Slicing for Modern Program Structures: a Theory for Eliminating Irrelevant Loops [1]

Torben Amtoft [2]

Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas 66506, USA

Abstract

Slicing is a program transformation technique with numerous applications, as it allows the user to focus on the parts of a program that are relevant for a given purpose. Ideally, the slice program should have the same termination properties as the original program, but to keep the slices manageable, it might be preferable to slice away loops that do not affect the values of relevant variables. This paper provides the first theoretical foundation to reason about non-termination *in*sensitive slicing *without* assuming the presence of a unique end node. A slice is required to be closed under data dependence and under a recently proposed variant of control dependence, called *weak order dependence*. This allows a simulation-based correctness proof for a correctness criterion stating that the observational behavior of the original program must be a *prefix* of the behavior of the slice program.

Key words: program slicing, control dependence, observable behavior, simulation techniques

1. Introduction

Program slicing [12,11] has been applied for many purposes: compiler optimizations, debugging, model checking, protocol understanding, etc. Given the control flow graph (CFG) of a program and given a *slicing criterion*, the sets of nodes of interest, the following steps are involved in slicing:

(i) compute the *slice*, a set of nodes which includes the slicing criterion as well as those nodes that the slicing criterion depends on (directly or indirectly, wrt. data or wrt. control);

(ii) create the *slice program*, essentially by removing the nodes that are not in the slice.

One way to express the correctness of slicing is to demand that the observable behavior of the slice program is "similar" to the observable behavior of the original program. If "similar" implies that one is infinite exactly when the other is, as is often required in applications such as model checking, the slice must include all nodes that may influence the guards of potential loops. This can be achieved by weakening the "control dependence" relation, but the resulting slices may be so large that they are less useful in applications such as program comprehension.

Most previous work on the theoretical foundation of slicing [2,5] assumes that the underlying CFG has a unique end node. This restriction prevents a smooth handling of control structures where methods have multiple exit points, or—more importantly—zero exit points (as in case of indefinitely running reactive systems). As reported in [9,10], the author was part of work that investi-

Email address: `tamtoft at ksu dot edu` (Torben Amtoft).
URL: `http://people.cis.ksu.edu/~tamtoft` (Torben Amtoft).

[1] The author would like to thank John Hatcliff and Venkatesh Prasad Ranganath for many interesting discussions and for comments on a draft of this paper, and also several anonymous referees for useful suggestions.
[2] Supported in part by AFOSR and by Rockwell Collins.

gated notions of control dependencies suitable for handling arbitrary CFGs. The main result was to propose one control dependence ($\overset{ntscd}{\to}$) designed to preserve termination properties, and another ($\overset{nticd}{\to}$) which allows the termination domain to increase; both coincide with classical definitions on CFGs with unique end nodes.

In [9] it is shown, using (weak) bisimulation as is known from concurrency [7] and first proposed for slicing purposes (for multi-threaded programs) in [4], that slicing based on $\overset{ntscd}{\to}$ preserves observable behavior, in particular termination, provided the CFG is reducible (with or without a unique end node). To handle also irreducible CFGs (as is needed to model state charts), [10] proposed several notions of "order dependence", like $\overset{dod}{\to}$ ("decisive") and $\overset{wod}{\to}$ ("weak"), and proved that for an *arbitrary* CFG, slicing based on $\overset{ntscd}{\to}$ and on $\overset{dod}{\to}$ preserves observable behavior. Still, [9,10] does *not* attempt to prove the correctness (modulo termination properties) of slicing based on definitions like $\overset{nticd}{\to}$.

The main contribution of this paper is to provide a result yet missing in the literature: a provably correct slicing technique which is able to handle arbitrary CFGs, including those needed to model reactive systems (and/or state charts), and which allows loops not influencing relevant values to be sliced away.

Our approach explores the abovementioned notion of weak order dependence, to be motivated in Section 3 which argues that it also captures control dependence. Its key virtue is to ensure that each node has a unique "next observable", with an *observable* being a node relevant to the slicing criterion. This allows (Section 4) a crisp correctness proof, establishing a (one-way) simulation property which states that if the original program can do some observable action then so can the slice program. (The reverse does not hold, as the original program may get stuck in some unobservable loop.) First we briefly summarize concepts important to program slicing, most of which are standard (see, e.g., [8,2]) but with a twist similar to [9,10].

2. Standard Definitions

A control flow graph G is a labeled directed graph, with nodes representing statements in a program, and with edges representing control flow. A node is either a *statement node*, having at most one successor, or a *predicate node*, having two successors—one labeled T, and another labeled F. There is a distinguished *start node*, with no[3] incoming edges, from which all nodes are reachable. An *end node* is a node with no outgoing edges; if there is exactly one end node n_e and n_e is reachable from all other nodes, we say that G satisfies *the unique end node property.*

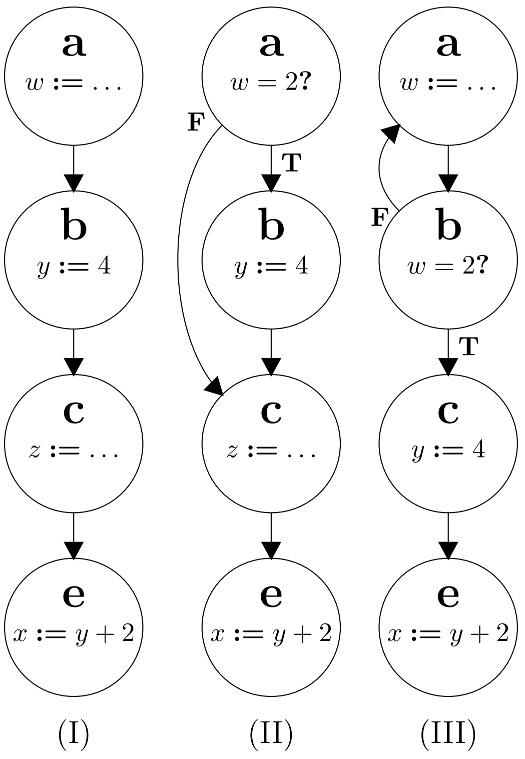

Fig. 1. Examples of CFGs, with unique end node **e**.

To relate a procedure (method) body to its CFG we use a "code map" *code* that maps each CFG node to the code for the corresponding program statement. The function *def* (*ref*) maps each node to the set of program variables defined (referenced) at that node. For example, a statement that branches on the boolean expression B is represented as a predicate node n with $code(n) = B?$ and $def(n) = \emptyset$.

A *path* π in G from n_1 to n_k, written as $[n_1..n_k]$, is a sequence of nodes $n_1, n_2, \ldots, n_k$ where for all $i \in 1 \ldots k-1$, G contains an edge from n_i to n_{i+1}; here k (≥ 1) is the length of the path which is non-trivial if $k > 1$. If there is a path of length k from n to m, but no shorter path, we write $dist^G(n, m) = k$.

To illustrate the standard notions of dependence, consider the CFGs in Fig. 1 which all have a unique

[3] To save space, not all our examples satisfy this, but they can easily be transformed so as to do.

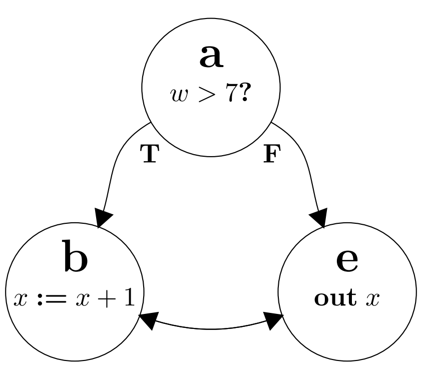

Fig. 2. A CFG (irreducible) with no end node.

end node e, chosen as our slicing criterion.

In (I), e is *data dependent* on b, according to

Definition 1 *Node n is data dependent on m, written $m \overset{dd}{\rightarrow} n$, if there exists a variable v such that $v \in def(m) \cap ref(n)$, and there exists a non-trivial path $[n_1..n_k]$ with $n_1 = m$ and $n_k = n$ where for all $i \in 2 \ldots k-1$, $v \notin def(n_i)$.*

But neither b or e are data dependent on a or c which thus are irrelevant to the slicing criterion, so we may safely update *code* so that $code(a) = code(c) =$ **skip**. The resulting program and the original program behave identically on e.

In (II), it holds that $b \overset{dd}{\rightarrow} e$ so the slice must include b —and also the predicate node a, since otherwise slicing would update $code(a)$ to either *true*? or *false*?, causing b to be possibly either improperly executed or improperly bypassed; in all cases, y might end up with a wrong value. Intuitively, b is *control dependent* on a. For a CFG with a unique end node, the standard way of formulating control dependence (of n on m) is: $m \overset{cd}{\rightarrow} n$ holds if m is not strictly postdominated by n, but there is a non-trivial path $[n_1..n_k]$ with $n_1 = m$ and $n_k = n$ where for all $i \in 2 \ldots k-1$, n_i is postdominated by n. Here n postdominates m if all paths from m to the unique end node contain n; if $n \neq m$ the postdomination is strict.

In (III), we have $c \overset{dd}{\rightarrow} e$, but we do not have $b \overset{cd}{\rightarrow} c$ since b is strictly postdominated by c. But if a and b are sliced away, the slice program will always reach e, while the original program may never reach e. This motivates Podgurski and Clarke's notion of "weak control dependence" [8], where $b \overset{wcd}{\rightarrow} c$ captures that b may cause c to be infinitely delayed. Chen & Rosu [3] present a more flexible approach which annotates loops depending on whether they are known to terminate: if so, $\overset{cd}{\rightarrow}$ can be used; if not, $\overset{wcd}{\rightarrow}$ must be used.

3. Weak Order Dependence

Now consider the CFGs in Figs. 2 and 3, none of which has an end node. In all cases, e is our slicing criterion, with $code(e) =$ **out** x so as to express that the value of x (initialized to 0) is observable when control is at e.

First consider Fig. 2 (where the CFG is even irreducible). The original program will output $1, 2, 3, \ldots$ if $w > 7$, and $0, 1, 2, \ldots$ otherwise. Therefore we must demand that the slice, which does contain b (since $b \overset{dd}{\rightarrow} e$), also contains a. But $a \overset{dd}{\not\rightarrow} b$ and $a \overset{dd}{\not\rightarrow} e$, and there seems no obvious way to generalize $\overset{cd}{\rightarrow}$ or $\overset{wcd}{\rightarrow}$ so that b, or e, becomes control dependent on a. Intuitively, this is because control will *always* hit b and e, no matter what happens at a. Yet a determines the *order* in which b and e are reached. This motivates, as first proposed in [10, Def. 14] though in a slightly different [4] version and not further explored, a *ternary* relation:

Definition 2 *Nodes n_b and n_c are* weakly order dependent *on n_a, written $n_a \overset{wod}{\rightarrow} n_b, n_c$, iff all of the following holds (implying $n_a \neq n_b \neq n_c$):*
(i) there is a path from n_a to n_b not containing n_c;
(ii) there is a path from n_a to n_c not containing n_b;
(iii) n_a has a successor n' such that either
(a) n_b is reachable from n', and all paths from n' to n_c contain n_b; or
(b) n_c is reachable from n', and all paths from n' to n_b contain n_c.

We shall see that for the observable behavior of the original program to be a *prefix* of the observable behavior of the slice program, it is *sufficient* to demand that the slice S_C (containing the slicing criterion C) is closed under $\overset{dd}{\rightarrow}$ and $\overset{wod}{\rightarrow}$. That is, if $n \overset{dd}{\rightarrow} m$ and $m \in S_C$, or if $n \overset{wod}{\rightarrow} n_a, n_b$ and $n_a, n_b \in S_C$, then also $n \in S_C$.

In Fig. 2, $a \overset{wod}{\rightarrow} b, e$ clearly holds (we can use either b or e as the n' of Def. 2). Hence a slice that

[4] Apart from being phrased somewhat differently, the definition of weak order dependence proposed in [10, Def. 14] differs from Def. 2 by requiring that in (iii)(a), all maximal paths from n' contain n_b, and in (iii)(b), all maximal paths from n' contain n_c. To see the difference, consider a CFG where n_a is a predicate node with edges to n'_b and n'_c, n'_b is a predicate node with edges to itself and n_b, and n'_c is a predicate node with edges to itself and n_c. Then n_b and n_c are weakly order dependent on n_a, according to Def 2, but not according to [10, Def. 14] which thus in itself is not sufficient for slicing.

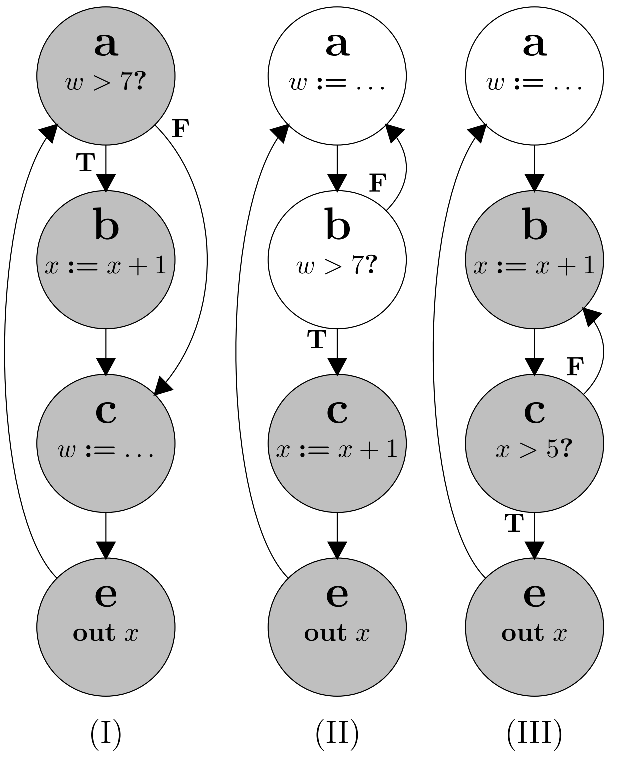

Fig. 3. Examples of CFGs with no end node (slice shaded).

contains e and is closed under $\xrightarrow{wod}$ will indeed contain a, provided it is also closed under $\xrightarrow{dod}$ so as to contain b.

Let us examine how Def. 2 works on the CFGs from Fig. 3, where we have shaded the nodes that end up in the slice. In (I), we clearly have $a \xrightarrow{wod} b, e$, causing a to be in the slice (since $b \xrightarrow{dd} e$). And indeed, a slice not including a would produce observable values either $1, 2, 3, \ldots$ or $0, 0, 0, \ldots$, while the original program may produce say $1, 1, 2, 3, 3, \ldots$.

In (II), we have $c \xrightarrow{dd} e$ but not $b \xrightarrow{wod} c, e$, since there is no path from b to e not containing c. Thus the slice does not include a or b; accordingly, $code(a)$ is updated to **skip**, and $code(b)$ is updated to $true$**?**. (Choosing $code_2(b) = false$**?** would be incorrect, as then the slice program produces no observable output.) The slice program thus yields observable behavior $1, 2, 3, \ldots$, while the original program may yield either the same behavior, or a finite *prefix*, like $1, 2, 3, 4$.

In (III), the original program produces observable behavior $6, 7, 8, \ldots$. It is easy to check that $c \xrightarrow{wod} b, e$ holds, and thus c is in the slice (since $b \xrightarrow{dd} e$). And indeed, not including c would result in a slice with observable behavior $1, 2, 3, \ldots$.

The key property of weak order dependence is that a slice closed under $\xrightarrow{wod}$ has at most one "next observable", as formalized in Def. 3 and Lemma 5. (The importance of this property was recognized already in [10, Thm 5] which states that a sufficient condition is to be closed under $\xrightarrow{ntscd}$ and $\xrightarrow{dod}$, and less explicitly in [9].)

Definition 3 *Given a CFG G, a node n, and a set of nodes S, $obs_S^G(n)$ is the set of nodes $m \in S$ with the property that in G there exists a path $[n_1..n_k]$ ($k \geq 1$) with $n_1 = n$ and $n_k = m$ such that for all $i \in 1 \ldots k-1$, $n_i \notin S$.*

If $n \in S$ we clearly have $obs_S^G(n) = \{n\}$.

Example 4 *In Fig. 3, with S as indicated, for (II) we have $obs_S(a) = obs_S(b) = \{c\}$, and for (III) $obs_S(a) = \{b\}$.*

Lemma 5 *Given a CFG G, and assume that S is closed under $\xrightarrow{wod}$. Then for all n in G, $obs_S^G(n)$ is at most a singleton.*

Proof: Assume, to get a contradiction, that there exists n_1 with $|obs_S^G(n_1)| > 1$. Let n belong to $obs_S^G(n_1)$; thus $n \in S$ and there is a path $\pi = [n_1..n_k]$ with $n_k = n$ such that for all $i \in 1 \ldots k-1$, $n_i \notin S$. Since $obs_S^G(n) = \{n\}$, we infer that there exists $j \in 1 \ldots k-1$ such that $obs_S^G(n_{j+1}) = \{n\}$ but $|obs_S^G(n_j)| > 1$. Let $m \neq n$ belong to $obs_S^G(n_j)$; we shall show that $n_j \xrightarrow{wod} n, m$. Looking at Def. 2, requirements (i) and (ii) obviously hold; concerning requirement (iii), observe that n_{j+1} is a successor of n_j from which n is reachable and from which there is no path to m not containing n (because such a path would contradict $obs_S^G(n_{j+1}) = \{n\}$). Having established $n_j \xrightarrow{wod} n, m$, from $n, m \in S$ and S being closed under $\xrightarrow{wod}$ we infer $n_j \in S$, yielding the desired contradiction. □

4. Formalizing Slicing and its Correctness

The technical development is implicitly parameterized wrt. a given CFG G, and a slice S_C which contains the slicing criterion C. Slicing is defined only if all nodes have at most one next observable, as is the case (Lemma 5) when S_C is closed under $\xrightarrow{wod}$. Much as[5] in [9,10], the slice program has the same CFG as the original program but different code maps:

[5] A main difference is that in those papers, if n is a predicate node not in S_C then $code_2(n)$ is a non-deterministic choice.

Definition 6 *Slicing transforms a code map $code_1$ into a code map $code_2$ given by*

(i) if $n \in S_C$, then $code_2(n) = code_1(n)$;

*(ii) if $n \notin S_C$ and n is a statement node, then $code_2(n) =$ **skip**;*

(iii) if $n \notin S_C$ and n is a predicate node, with successors n_T labeled T and n_F labeled F, then

(a) if $obs^G_{S_C}(n)$ is a singleton $\{m\}$ then

*(i) if m is reachable from n_T with $dist^G(n_T, m) < dist^G(n, m)$, then $code_2(n) = true$**?**;*

*(ii) otherwise, if m is reachable from n_F with $dist^G(n_F, m) < dist^G(n, m)$, then $code_2(n) = false$**?**;*

*(b) if $obs^G_{S_C}(n) = \emptyset$, then $code_2(n) = true$**?**. (Choosing "false**?**" would also work.)*

The clauses a.i and a.ii are exhaustive, since if m is reachable from n then at least one of the successors of n will be closer to m than n is. An implementation would let $code_2(n)$ be an unconditional jump to m, thus changing the CFG, but we leave this optimization to a post-processing phase so as to keep the core of the correctness proof conceptually simple.

Example 7 *Consider Fig. 3 (II) where $S_C = \{c, e\}$. Since $obs_{S_C}(b) = \{c\}$, with $dist(c, c) < dist(b, c) < dist(a, c)$, we have $code_2(a) =$ **skip**, $code_2(b) = true$**?**, $code_2(c) = x$ **:=** $x + 1$, and $code_2(e) =$ **out** x.*

The *semantics* of a program is phrased in terms of transitions on program states, where a program state s is of the form (n, σ) with n a node and σ a store mapping variables to values. For $i = 1, 2$, we write $i \vdash s \to s'$ if $code_i$ transforms state s into state s': if $code_i(n) = x$ **:=** E then $i \vdash (n, \sigma) \to (n', \sigma')$ where n' is the successor of n and $\sigma' = \sigma[x \mapsto v]$ with $v = [\![E]\!]\sigma$ (the value of E in σ); if $code_i(n) = B$**?** then $i \vdash (n, \sigma) \to (n', \sigma)$ where n' is the T-successor of n if $[\![B]\!]\sigma = true$, and the F-successor otherwise.

Using this (deterministic) semantics, we now define labeled transitions, with the label identifying the observable node (if any) whose code been executed; for $i = 1, 2$ we write

– $i \vdash (n, \sigma) \xrightarrow{n} s'$ if $i \vdash (n, \sigma) \to s'$ and $n \in S_C$ (observable move);

– $i \vdash (n, \sigma) \xrightarrow{\tau} s'$ if $i \vdash (n, \sigma) \to s'$ and $n \notin S_C$ (silent move);

– $i \vdash s \overset{\tau}{\Rightarrow} s'$ for the reflexive transitive closure of $i \vdash s \xrightarrow{\tau} s'$;

– $i \vdash s \overset{n}{\Rightarrow} s'$ if there is s_1 with $i \vdash s \overset{\tau}{\Rightarrow} s_1$ and $i \vdash s_1 \xrightarrow{n} s'$. (Note that we do not allow silent moves *after* the observable move.)

An easy induction in $dist(n, m)$ yields

Lemma 8 *For all (n, σ), if $obs(n) = \{m\}$ then $2 \vdash (n, \sigma) \overset{\tau}{\Rightarrow} (m, \sigma)$.*

Definition 9 (Weak Simulation) *A binary relation ϕ is a weak simulation if whenever*

$s_1 \, \phi \, s_2$ and $1 \vdash s_1 \overset{n}{\Rightarrow} s'_1$

there exists s'_2 such that

$s'_1 \, \phi \, s'_2$ and $2 \vdash s_2 \overset{n}{\Rightarrow} s'_2$.

That is, if the original program can perform an observable action then so can the slice program, but not necessarily vice versa. We now prepare for defining a relation R that is a weak simulation.

Definition 10 (Relevant Variables) *For a node n we define $rv(n)$, the set of relevant variables at n, by stipulating that $v \in rv(n)$ iff there exists $m \in S_C$ and a path $[n_1..n_k]$ with $n_1 = n$ and $n_k = m$ such that $v \in ref(n_k)$, but for all $i \in 1 \ldots k-1$, $v \notin def(n_i)$.*

Strictly speaking, we should have defined, for $i = 1, 2$, functions ref_i / def_i to return the variables referenced/defined at node n wrt. code map $code_i$, and functions rv_i and relations $\xrightarrow{dd}_i$ which are parameterized wrt. ref_i and def_i. However, the following result [10, Sect. 5, Lemma 7] shows that we can safely ignore the subscripts:

Lemma 11 *Assume, with $\xrightarrow{dd}_i$ etc. as defined just above, that S_C is closed under $\xrightarrow{dd}_1$. Then* (i) *S_C is closed also under $\xrightarrow{dd}_2$;* (ii) *for all n, $rv_1(n) = rv_2(n)$.*

Proceeding much as in [10, Sect. 5, Lemma 8], one can prove [1, Lemma 4]

Lemma 12 *Assume that S_C is closed under $\xrightarrow{dd}$, with each $obs(n)$ at most a singleton. For n_a, n_b such that $obs(n_a) = obs(n_b)$ we have $rv(n_a) = rv(n_b)$.*

We are now ready to define a relation R which is a weak simulation if S_C is closed under $\xrightarrow{dd}$ and $\xrightarrow{wod}$.

Definition 13 *$(n_1, \sigma_1) \; R \; (n_2, \sigma_2)$ holds iff*

(i) $obs(n_1) = obs(n_2)$

(ii) for all $v \in rv(n_1)$ $(= rv(n_2))$, $\sigma_1(v) = \sigma_2(v)$.

Example 14 *Consider Fig. 3 (II) where $S_C = \{c, e\}$. Here $(n_1, \sigma_1) \; R \; (n_2, \sigma_2)$ iff $\sigma_1(x) = \sigma_2(x)$ and either $n_1 = n_2 = e$ or $n_1 \neq e$, $n_2 \neq e$.*

The proof that R is a weak simulation is split into two subparts: Lemma 16 is for when the original program makes an observable move, in which case also the slice program must make an observable move (perhaps preceded by some silent moves); Lemma 15 is for when the original program makes a silent move (on its way to an observable), in which case the slice program does not need to move.

Lemma 15 *Let S_C be closed under $\xrightarrow{dd}$ and $\xrightarrow{wod}$. If $s_1 \; R \; s_2$ and $1 \vdash s_1 \xrightarrow{\tau} s'_1$*

and if with $s_1' = (n_1', \sigma_1')$ we have $obs(n_1') \neq \emptyset$, then $s_1' \; R \; s_2$.

Proof: Let $s_i = (n_i, \sigma_i)$ $(i = 1, 2)$. By assumption, there exists $n \in S_C$ such that $obs(n_1') = \{n\}$; since $n_1 \notin S_C$ we infer $obs(n_1) = \{n\}$. From $s_1 \; R \; s_2$ we see that $obs(n_2) = \{n\}$, and by Lemma 12 we have $rv(n_1) = rv(n_2) = rv(n_1') = rv(n)$.

To establish $s_1' \; R \; s_2$, we must consider $v \in rv(n_1)$ and show $\sigma_1'(v) = \sigma_2(v)$. But from $n_1 \notin S_C$ we infer $v \notin def(n_1)$ and hence $\sigma_1'(v) = \sigma_1(v)$, and since from $s_1 \; R \; s_2$ we have $\sigma_1(v) = \sigma_2(v)$, this yields $\sigma_1'(v) = \sigma_2(v)$. $\square$

Lemma 16 *Let S_C be closed under $\overset{dd}{\rightarrow}$ and $\overset{wod}{\rightarrow}$. If*
$s_1 \; R \; s_2$ and $1 \vdash s_1 \overset{n}{\rightarrow} s_1'$
then there exists s_2' such that
$s_1' \; R \; s_2'$ and $2 \vdash s_2 \overset{n}{\Rightarrow} s_2'$.

Proof: Let $s_i = (n_i, \sigma_i)$ $(i = 1, 2)$; from $1 \vdash (n_1, \sigma_1) \overset{n}{\rightarrow} s_1'$ we infer $n_1 = n \in S_C$. From $(n_1, \sigma_1) \; R \; (n_2, \sigma_2)$ and $obs(n_1) = \{n\}$ we infer $obs(n_2) = \{n\}$ and that

$$\forall v \in rv(n) : \sigma_1(v) = \sigma_2(v). \tag{1}$$

In particular, $\sigma_1(v) = \sigma_2(v)$ for all $v \in ref(n)$; therefore the slice program agrees with the original program on the outcome of any test in n (if n is a predicate node). With $s_1' = (n_1', \sigma_1')$, there thus exists $s_2' = (n_2', \sigma_2')$ with $n_1' = n_2'$ such that $2 \vdash (n, \sigma_2) \overset{n}{\rightarrow} s_2'$. From $obs(n_2) = \{n\}$, Lemma 8 gives $2 \vdash (n_2, \sigma_2) \overset{\tau}{\Rightarrow} (n, \sigma_2)$, yielding $2 \vdash (n_2, \sigma_2) \overset{n}{\Rightarrow} s_2'$.

We must also establish $(n_1', \sigma_1') \; R \; (n_2', \sigma_2')$ but since $n_1' = n_2'$ our only remaining proof obligation is, given $v \in rv(n_1')$, to show $\sigma_1'(v) = \sigma_2'(v)$.

- if $v \in def(n)$, then there exists E such that $code_1(n) = code_2(n) = (v := E)$; thus $\sigma_1'(v) = [\![E]\!]\sigma_1$ and $\sigma_2'(v) = [\![E]\!]\sigma_2$. From (1) we infer $[\![E]\!]\sigma_1 = [\![E]\!]\sigma_2$ and thus $\sigma_1'(v) = \sigma_2'(v)$.
- if $v \notin def(n)$, then $v \in rv(n)$, and the claim follows from (1) since $\sigma_1'(v) = \sigma_1(v) = \sigma_2(v) = \sigma_2'(v)$.
$\square$

Theorem 17 *Assume S_C is closed under $\overset{dd}{\rightarrow}$ and $\overset{wod}{\rightarrow}$. Then R (cf. Def. 13) is a weak simulation.*

Proof: Our assumption is that $s_1 \; R \; s_2$ and $1 \vdash s_1 \overset{n}{\Rightarrow} s_1'$, that is there exists $s_1^1 \ldots s_1^k$ $(k \geq 1)$ with $s_1^1 = s_1$ such that $1 \vdash s_1^k \overset{n}{\rightarrow} s_1'$, and $1 \vdash s_1^i \overset{\tau}{\rightarrow} s_1^{i+1}$ for all $i \in 1 \ldots k-1$. With $s_1^i = (n_1^i, \_)$ we have $obs(n_1^i) = \{n\}$ for all $i \in 1 \ldots k$, so we can repeatedly apply Lemma 15 to infer $s_1^k \; R \; s_2$. By Lemma 16 we now infer from $1 \vdash s_1^k \overset{n}{\rightarrow} s_1'$ that there exists s_2' such that $s_1' \; R \; s_2'$ and $2 \vdash s_2 \overset{n}{\Rightarrow} s_2'$. $\square$

5. Discussion

We have provided the first theoretical foundation for an approach to intraprocedural slicing which can handle even control flow graphs without end nodes, and which allows loops that do not influence relevant variables to be sliced away. Our approach is based on the notion of "weak order dependence" (first mentioned in [10] but not further explored) which generalizes classical notions of control dependence: for a CFG *with* unique end node n_e, one can show [1] that if S_C is closed under $\overset{cd}{\rightarrow}$ then S_C is also closed under $\overset{wod}{\rightarrow}$, and that if S_C is closed under $\overset{wod}{\rightarrow}$ and contains n_e then S_C is also closed under $\overset{cd}{\rightarrow}$ (since if $n \overset{cd}{\rightarrow} m$ with $m \neq n$ then $n \overset{wod}{\rightarrow} m, n_e$). We expect that an algorithm for computing weak order dependence can be constructed along similar lines as those given in [10, Sect. 6] for other kinds of dependencies, e.g., $\overset{dod}{\rightarrow}$.

An early work on the correctness of slicing was by Horwitz et al. [6] who use a semantics based multi-layered approach to reason about slicing in the realm of data dependence. Assuming the unique end node property, the correctness result of Ball & Horwitz [2] states that at each node in the slicing criterion, the sequence of observed values for the original program and the sequence of observed values for the slice program are identical, or—if termination properties differ—one is a prefix of the other. Note that simulation, arguably more elegant as it is expressed within a very general framework, is in principle a strictly stronger correctness property since it requires observable nodes to execute in the same order, even if they operate on disjoint sets of variables. Hatcliff et al. [5] provide a very detailed correctness proof but only consider the case where *both* executions terminate; for how to deal with non-termination, they refer to [4] which proposes a correctness property based on a variant of bisimulation but does not work out the details. The correctness results by Ranganath et al. [9,10], cf. Sect. 1, were the first that did not assume the unique end node property, but as they are based on bisimulation, they apply only to slicing that preserves observable behavior. One contribution of this paper is to argue that for practical applications of slicing, an asymmetric notion of bisimulation is needed.

References

[1] T. Amtoft, Correctness of practical slicing for modern program structures, technical report CIS TR 2007-3, Kansas State University, May 2007.

[2] T. Ball, S. Horwitz, Slicing programs with arbitrary control flow, in: 1st International Workshop on Automated and Algorithmic Debugging, vol. 749 of LNCS, Springer-Verlag, 1993.

[3] F. Chen, G. Rosu, Parametric and termination-sensitive control dependence, in: SAS'06, vol. 4134 of LNCS, Springer-Verlag, 2006.

[4] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, H. Zheng, A formal study of slicing for multi-threaded programs with JVM concurrency primitives, in: SAS'99, vol. 1694 of LNCS, Springer-Verlag, 1999.

[5] J. Hatcliff, M. B. Dwyer, H. Zheng, Slicing software for model construction, Higher-Order and Symbolic Computation 13 (4) (2000) 315–353.

[6] S. Horwitz, P. Pfeiffer, T. Reps, Dependence analysis for pointer variables, in: PLDI'89 (Programming Language Design and Implementation), 1989.

[7] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[8] A. Podgurski, L. A. Clarke, The implications of program dependencies for software testing, debugging, and maintenance, in: ACM SIGSOFT'89, 3rd Symposium on Software Testing, Analysis, and Verification, 1989.

[9] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, J. Hatcliff, A new foundation for control-dependence and slicing for modern program structures, in: M. Sagiv (ed.), ESOP 2005, Edinburgh, vol. 3444 of LNCS, Springer-Verlag, 2005.

[10] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, 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.

[11] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (1995) 121–189.

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