\!\!\rightarrow} S_3$ since S_3 appears before any other post-dominators of C_1 on any path from C_1 to END; in Figure 2 (B), $C_1 \mathrel{\dot{\diamond}\!\!\rightarrow} S_2$.

Proposition 1. *$\mathrel{\dot{\diamond}\!\!\rightarrow}$ is an inverted tree rooted at END.*

One can encode $\mathrel{\dot{\diamond}\!\!\rightarrow}$ as a *post-dominance tree* [14, 12] with *END* at its root. Using post-dominance, *direct* control dependence can be defined as in [12]:

Definition 4. *Node v is* ***directly control dependent*** *on node u, written $u \overset{dcd}{\rightsquigarrow} v$, iff (1) there exists a $u - v$ path α such that v post-dominates every node in α different from u; and (2) u is $\underline{\text{not}}$ post-dominated by v.*

For example, in Figure 2 (A), S_1 and S_2 are directly control dependent on C_1 but S_3 is not; while in Figure 2 (B), S_1 is directly control dependent on C_1 but S_2 is not. In Figure 2 (C), S_1 is directly control dependent on C_1 but not on C_2 (because S_1 does not post-dominate C_1). Note that direct control dependence is *not* a partial order on nodes: in Figure 2 (C), C_1 and C_2 are directly control dependent on each other.

The notion of *direct* control dependence has been widely used in program analysis, e.g., in program slicing [13, 18] and compiler construction [12], where it was called just "control dependence". However, this relation only captures the *direct* dependence among statements; it does *not* capture *indirect* dependence. Recall, e.g., that in Figure 2 (C), S_1 does *not* directly control depend on C_2; however, note that once C_2 is reached, *the execution of* S_1 *depends on the control decision made at* C_2! Therefore, S_1 *control depends* on C_2 by all means, suggesting that the terminology proposed in [12] for control dependence is, perhaps, not the most appropriate one. We will shortly see that S_1 is in the *transitive closure* of the direct control dependence on C_2; for some reason, this transitive closure of direct control dependence was misleadingly called "strong control dependence" in [15]. We will call it simply "control dependence" in what follows, because we think it captures best the *dependence* of some statements on the *control* decision made by others. As an example of an application where (indirect) control dependence is needed in the context of information flow, see [11]. Another use of it appears in the context of debugging multithreaded systems (see the discussion in Section 1 on predictive runtime analysis regarding the sample execution trace in Figure 1); e.g., in Figure 2 (C), if $C_1C_2C_1S_1S_2$ is an execution, the analysis needs to know that S_1 also depends on the choice made at C_2 to *not* exit the loop, which is caused by an indirect control dependence in the CFG.

In fact, even before direct control dependence was introduced in [12], Dennings already discussed the indirect influence of control statements on the program flow in [11]. It was also called the *range* of branches in [19], which is nothing but the transitive closure of direct control dependence, as informally mentioned in [12, 16] without proof. Podgurski and Clarke [15] called it "strong control dependence", to emphasize that it was stronger than their "weak" control dependence, still without proving that it was the transitive closure of the direct control dependence, thus leading to a slightly inconsistent terminology: for a relation R (control dependence in their case) "strong R" ended up strictly including R. For reasons explained above, we prefer to drop the adjective "strong" and call it just control dependence:

Definition 5. *Node v is **control dependent** on u, written $u \overset{cd}{\rightsquigarrow} v$, iff there exists some $u - v$ path that does* not *contain $ipd(u)$, the immediate post-dominator of u.*

For example, in Figure 2 (C), $C_2 \overset{cd}{\rightsquigarrow} S_1$. One can prove the following properties of control dependence, all of which follow from our parametric framework:

Corollary 2. *(Follows by Theorem 1 and Proposition 4) For $\overset{dcd}{\rightsquigarrow}$ and $\overset{cd}{\rightsquigarrow}$, the following hold:*

1. *$\overset{cd}{\rightsquigarrow} = \overset{dcd^+}{\rightsquigarrow}$ (one cannot replace $\overset{dcd^+}{\rightsquigarrow}$ by $\overset{dcd^*}{\rightsquigarrow}$ because $\overset{cd}{\rightsquigarrow}$ needs not be reflexive);*
2. *If $u \overset{cd}{\rightsquigarrow} v$ then $PostDom(u) \subseteq PostDom(v)$; in particular, $ipd(v) \diamond\!\!\rightarrow ipd(u)$;*
3. *$u \overset{cd}{\rightsquigarrow} v$ iff there exists some $u - v$ path α such that $\alpha \cap PostDom(u) = \emptyset$.*

Therefore, control dependence is nothing but the transitive closure of the direct control dependence, so it is a relation *weaker* than the direct control dependence.

2.2 Weak Control Dependence

Although control dependence now also captures "indirect" dependence, it still has another important limitation: it is insensitive to (non-terminating) loops; e.g., in Figure 2 (C), S_2 is *not* control dependent on C_1 (the former is the post-dominator of the latter). This may lead, e.g., to incorrect runtime analysis of multi-threaded systems. Reconsider the execution in Figure 1. Suppose it is generated by the program in Figure 2 (C). More specifically, suppose that e_1 is a write on the shared variable j, e_2 is the following read on j generated by C_1, e'_3 is the write on j generated by S_1, and e_3 is the write on z generated by S_2. One may think that e_3 is *not* control dependent on e_2 by definition, that is, that e_3 will happen regardless of e_2. However, since the loop is potentially non-terminating, S_2 may *never be executed* at runtime. Thus, the observed existence of e_3 is a consequence of a fortunate control choice made by C_1 when e_2 took place. Therefore, e_3 *should be control dependent* on e_2. Podgurski and Clarke [15] introduced strong post-dominance to handle control dependence in the presence of loops:

Definition 6. *Node u is* ***strongly post-dominated*** *by v, written $u \overset{s}{\diamond\!\!\rightarrow} v$, iff (1) $u \diamond\!\!\rightarrow v$ and (2) there is some integer $k \geq 1$ s.t. every u–path of length larger than or equal to k passes through v. Node v is a* ***proper strong post-dominator*** *of u iff $u \overset{s}{\diamond\!\!\rightarrow} v$ and $u \neq v$.*

In other words, u is strongly post-dominated by v iff u is post-dominated by v and there is no infinite u–path that does *not* pass through v; e.g, in Figure 2 (B), S_2 does not strongly post-dominate C_1, because there is an infinite path from C_1 that will not pass through S_2, while in Figure 2 (D), S_1 is strongly post-dominated by C_2 but C_2 is not strongly post-dominated by S_3. There may be no proper strong post-dominators for some nodes; e.g., in Figure 2 (C), neither C_1 nor C_2 have proper strong post-dominators, since they can choose to either stay in the loop forever or jump out of it. Based on strong post-dominance, weak control dependence is defined in [15] as follows:

Definition 7. *Node v is* ***directly weakly control dependent*** *on u, written $u \overset{dwcd}{\rightsquigarrow} v$, iff u has successors u' and u'' s.t. $u' \overset{s}{\diamond\!\!\rightarrow} v$ but u'' is not strongly post-dominated by v;* ***weak control dependence****, written $\overset{wcd}{\rightsquigarrow}$, is the transitive closure of $\overset{dwcd}{\rightsquigarrow}$.*

In Figure 2 (D), $C_1 \overset{dwcd}{\rightsquigarrow} S_4$ because $S_2 \overset{s}{\diamond\!\!\rightarrow} S_4$ but not $S_1 \overset{s}{\diamond\!\!\rightarrow} S_4$. Weak control dependence is a generalization of control dependence, that is, every control dependence is a weak control dependence. This was informally mentioned in [15], but it is not straightforward to prove it rigorously using their original definitions. However, it will follow as a corollary of our parametric framework, as shown at the end of Section 3. What makes this result even more interesting is that *direct* weak control dependence is *not* a generalization of *direct* control dependence. E.g., in Figure 2 (D), S_3 is directly control dependent but not directly weak control dependent on C_1, while it is directly weak control dependent but not directly control dependent on C_2. Weak control dependence is not a partial order either: e.g., in Figure 2 (C), both $C_1 \overset{dwcd}{\rightsquigarrow} C_2$ and $C_2 \overset{dwcd}{\rightsquigarrow} C_1$. The (direct) weak control dependence makes the worst-case assumption that all loops are non-terminating, which is very rarely the case in practice. In fact, most loops *terminate*.

3 Parametric Control Dependence

We next propose a parametric framework to define and reason about control dependence, which incorporates both direct control dependence and direct weak control dependence, as well as their indirect variants, as special cases. This framework can be easily instantiated to define other control dependence relations, such as the termination-sensitive control dependence discussed in Section 4. It is fair to say that here we do *not* intend to generalize *all* approaches to control dependence. For example, we believe that the nice recent work in [16] on extending control dependence to work with CFGs with more than one or with no end nodes could also be cast as an instance of a parametric framework, but it is not trivial and we do not attempt to explicitly capture that here. Also, we believe that the symbolic approach in [3] which interprets CFGs as Kripke structures and then calculating post-dominators by efficient fair CTL model-checking queries, can be also extended to well-presentable properties on paths, like our "parameters" below, but again, we do not intend to investigate this interesting problem here.

Definition 8. *A set $\mathcal{P} \subseteq Paths(CFG)$ is a* ***prefix-invariant property*** *on paths iff (1) $\mathcal{P}(\lambda_{END})$; and (2) $\mathcal{P}(\alpha\pi) \Leftrightarrow \mathcal{P}(\pi)$ for any $\alpha\pi \in Paths(CFG)$ (α is finite). A $u \overset{\mathcal{P}}{-}$**path** is any u–path in $\mathcal{P}$. Node u is* ***$\mathcal{P}$-post-dominated*** *by node v, written $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$, iff all $u \overset{\mathcal{P}}{-}$paths contain v.* ***PostDom***$_{\mathcal{P}}(u)$ *is the set of $\mathcal{P}$-post-dominators of u different from u.*

From now on in this section, we fix a prefix-invariant property $\mathcal{P}$. One can show that $\mathcal{P}$ contains all $u - END$ paths, that is, $\mathcal{P}(\alpha)$ holds for any $u - END$ path α. By Definition 1 (*END* is reachable from any u), there exists at least one finite $u \overset{\mathcal{P}}{-}$path. Note that for some nodes u, $PostDom_{\mathcal{P}}(u)$ can be empty. For example, as shown after Definition 6, some nodes may not have strong post-dominators, which will be shown shortly to be a special case of $\mathcal{P}$-post-dominators for a well chosen property $\mathcal{P}$.

Proposition 2. *For $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$, the following hold:*

1. *$\overset{\mathcal{P}}{\diamond\!\!\rightarrow} \subseteq \diamond\!\!\rightarrow$, that is, $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ implies $u \diamond\!\!\rightarrow v$;*
2. *$\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$ is a partial order;*
3. *If $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ and there is a $u - u'$ path that does not contain v, then $u' \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$;*
4. *If $v_1 \neq v_2 \in PostDom_{\mathcal{P}}(u)$, then either $v_1 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_2$ or $v_2 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_1$; in other words, $\langle PostDom_{\mathcal{P}}(u), \overset{\mathcal{P}}{\diamond\!\!\rightarrow}\rangle$ is a total order;*
5. *If $PostDom_{\mathcal{P}}(u) \neq \emptyset$ then $PostDom_{\mathcal{P}}(u)$ has a unique first element w.r.t. $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$;*
6. *$\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$ is a forest of inverted trees, where $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ iff $v = ipd_{\mathcal{P}}(u)$, where $ipd_{\mathcal{P}}(u)$ is the first element of $\langle PostDom_{\mathcal{P}}(u), \overset{\mathcal{P}}{\diamond\!\!\rightarrow}\rangle$, called the* ***immediate $\mathcal{P}$-post-dominator*** *of u.*

One can show that post-dominance and strong post-dominance are two special cases of $\mathcal{P}$-post-dominance by choosing appropriate parameters $\mathcal{P}$: let $\mathcal{P}_{\perp}$ denote the set of all finite paths ending with *END* and let $\mathcal{P}_{\perp\infty}$ be the union of $\mathcal{P}_{\perp}$ with all infinite paths.

Proposition 3. *Both* $\mathcal{P}_\perp$ *and* $\mathcal{P}_{\perp\infty}$ *are prefix-invariant, and* $\diamond\!\!\rightarrow = \overset{\mathcal{P}_\perp}{\diamond\!\!\rightarrow}$ *and* $\overset{s}{\diamond\!\!\rightarrow} = \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow}$.

We will discuss a third special case of $\mathcal{P}$-post-dominance in Section 4, where additional termination information of loops will be taken into account.

Definition 9. *Node* v *is* ***directly*** $\mathcal{P}$***-control dependent*** *on* u*, written* $u \overset{d\mathcal{P}}{\rightsquigarrow} v$*, iff: (1) there is a* $u-v$ *path s.t.* v $\mathcal{P}$*-post-dominates its nodes except* u*; (2)* v *does* <u>*not*</u> $\mathcal{P}$*-post-dominate* u*. Node* v *is* $\mathcal{P}$***-control dependent*** *on* u*, written* $u \overset{\mathcal{P}}{\rightsquigarrow} v$*, iff there exists some* $u-v$ *path that does not contain* $ipd_{\mathcal{P}}(u)$*.*

Note that $\overset{d\mathcal{P}}{\rightsquigarrow}$ is *not* a partial order. For example, $\overset{dcd}{\rightsquigarrow}$ and $\overset{dwcd}{\rightsquigarrow}$, which will be shortly shown to be special cases of $\overset{d\mathcal{P}}{\rightsquigarrow}$, are not partial orders. This means that, in the worst case, the time needed to compute the transitive closure of $\overset{d\mathcal{P}}{\rightsquigarrow}$ is $O(|V|^3)$ [10].

Theorem 1. *For* $\overset{d\mathcal{P}}{\rightsquigarrow}$ *and* $\overset{\mathcal{P}}{\rightsquigarrow}$*, the following hold:*

1. $\overset{\mathcal{P}}{\rightsquigarrow} = \overset{d\mathcal{P}}{\rightsquigarrow}^{+}$;
2. *If* $u \overset{\mathcal{P}}{\rightsquigarrow} v$ *then* $PostDom_{\mathcal{P}}(u) \subseteq PostDom_{\mathcal{P}}(v)$*; in particular,* $ipd_{\mathcal{P}}(v) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}}(u)$*;*
3. $u \overset{\mathcal{P}}{\rightsquigarrow} v$ *iff there exists some* $u-v$ *path* α *such that* $\alpha \cap PostDom_{\mathcal{P}}(u) = \emptyset$*.*

One can also show that direct control dependence and direct weak control dependence are two special cases of direct $\mathcal{P}$-control dependence, while control dependence and weak control dependence are two special cases of $\mathcal{P}$-control dependence:

Proposition 4. $\overset{dcd}{\rightsquigarrow} = \overset{d\mathcal{P}_\perp}{\rightsquigarrow}$ *and* $\overset{dwcd}{\rightsquigarrow} = \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$*, and* $\overset{cd}{\rightsquigarrow} = \overset{\mathcal{P}_\perp}{\rightsquigarrow}$ *and* $\overset{wcd}{\rightsquigarrow} = \overset{\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$*.*

The following proposition will allow us to *compare* control dependencies, based on just a simple comparison of their corresponding parameters:

Proposition 5. *If* $\mathcal{P} \subseteq \mathcal{P}'$ *are prefix-invariant properties then: (1)* $\overset{\mathcal{P}'}{\diamond\!\!\rightarrow} \subseteq \overset{\mathcal{P}}{\diamond\!\!\rightarrow}$*; (2)* $PostDom_{\mathcal{P}'}(u) \subseteq PostDom_{\mathcal{P}}(u)$*; (3)* $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}'}(u)$*; and (4)* $\overset{\mathcal{P}}{\rightsquigarrow} \subseteq \overset{\mathcal{P}'}{\rightsquigarrow}$*.*

Corollary 3. $\overset{cd}{\rightsquigarrow} \subseteq \overset{\mathcal{P}}{\rightsquigarrow}$ *for any prefix-invariant property* $\mathcal{P}$*; in particular,* $\overset{cd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$*.*

Interestingly, the inclusion of the direct versions of the dependences in the corollary above does *not* hold. For example, it is *not* the case that $\overset{dcd}{\rightsquigarrow} \subseteq \overset{dwcd}{\rightsquigarrow}$.

4 Termination-Sensitive Control Dependence

Weak control dependence takes loops into account using strong post-dominance, which is more suitable for proving total correctness of programs [15] than classic control dependence. However, weak control dependence unfortunately makes the worst-case assumption about the termination of loops in the program, namely, all loops are assumed to be potentially infinite. Considering the fact that *most loops terminate* in real programs, this assumption is too conservative in practice. Let us look at the example in Figure 2 (D). The loop containing S_1 and C_2 obviously terminates, so S_3 will be eventually executed once C_2 is reached. In other words, the execution of S_3 *does not depend* on the choice made at C_2. However, by Definition 7, $C_2 \overset{wcd}{\rightsquigarrow} S_3$. Such over-restrictive assumptions may bring *false positives* to static program analysis, while for our runtime predictive analysis, they may generate over-restrictive control dependences on events, reducing the number of potential permutations of events when investigating possible actual executions, resulting in more *false negatives*, i.e., a reduced coverage.

In this section, we introduce a new control dependence relation, named *termination-sensitive control dependence*, as another instantiation of the parametric control dependence framework presented in Section 3. As indicated by its name, this control dependence takes the termination information of loops into account to improve the precision of program analyses that make use of control dependence. Although termination analysis is an undecidable problem, there exist some effective algorithms to approximately determine termination of programs, e.g., [9, 5] (more discussion on these algorithms is out of the scope of this paper). Besides, termination information can also be provided by users (e.g., using special annotations) or detected by heuristics-based criteria (for example, a loop whose condition is $i < n$ and in which i is increased at each iteration will always terminate). Here we only focus on defining a more precise control dependence relation using existing termination information, which is assumed to be correct.

Definition 10. *A* ***termination-sensitive control flow graph*** $\langle V, E, START, END, V_\infty\rangle$ *is a CFG* $\langle V, E, START, END\rangle$ *together with a distinguished set of nodes* $V_\infty \subseteq V$.

The nodes in V_∞ can be thought of as nodes that can lead to non-terminating executions. In practice, one would like to annotate as few statements as possible to provide the termination information; if that is the case, then V_∞ can contain precisely the conditions of those loops that may not terminate.Theoretically, one can add to V_∞ *all* the unavoidable statements in such loops, but this is not necessary. Besides, some of these statements can themselves be loops, but ones which terminate. From here on, we fix an arbitrary termination-sensitive CFG and define complete paths as follows:

Definition 11. *A* ***complete path*** π *is a path that is either finite and ends with END, or is infinite and* $inf(\pi) \cap V_\infty \neq \emptyset$*, where* $inf(\pi)$ *gives those nodes visited infinitely often in* π*. Let* $\mathcal{P}_\top$ *denote the set of complete paths of the termination-sensitive CFG.*

Note that infinite paths generated by "nested" loops in which the outer ones are annotated as "non-terminating" (in V_∞), while the inner ones are "terminating", are considered complete as far as the outer loop is executed infinitely often. One may want to instead annotate the "terminating" nodes as a subset $V_\top \subseteq V$ and then require the complete path to satisfy $inf(\pi) \cap V_\top = \emptyset$; while this is reasonable and fits our parametric setting as well, such an approach would be less precise, because it would exclude common paths as the ones generated by nested loops as above. There is an interesting similarity between termination-sensitive CFGs and Buchi automata [6], where the role of *accepting states* is played by V_∞ and that of *accepted words* by complete paths.

One can show that $\mathcal{P}_\top$ is also a prefix-invariant property on paths. Indeed, for any $u - v$ path α and v–path π, $\alpha\pi$ is a $u - END$ path iff π is a $v - END$ path. Besides, if $\alpha\pi$ is infinite, then since α is finite, $inf(\alpha\pi) = inf(\pi)$. Therefore, $inf(\alpha\pi) \cap V_\infty = inf(\pi) \cap V_\infty$; in particular, $inf(\alpha\pi) \cap V_\infty \neq \emptyset$ iff $inf(\pi) \cap V_\infty \neq \emptyset$. Based on the parametric framework for control dependence introduced in Section 3, we can define corresponding post-dominance and dependence notions: $\mathcal{P}_\top$-post-dominance, immediate $\mathcal{P}_\top$-post-dominance, direct $\mathcal{P}_\top$-control dependence, and $\mathcal{P}_\top$-control dependence. The following results follow immediately from the generic framework in the previous section:

Corollary 4. *For $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$, the following hold:*

1. $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} \subseteq \diamond\!\!\rightarrow$*, that is, $u \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v$ implies $u \diamond\!\!\rightarrow v$;*
2. $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$ *is a partial order;*
3. *If $v_1 \neq v_2 \in PostDom_{\mathcal{P}_\top}(u)$, then either $v_1 \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v_2$ or $v_2 \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v_1$; in other words, $\langle PostDom_{\mathcal{P}_\top}(u), \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}\rangle$ is a total order;*
4. *If $PostDom_{\mathcal{P}_\top}(u) \neq \emptyset$ then $PostDom_{\mathcal{P}_\top}(u)$ has a unique first element w.r.t. $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$;*
5. $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$ *is a forest of inverted trees;*

Corollary 5. *For $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ and $\overset{\mathcal{P}_\top}{\rightsquigarrow}$, the following hold:*

1. $\overset{\mathcal{P}_\top}{\rightsquigarrow} = \overset{d\mathcal{P}_\top}{\rightsquigarrow}^{+}$;
2. *If $u \overset{\mathcal{P}_\top}{\rightsquigarrow} v$ then $PostDom_{\mathcal{P}_\top}(u) \subseteq PostDom_{\mathcal{P}_\top}(v)$; in particular, $ipd_{\mathcal{P}_\top}(v) \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}_\top}(u)$;*
3. *$u \overset{\mathcal{P}_\top}{\rightsquigarrow} v$ iff there exists some $u - v$ path α such that $\alpha \cap PostDom_{\mathcal{P}_\top}(u) = \emptyset$.*

Now we are ready to define termination-sensitive control dependence and to compare this new control dependence with classical and weak control dependence:

Definition 12. *Let $\overset{tscd}{\rightsquigarrow} := \overset{\mathcal{P}_\top}{\rightsquigarrow}$ be the **termination-sensitive control dependence**.*

Proposition 6. $\overset{cd}{\rightsquigarrow} \subseteq \overset{tscd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$ *(it follows by Proposition 5, since $\mathcal{P}_\perp \subseteq \mathcal{P}_\top \subseteq \mathcal{P}_{\perp\infty}$).*

Note that there are no inclusions between the direct versions of these control dependences, i.e., between $\overset{d\mathcal{P}_\perp}{\rightsquigarrow}$ (or $\overset{dcd}{\rightsquigarrow}$) and $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ or between $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ and $\overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$ (or $\overset{dwcd}{\rightsquigarrow}$). For example, consider the CFG in Figure 2 (D). Suppose that $C_2 \in V_\infty$ (i.e., the loop containing S_1 and C_2 is annotated as "non-terminating"). Then $C_1 \overset{d\mathcal{P}_\perp}{\rightsquigarrow} S_3$ but S_3 is not directly $\mathcal{P}_\top$-control dependent on C_1, while $C_2 \overset{d\mathcal{P}_\top}{\rightsquigarrow} S_3$ but S_3 is not directly control dependent on C_2. Suppose next that $C_2 \notin V_\infty$. Then $C_1 \overset{d\mathcal{P}_\top}{\rightsquigarrow} S_3$ but S_3 is not directly weak control dependent on C_1, while $C_2 \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow} S_2$ but S_2 is not directly $\mathcal{P}_\top$-control dependent on C_2.

By Proposition 6, the set V_∞ acts as a "knob" tuning the precision of the control dependence relation. For example, if $V_\infty = \emptyset$ then termination-sensitive control dependence becomes precisely classic control dependence. If $V_\infty = V$ then it becomes weak control dependence. In practice, V_∞ is somewhere in-between $\emptyset$ and V. However, the more nodes are added to V_∞, the more dependences are added, i.e., the weaker the dependence relation becomes. For example, in Figure 2 (C), suppose that $C_2 \notin V_\infty$. Then S_2 is not termination-sensitive control dependent on C_2. But if the user declares that $C_2 \in V_\infty$ despite the actual semantics of the program, we will have $C_2 \overset{tscd}{\rightsquigarrow} S_2$.

Ideally, one would like to pick a V_∞ which would generate a *minimal* set of complete paths $\mathcal{P}_\top$ that includes all the actual execution paths of the program to analyze. Unfortunately, the selection of such an optimal V_∞ is difficult to achieve, because one would need to automatically prove termination of loops, an undecidable problem. A safe approach would be to start with $V_\infty = V$, and then remove from it all the statements which are not loop conditions, then all those loop conditions controlling terminating loops which can be detected by heuristic criteria or declared so by users or code generators.

5 Control Scope

The *control scope* of a conditional statement is the set of statements that control depend on it, where the control dependence is *termination-sensitive* and *indirect*. In other words, S is in the control scope of C iff the execution of S depends upon a fortunate choice made by C. Algorithms to compute direct control dependence [12] and direct weak control dependence [4] are well-known. These algorithms take linear time to detect all the statements that *directly* depend upon a given statement C, and can be used to construct program dependence graphs (PDG) [13], which are widely adopted in program slicing. These linear algorithms to calculate control dependencies are sufficient in applications where high online speed is not crucial and where only the direct dependencies are necessary, such as debugging. However, there are applications that need the transitive versions of the control dependences. For example, in [19], the (indirect) control dependence is used to define and reason about information flow in security, and in [15], (indirect) weak control dependence is used to prove total correctness of programs. Also, in predictive runtime analysis, one prefers to calculate all the dependencies statically and then spend constant time at runtime to check whether the statements generating two events depend upon each other, to reduce the runtime overhead.

From here on, by control dependence we mean *termination-sensitive control dependence*. Statically calculating all the direct dependencies for all the statements can therefore be achieved in $O(|V|^2)$, since the parameter property on paths that leads to our control dependence fits the framework in [4]. However, it is not clear how to effectively calculate *indirect* control dependencies. A blind application of the transitive closure of direct dependence would yield an $O(|V|^3)$ algorithm (since direct control dependence is not a partial order), which can be impractical even on relatively small programs. Without additional information about the program which generates the CFG, there is nothing that one can do to decrease the complexity of calculating control dependence. However, CFGs are typically generated from actual code that is stored as lines of sequences of characters in files. In what follows, we augment the CFG with code references and show that, under some mild and common restrictions, we can calculate the entire control dependence relation in $O(|V|^2)$, which is the same as the complexity of calculating direct control dependence. These results appear to be new even for classic and weak control

```
1: if (i > 0)
2:   then x = 1;
3:   else y = 1;
4: z = x;
```

(A)

```
1: do {
2:   y ++;
3: } while (y < n);
4: z = y;
```

(B)

```
1: while (i > 0) {
2:   if (j < 0)
3:     then break;
4:     else i--;
5:   j -- ;
6: }
7: z = i;
```

(C)

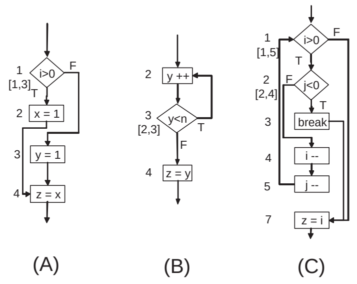


Fig. 3. Sample programs and their SCFGs

dependence relations, both special cases of our (termination-sensitive) control dependence. It may seem that $O(|V|^2)$ is still impractical in large applications; however, in the case of predictive runtime analysis or unit testing, we only need to calculate the control scopes for relatively small units, e.g., only intra-procedurally.

The nodes of a CFG generally correspond to either *simple statements* (ones that do not contain sub-statements) or to conditions that are part of *compound statements* (ones that contain sub-statements); these are formalized in Definition 14. We consider two types of compound statements, conditionals and loops; note that although a programming language may also support other kinds of compound statements, e.g., `try..catch`, such statements are decomposed into simple statements when constructing the CFG, so they need not appear explicitly in the CFG (they appear only implicitly, encoded by corresponding edges). Even though CFGs capture faithfully the control flow of a program, unfortunately, precious structural information about the program, such as where a compound statement starts and where it ends, is generally not reflected in a CFG.

In what follows we augment CFGs with structural information by adding to each node a corresponding unique line, or code reference number, which can be thought of as the position in the program of the statement corresponding to that node. The reference numbers of all nodes are assumed distinct. Since there is a one-to-one correspondence between (simple and compound) statements in the program and nodes in the CFG, we can identify statements with the reference numbers of their corresponding nodes. Since the corresponding node in the CFG of a loop is its condition, the reference number of a statement is not necessarily the line number where that statement starts! E.g., the reference of `do..while` in Fig. 3 (B) is 3. We next formalize this:

Definition 13. *A* ***sequential CFG (SCFG)*** $\langle V, E, START, END, \#, \flat\rangle$ *is a CFG together with injective maps* $\# : V \to \mathbb{N}$ *and* $\flat : V_C \to Intervals(\mathbb{N})$ *such that: (1)* $\#(C) \in \flat(C)$ *for any* $C \in V_C$*; and (2)* $\flat(C') \subset \flat(C)$ *for any* $C \neq C' \in V_C$ *with* $\#(C') \in \flat(C)$.

# associates to each node (simple statement with out-degree 1 or condition part of a compound –conditional or loop– statement with out-degree 2) a unique number. $\flat$ returns for each condition the code boundaries of its compound statement, as an interval bounded by the smallest and the largest reference numbers of nodes in the SCFG covered by that statement; some statements *may include but not overlap* other statements.

Fig. 3 shows some SCFGs. Nodes are shown in ascending order of and labeled with their line numbers; conditions are also labeled with their statement boundaries. The computation of the $\flat$ function is straightforward and can be done at parse time at no additional cost. For example, in Fig. 3 (A), $\flat(1) = [1,3]$; in (B), $\flat(3) = [2,3]$; and in (C), $\flat(1) = [1,6]$. For each SCFG, one can define a function $next : V - V_C - \{END\} \to \mathbb{N}$, which associates to each node $S \in V - V_C - \{END\}$ the number $\#(S')$ where $(S, S') \in E$ is the unique outgoing edge from S. For "jump" statements, including `break`, `continue`, `return`, and exception throwing, *next* is the reference number of the statement that S jumps to; e.g., in Fig. 3 (C), $next(3) = 7$. If S is a simple non-jump statement at the end of a loop body, then $next(S)$ is the reference number of the loop statement; e.g., in (B), $next(2) = 3$, and in (C), $next(5) = 1$. For all other simple statements, the *next* function simply returns the reference number of the next statement in the program; e.g., in (A), $next(2) = next(3) = 4$, and in (C), $next(4) = 5$. We can identify statements in the program with their corresponding nodes in the SCFG. From here on, we call *all* the nodes in an SCFG statements and define the following SCFG terminology:

Definition 14. *Nodes in V_C are called **compound statements** and those in $V - V_C$ are called **simple statements**. If C is compound statement and S is any statement with $\#(S) \in \flat(C)$ then S is a **sub-statement** of C, or C **contains** S; if additionally there is no proper sub-statement C' of C that properly contains S then S is a **direct sub-statement** of C.*

The requirements of SCFGs are common to all programing languages. Most higher level structured programming languages, such as Java and C#, impose additional restrictions on jump statements; e.g., `continue`, `break`, `return`, exception throwing, can only jump to specific positions determined statically at compile time. We next define a corresponding version of SCFG that captures formally such restrictions on jumps:

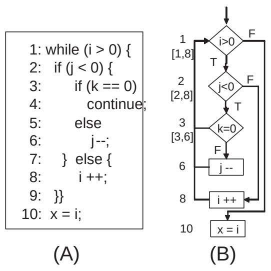
Definition 15. *A **structured SCFG (SSCFG)** is an SCFG $\langle V, E, START, END, \#, \flat\rangle$ s.t.: (1) Each compound statement C has a unique entry point, $entry(C)$, which is the lower bound of $\flat(C)$; if $\#(S) \notin \flat(C)$ and $next(S) \in \flat(C)$ then $next(S) = entry(C)$; and (2) Backward control flows can only be caused by loops: for any $(S, S') \in E$ with $\#(S) > \#(S')$, there is a compound statement C such that $\#(S) \in \flat(C)$ and $\#(S') = entry(C)$; in this case, we call C a **loop statement**; all compound statements which are not loops are called **conditional statements**. For every loop statement L, we let **next**(L) be the statement following L, i.e., $next(L) := max(\#(S_1), \#(S_2))$ where $(L, S_1), (L, S_2) \in E$.*

We next focus on computing the control scope of compound statements. The *control scope* of a compound statement C is the set of statements that are control-dependent on C. Unfortunately, such statements can be spread all over the program, thus making their precise bookkeeping hard. We show that in the context of an SSCFG, the statements that control depend on a compound statement C are located in a *window*, or an *interval*, of references, say *scope* (C), which we call *control scope interval*. Note that our use of intervals is *not* related to the concept of (maximal) interval discussed in [2] and used in elimination methods [17]. The control scope intervals may be larger than the control scopes, but we show that the extra statements can be efficiently detected. In other words, *scope* (C) characterizes unambiguously the statements that are control-dependent on C.

An immediate observation is that all sub-statements of a compound statement are control dependent on it. Besides, a jump statement from within a compound statement C may extend the control scope of C. For example, in Fig. 3 (C), the `break` statement extends the scope of the `if` statement to the end of the loop, thus making statement 5 control-depend on the compound statement 2. This can be formalized as follows:

Definition 16. *Given C a compound statement with $\flat(C) = [b_1, b_2]$, let **pre-scope**(C) be $\flat(C)$ when C is a loop statement, and $[b_1, max(b_2, next(J_1) - 1, .., next(J_n)) - 1]$ when C is a conditional statement, where J_i for $i \in [1, n]$ are the direct substatements of C.*

For example, in Fig. 3 (C), the pre-scope of the loop is $[1, 6]$ while the pre-scope of the `if` statement is $[2, 6]$. Note that in this definition, the pre-scopes of loop statements do *not* consider the effects of their direct sub-statements (when, e.g., an exception is thrown or a break/continue for an outer loop) because, as we discuss below, the backward edges of loops cause a different situation to handle. Pre-scopes of statements can be calculated at no additional cost at parse time, since the targets of jumps are known statically (we focus on intra-procedure analysis here; exceptions not caught in the analyzed procedure, are assumed to jump to the end of the procedure). Note, however, that the pre-scope of C may already contain statements that do *not* control-depend on C: e.g., in Fig. 4, the pre-scope of the conditional 3 is $[3, 8]$ (due to the `continue` statement), so 8 is in pre-scope(3); however, 8 does *not* control-depend on 3. To filter out such statements, we next introduce a new relation between statements:



```
 1: while (i > 0) {
 2:   if (j < 0) {
 3:       if (k == 0)
 4:          continue;
 5:       else
 6:          j--;
 7:    }  else {
 8:        i ++;
 9:   }}
10:  x = i;
```

Fig. 4.

Definition 17. *Statement S' is **forward-reachable** from S iff there exists an $S - S'$ path that contains no loop statement L such that L contains both S and S'.*

In Fig. 3 (C), node 3 is reachable but not forward-reachable from 4, and in Fig. 4, statement 8 is reachable but not forward-reachable from statement 3. Although the intuition for forward-reachability is "from S one can go forward and reach S'", it is *not* always the case that one can find an $S - S'$ path with increasing reference numbers: in Fig. 4, statement 10 is forward-reachable from 2, but the path between them always contains 1. Next proposition gives an effective way to compute forward-reachability:

Proposition 7. *Given statements S and S' in an SSCFG G, S' is forward-reachable from S iff S' reachable from S in a graph that replaces each edge $e = (n_1, n_2)$ with $n_1 > n_2$ in G (i.e., one corresponding to a loop L with $entry(L) = n_2$), by $(n_1, next(L))$.*

The following allows us now to relate the pre-scopes and control dependence:

Proposition 8. *If $\#(S) \in$ pre-scope(C) and S forward-reachable from C, then $C \overset{tscd}{\rightsquigarrow} S$.*

Definition 18. *A **control scope interval** of C is one that contains: (1) all nodes that control depend on C; and (2) only forward-reachable nodes that control-depend on C.*

Recall that the control scope of a compound statement C is the set of all statements that control-depend on C, and note that a control scope interval of C can contain statements that are not forward-reachable but still control depend on C.

We next describe an $O(|V|^2)$ algorithm that computes control scope intervals for *all* the compound statements. Theorem 2 (given below) will then provide us an efficient procedure to extract the actual control scopes from our control scope intervals, that is, to filter out all the statements in the control scope interval of each C that do not control-depend on C.

Let us depict prescopes on SSCFGs, like in Fig. 5. The ranges of arrows give the prescopes of the statements; forward arrows represent branch statements and backward arrows represent loop statements. There are two types of overlapped prescopes, shown in Fig. 5 (A) and (B). In the first case, C_2 is forward reachable from C_1. Then the control scope interval of C_1 should contain that of C_2: consider $S_1 \notin$ *pre-scope*(C_1) in (A); C_1 may choose the branch with C_2 and then skip S_1, so $C_1 \overset{tscd}{\rightsquigarrow} S_1$. In the second case, C_1 and C_2 must have the same control scope intervals: in (B), the execution of S_1 in the second iteration of the loop depends on the choice made at C_1 in the first iteration. When *pre-scope*(C_1) overlaps several nested loops, like in (C), then all loops must have the same control scope interval as C_1. Based on these observations, we can derive the following algorithm which is explained in more detail in [8]:

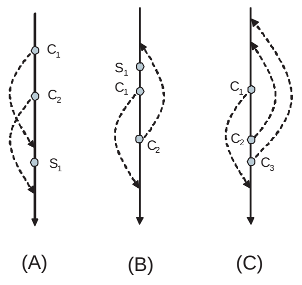


Fig. 5. Prescopes overlap

(Step 1) Extend prescopes (Fig. 5 (A)) by a backward traversal of the code/SSCFG: if prescopes of two statements overlap, then extend the prescope of the outer statement accordingly;
(Step 2) Compute equivalence classes of statements that have the same control scope (Fig. 5 (B) and (C)); these are precisely the *connected components* of the graph representing the overlap between loops and other conditionals;
(Step 3) Compute the actual control scope interval of each equivalence class as the *union* of the extended prescopes of all the statements in that class; if the class contains loops in V_∞, then the upper bound of its interval is set to ∞.

Steps 1 and 2 take $O(|V|^2)$ and step 3, which also takes the termination information of loops into account, takes $O(|V|)$. To calculate the actual control scopes, all one needs to do is to remove from control scope intervals those statements that are not control-dependent. The following theorem gives us a simple way to do it:

Theorem 2. *$C \overset{tscd}{\rightsquigarrow} S$ iff $\#(S)$ is in the control scope interval of C, and S is forward-reachable from C or there is some loop L with $\hat{C} = \hat{L}$ (same equiv class) and $S \in \flat(L)$.*

6 Conclusion

This paper presented three novel contributions to control dependence. First, it introduced *parametric control dependence* as a general framework to define various control dependence relations, both direct and indirect. Second, it defined a new control dependence relation, called *termination-sensitive control dependence*, generalizing both classic and weak control dependence by taking explicit termination information of loops into account. Finally, an $O(|V|^2)$ algorithm was described to compute the (indirect) control dependence of all the statements; this algorithm works only for languages without arbitrary jumps inside blocks, including Java and C# (but not C). It would be interesting to also incorporate the recent work on control dependence in [16] in a parametric framework. Another question is whether one can combine the TSCD analysis with data-flow analysis and extend the algorithm in Section 5 with inter-procedural analysis.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.
2. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.
3. B. Aminof, T. Ball, and O. Kupferman. Reasoning about systems with transition fairness. In *the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004.
4. G. Bilardi and K. Pingali. A framework for generalized control dependence. In *PLDI'96*, 1996.
5. A. R. Bradley, Z. Manna, and H. Sipma. Termination analysis of integer linear loops. In *the 16th International Conference on Concurrency Theory (CONCUR'05)*, 2005.
6. J. Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.
7. F. Chen and G. Roşu. Predicting concurrency errors at runtime using sliced causality. Technical Report UIUCDCS-R-2005-2660, Dept. of CS at UIUC, 2005.
8. F. Chen and G. Roşu. Parametric and termination-sensitive control dependence. Technical Report UIUCDCS-R-2006-2712, Dept. of CS at UIUC, 2006.
9. M. Colon and H. Sipma. Practical methods for proving program termination. In *CAV'02*, 2002.
10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
11. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
12. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
13. S. Horwitz and T. W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, 1992.
14. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.
15. A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.
16. V. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *The European Symposium on Programming (ESOP'05)*, 2005.
17. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.
18. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centre for Mathematics and Computer Science, 1994.
19. M. Weiser. Program slicing. In *ICSE'81*, 1981.