An Alternative Characterization of Weak Order Dependence

Torben Amtoft[a], Kelly Androutsopoulos[b], David Clark[b], Mark Harman[b], Zheng Li[b]

[a] *Department of Computing and Information Sciences, Kansas State University, Manhattan KS 66506, USA*
[b] *CREST, Department of Computer Science, King's College London, Strand, London, United Kingdom.*

Abstract

Control dependence forms the basis for many program analyses, such as program slicing. Recent work on control dependence analysis has led to new definitions of dependence that cater to reactive programs with their necessarily non–terminating computations. One important such definition is the definition of Weak Order Dependence, which was introduced to generalize classical control dependence for a control flow graph (CFG) without exit nodes. In this paper we show that for a CFG where all nodes are reachable from each other, weak order dependence can be expressed in terms of traditional control dependence where one node has been converted into an end node.

Keywords:
Control dependence, Programming languages

1. Introduction

Dependence analysis allows us to capture, formalise and investigate the potential and actual interactions between parts of a software system. These interactions may be complex. However, because a change potentially has an impact upon its transitive dependents, understanding these dependencies is an important part of assessing and managing the process of software development and its improvement.

Dependence analysis has proved to be widely applicable, with application areas including program comprehension [10], software maintenance [9], testing and debugging [3, 11], impact analyses [5, 20], virus detection [15], integration [4], refactoring [14], restructuring, reverse engineering and reuse [6].

Email addresses: tamtoft@ksu.edu (Torben Amtoft), kellyandrou@googlemail.com (Kelly Androutsopoulos), david.j.clark@kcl.ac.uk (David Clark), mark.harman@kcl.ac.uk (Mark Harman), zheng.li@kcl.ac.uk (Zheng Li)

Typically research in dependence analysis has considered two primary forms of dependence: data dependence and control dependence. Data dependence is relatively uncontroversial. It occurs between two parts of the program which define and then subsequently use a value. The value computed at the definition is said to flow (by data dependence) to the site at which the value is used. The using site is said to '(data) depend' upon the defining site.

By contrast, control dependence has proved to be a more complex and subtle property, that requires careful formulation. Early formulations of control dependence such as Weiser's 'colour dominance' [21], were superseded by a now more standard definition, which was captured in terms of reachability and post dominance in the Control Flow Graph(CFG) [8]. The definition of Ferrante, Ottenstein and Warren became very widely accepted and was used by many other authors, forming the basis for the popular Program Dependence Graph [16] and System Dependence Graph [12].

However, traditional control dependence applies only to programs that are able to terminate normally, i.e., that have CFGs with a unique exit node. This renders traditional control dependence inapplicable for handling computations that purposefully do not terminate such as in embedded systems controllers, operating systems and other forms of so-called 'reactive systems'. Such systems simply cannot be ignored; it is estimated that by 2010 there will be 16 billion embedded systems worldwide, rising to 40 billion by 2020 [2].

Ranganath et al. [19] and Amtoft [1] address this problem by introducing new definitions of control dependence for arbitrary CFGs, so as to handle CFGs with zero exit nodes (and even CFGs with multiple exit nodes). Furthermore, the authors categorise control dependence as being either non-termination sensitive or non-termination insensitive.

Non-termination sensitive control dependence (NTSCD) [19] is sensitive to non-termination and when used together with Decisive Order Dependence (DOD) [19] leads to slices that preserve termination properties i.e. the observable behaviour of the slice is infinite exactly when the original is.

Weak Order Dependence (WOD) [1] is insensitive to non-termination and leads to slices that allow the termination domain to increase i.e. loops that do not influence relevant values are sliced away. This definition has been shown to generalise traditional notions of control dependence given in terms of CFGs with a unique exit node. However, it requires researchers to abandon the definitions and concepts with which they have been familiar for over two decades. This paper seeks to overcome this conceptual difficulty by showing how this new notion of control dependence can be re-formulated in terms of traditional dependence for a wide class of Control Flow Graphs.

The class of control flow graphs we consider is those for which every node is reachable from every other node. This is a very wide class of graphs; it includes, for example, those originally envisaged by Ferrante, Ottenstein and Warren in their seminal paper on the Program Dependence Graph [8]. Some of our results can even be applied to graphs that are not strongly connected, as long as the nodes in question are reachable from all other nodes.

Traditional control dependence is a relation between two nodes, in which one

node controls the execution of another. By contrast, weak order dependence, which captures not only the traditional control dependence but also the ordering relationships between nodes in irreducible regions of a CFG, requires a ternary relation. The fundamental insight that underlies our re-formulation is that capturing control dependence in the presence of ordering relations requires the participation of three nodes simply because one of the three has to play the role of a 'local pseudo exit node'. Using this insight, we are able to reformulate Weak Order Dependence (a ternary relation) in terms of traditional control dependence (a binary relation).

The contribution of this paper is a formal proof of the equivalence of our reformulation of the Weak Order Dependence Relation in terms of traditional control dependence, in the style of Ferrante, Ottenstein and Warren. This reformulation allows us to think of the new notions of control dependence in terms of the more familiar control dependence used, for example, in the System Dependence Graph [12] and in Weiser's Program slicing algorithm [22].

2. Preliminaries

A control-flow graph (CFG) is a pair (V, E) where V is the set of nodes, and $E \subseteq V \times V$ is the set of edges. If $(n, m) \in E$ we say that there is an edge from n to m, and that m is a *successor* of n.

Given a CFG G, a *path* from node a to node b is a sequence of nodes, $n_1..n_k$ where $a = n_1$ and $b = n_k$, such that for all $i \in 1 \ldots k-1$, G contains an edge from n_i to n_{i+1}. We say that the path is non-trivial if $k > 1$, and we say that b is reachable from a if there is a path from a to b.

Definition 1. *An* end node *of a CFG is a node e such that*

- *e has no outgoing edges, and*
- *e is reachable from all nodes.*

Fact 1. *A CFG can have at most one end node.*

For a node q in a CFG G, we define $G/_{[q \to \cdot]}$ as the CFG that results from removing all outgoing edges from q. More formally, we have

Definition 2. *Given $G = (V, E)$ with $q \in V$, $G/_{[q \to \cdot]} = (V, E')$ where $E' = \{(n, m) \in E \mid n \neq q\}$.*

Fact 2. *Assume that in G, q is reachable from all other nodes. Then $G/_{[q \to \cdot]}$ has q as an end node.*

We now recall the standard notion of *control dependence*, formulated for a graph *with* an end node.

Definition 3. *Consider a CFG G with end node e.*

- *We say that* m postdominates n *if all paths from* n *to* e *contain* m*;*
- *we say that* m strictly postdominates n *if* m *postdominates* n *and* $n \neq m$*.*

Note that e postdominates all nodes.

Definition 4. *Consider a CFG* G *with end node* e*. We say that* $n \xrightarrow{cd} m$ *in* G *if*

1. m *does not strictly postdominate* n*, and*
2. *there is a non-trivial path* $n..m$ *where* m *postdominates all nodes but* n*.*

Lemma 3. *Let* G *have end node* e*. If* $n \xrightarrow{cd} m$ *in* G *then* $n \neq e$ *and* $m \neq e$*.*

Proof: Since there is a non-trivial path from n to m, n has an outgoing edge, and hence $n \neq e$. Since e postdominates all nodes, we thus infer that e strictly postdominates n, and hence $m \neq e$. □

We recall the definition given in [1] of weak order dependence, applicable for *arbitrary* CFGs:

Definition 5. *Given a CFG, we write* $a \xrightarrow{wod} b, c$ *if*

1. *there is a path from* a *to* b *not containing* c*,*
2. *there is a path from* a *to* c *not containing* b*, and*
3. a *has a successor* d *such that either*
   - b *is reachable from* d *and all paths from* d *to* c *contain* b*, or*
   - c *is reachable from* d *and all paths from* d *to* b *contain* c*.*

Fact 4. *If* $a \xrightarrow{wod} b, c$ *then* a,b,c *are distinct nodes.*

3. Weak Order Dependence is Control Dependence

Lemma 5. *Given* G *with nodes* a,b,c *where* b *and* c *are both reachable from all nodes in* G*. If* $a \xrightarrow{wod} b, c$ *in* G *then either*

- $a \xrightarrow{cd} b$ *in* $G/_{[c \rightarrow \cdot]}$*, or*
- $a \xrightarrow{cd} c$ *in* $G/_{[b \rightarrow \cdot]}$*.*

(By Fact 2, $G/_{[c \rightarrow \cdot]}$ *has end node* c*, and* $G/_{[b \rightarrow \cdot]}$ *have end node* b*.)*

Proof: We know (Fact 4) that a, b, c are distinct nodes. Without loss of generality, we can assume[1] that a has a successor d such that

[1] If d had been such that all paths from d to b contained c then we would prove that $a \xrightarrow{cd} c$ holds in $G/_{[b \rightarrow \cdot]}$.

b is reachable from d, and all paths from d to c contain b (1)

and we shall prove that $a \stackrel{cd}{\rightarrow} b$ holds in $G/_{[c\rightarrow\cdot]}$. Since $a \stackrel{wod}{\rightarrow} b, c$ entails that there is a path in G from a to c not containing b, we see that

in $G/_{[c\rightarrow\cdot]}$, b does not postdominate a. (2)

By (1) there exists a minimal path π from d to b; note that

b does not occur in any proper prefix of π. (3)

We now infer that π does not contain c. For assume, in order to get a contradiction, that π does contain c; since $c \neq b$ there would be a proper prefix of π that contains c and by (1) thus also b but this contradicts (3).

Thus π is a path in $G/_{[c\rightarrow\cdot]}$. Since $a \neq c$ we see that

in $G/_{[c\rightarrow\cdot]}$ there is a path $a\pi$ from a to b. (4)

We shall now prove that

in $G/_{[c\rightarrow\cdot]}$, all nodes in π are postdominated by b. (5)

We therefore assume that in $G/_{[c\rightarrow\cdot]}$ there is a path π_1 from n to c where n is part of π, and must establish that π_1 contains b.

In $G/_{[c\rightarrow\cdot]}$ there thus is a path $\pi_0\pi_1$ from d to c where π_0 is a proper prefix of π. But this is also a path in G, so by (1) we infer that $\pi_0\pi_1$ contains b. But by (3), π_0 cannot contain b. Hence π_1 contains b, establishing (5).

From (2,4,5) we see that $a \stackrel{cd}{\rightarrow} b$ does hold in $G/_{[c\rightarrow\cdot]}$. □

4. Control Dependence is Weak Order Dependence

Lemma 6. *Given G with nodes a,b,c where $a \neq c$ and where b is reachable from all nodes in G. If $a \stackrel{cd}{\rightarrow} c$ in $G/_{[b\rightarrow\cdot]}$ then $a \stackrel{wod}{\rightarrow} b, c$ in G.*

Note that we need the assumption $a \neq c$ since $\stackrel{cd}{\rightarrow}$ may relate a node to itself while $\stackrel{wod}{\rightarrow}$ never does that.

Proof: From $G/_{[b\rightarrow\cdot]}$ having b as end node, by Lemma 3 we infer from $a \stackrel{cd}{\rightarrow} c$ that $c \neq b$ and $a \neq b$. Thus a, b, c are distinct nodes.

From $a \stackrel{cd}{\rightarrow} c$ and $a \neq c$ we see that in $G/_{[b\rightarrow\cdot]}$, c does not postdominate a. Thus

there is a path in G from a to b that does not contain c. (6)

In $G/_{[b\rightarrow\cdot]}$ there is a non-trivial path from a to c where c postdominates all nodes but a. We infer that a has a successor d such that

in $G/_{[b\rightarrow\cdot]}$, c is reachable from d, and all paths from d to b contain c. (7)

We infer that in $G/_{[b\rightarrow\cdot]}$ there is a path from d to c that does not contain b, and since $a \neq b$ thus

there is a path in G from a through d to c that does not contain b. (8)

We shall now prove that

all paths in G from d to b contain c. (9)

But if π is a path in G from d to b, it has a (not necessarily proper) prefix π_1 such that π_1 is also a path from d to b but b does not occur in any proper prefix of π_1. Hence π_1 is a path in $G/_{[b\rightarrow\cdot]}$ from d to b, so by (7) we see that π_1 contains c. But then also π contains c, establishing (9).

From (6,8,9) we see that $a \overset{wod}{\rightarrow} b, c$ does hold in G. □

5. Equivalence Between Closures

Definition 6. *Given a CFG $G = (V, E)$. With S a subset of V, we say that S is closed under $\overset{wod}{\rightarrow}$ if $a \in S$ whenever $a \overset{wod}{\rightarrow} b, c$ for $b, c \in S$.*

Definition 7. *Given a CFG $G = (V, E)$, and let S be a subset of V with the property that all nodes in S are reachable from all nodes in V. We then say that S is $_\rightarrow$closed under $\overset{cd}{\rightarrow}$ if $a \in S$ whenever for $b, c \in S$ we have $a \overset{cd}{\rightarrow} b$ in $G/_{[c\rightarrow\cdot]}$.*

Theorem 1. *Given a CFG $G = (V, E)$. Let $S \subseteq V$ satisfy that all nodes in S are reachable from all nodes in V. Then S is closed under $\overset{wod}{\rightarrow}$ iff S is $_\rightarrow$closed under $\overset{cd}{\rightarrow}$.*

Proof: For the "if" direction, we assume $a \overset{wod}{\rightarrow} b, c$ with $b, c \in S$, so as to show $a \in S$. By Lemma 5, applicable since b, c are reachable from all nodes in V, either $a \overset{cd}{\rightarrow} b$ holds in $G/_{[c\rightarrow\cdot]}$ or $a \overset{cd}{\rightarrow} c$ holds in $G/_{[b\rightarrow\cdot]}$. But in both cases, $a \in S$ follows from S being $_\rightarrow$closed under $\overset{cd}{\rightarrow}$.

For the "only if" direction, we assume $a \overset{cd}{\rightarrow} b$ holds in $G/_{[c\rightarrow\cdot]}$ for $b, c \in S$, so as to show $a \in S$. If $a = b$, the claim is trivial. Otherwise, by Lemma 6, we have $a \overset{wod}{\rightarrow} b, c$, and $a \in S$ follows from S being closed under $\overset{wod}{\rightarrow}$. □

6. Related Work

In the introduction, we discuss two ways in which control dependence definitions may vary: they may require program CFGs to have a unique exit node, or not; or they may be non-termination sensitive or insensitive. Table 1 compares the control dependence definitions in the literature according to these variations.

Except for WOD [1] and NTSCD [19] that apply to arbitrary CFGs, all other definitions of control dependence for programs require CFGs to have a unique exit node.

We discuss the main definitions in the literature based on their sensitivity to non-termination.

Table 1: Comparison of control dependence definitions.

Non-Termination	**Unique Exit Node**	**Arbitrary CFG**
Insensitive	Weiser [21] Ferrante et al. [8, 12]	WOD [1]
Sensitive	Podgurski and Clarke [18, 17]	NTSCD and DOD [19]

The earliest definition of control dependence is Weiser's, that is derived from the dependence that Denning and Denning called 'implicit information flow' [7]. This dependence captures the influence a predicate p has over a node, n when p 'controls' whether n is either definitely executed (by selecting one of the two branches emerging from p), or whether it is possibly avoided (by selecting the other branch emerging from p).

Ferrante et al. [8, 12] redefine control dependence in terms of a Control Flow Graph requiring it to have an entry and exit node, such that all nodes can reach the exit node. However, this is not realistic for programs that may, deliberately and of necessity, fail to terminate; such program may not have an exit node in their flow graph.

The first non-termination sensitive definition of control dependence was given by Podgurski and Clarke [18, 17] in 1989 who observed that it is possible for a predicate p to influence the execution of a node n simply by making infinitely many choices (at p) to traverse a branch that leads back to p before n is executed. In this way, a predicate can 'choose' to fail to terminate and, thereby, exclude the execution of some other node that p would otherwise fail to control.

Kamkar [13] (page 166) argues that for some applications of slicing it will be necessary for slicing to use the termination sensitive form of dependence.

7. Conclusion and Future Work

In this paper we show how recent notions of control dependence for non-terminating computations can be re-formulated in terms of traditional notions of control dependence. More precisely, when a,b,c are distinct, a is weakly order dependent on b and c if and only if a is either control dependent on b when all outgoing edges from c have been removed, or control dependent on c when all outgoing edges from b have been removed. This is important because of the fundamental importance of control dependence in many other program analyses such as program slicing.

As a corollary to this work, we would like to explore how NTSCD (augmented with DOD) relates to Podgurski and Clarke's notion of control dependence [18, 17], and see if one can be expressed in terms of another.

[1] T. Amtoft. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Information Processing Letters*, 106(2):45–51, 2008.

[2] ARTEMIS consortium. The embedded computing systems initiative (ARTEMIS), 2007.

[3] D. W. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11 and 12):583–594, 1998.

[4] D. W. Binkley, S. Horwitz, and T. Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, 1995.

[5] S. E. Black. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13:263–279, 2001.

[6] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology Special Issue on Program Slicing*, 40(11 and 12):595–607, 1998.

[7] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, Aug. 1991.

[10] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[11] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, Jan. 2004.

[12] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[13] M. Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing*. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[14] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 155–169, N.Y., Jan. 19–21 2000. ACM Press.

[15] A. Lakhotia and P. Singh. Challenges in getting formal with viruses. *Virus Bulletin*, Sept. 2003.

[16] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices*, 19(5):177–184, 1984.

[17] A. Podgurski. *The signicance of program dependences for software testing, de- bugging, and maintenance*. PhD thesis, Computer and Information Science Department, University of Massachusetts, Amherst, 1989.

[18] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–79, 1990.

[19] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[20] X. Ren, B. G. Ryder, M. Störzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 664–665. ACM, 2005.

[21] M. Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

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