

Correctness of Practical Slicing for Modern Program Structures

Torben Amtoft

Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506, USA
tamtoft at ksu dot edu

Abstract

Slicing is a program transformation technique with numerous applications, since it allows the user to focus on the parts of a given program that are relevant for a given purpose. Ideally, the sliced program should have the same termination properties as the original program, but to achieve this, the slicing algorithm must include in the slice all commands that influence the guards of potential loops. For practical purposes, so as to keep the slices manageable, it might be better to slice away loops that do not affect the values of relevant variables.

This paper presents foundational work that accomplishes this goal for arbitrary control flow graphs, whereas previous approaches have assumed the presence of a unique end node; therefore, the proposed approach is able to handle the control flow graphs that arise from modern program structures, such as when modeling reactive systems. A slice set is required to be closed under data dependency and under a certain variant of control dependency, called “weak order dependency”. This allows a crisp correctness proof, based on simulation techniques, for a correctness criterion stating that the observational behavior of the original program must be a prefix of the behavior of the sliced program.

Keywords: *program slicing, observable behavior, simulation techniques, control dependencies.*

1 Introduction

Program slicing [24, 23] has been applied for many purposes: compiler optimizations [5], debugging [6], model checking [11] and protocol understanding [15]. Given a program, represented as a CFG (control flow graph), and given a *slicing criterion*, the sets of nodes of current interest, the following steps are involved in slicing:

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

2. create the *sliced program*, essentially by removing the nodes that are not in the slice set.

One way to express the correctness of slicing is to demand that the observable behavior of the sliced program is “similar” to the observable behavior of the original program. Ideally, we would take “similar” to imply that one is infinite exactly when the other is (called “strong” slicing in [9]), something which requires the slice set to include all nodes that may influence the guards of potential loops. To achieve this, it is necessary to make “control dependence” a weaker relation, sucking in *more* nodes. In practice, this may yield slices that are so large that they become less useful for their purposes. Therefore, in some applications, one might settle for a more liberal correctness criterion.

Most previous work on the theoretical foundation of slicing makes assumptions on the underlying CFGs that may significantly impair their relevance to the building of practical slicers for realistic applications. In particular, it is often taken for granted that a CFG should have a unique end node; this restriction prevents a smooth handling of control structures where methods have multiple exit points, or—more importantly—are designed to run indefinitely, as in reactive systems.

As reported in [19, 20], the author was part of a team investigating notions of control dependencies suitable for handling arbitrary CFGs. The main result¹ was to propose one control dependence, \xrightarrow{ntscd} , designed to preserve termination properties, and another, \xrightarrow{nticd} , which allows the termination domain to increase; both coincide with classical definitions on CFGs with unique end nodes.

In [19] it is indeed shown, using (weak) bisimulation as is known from concurrency [16] and first proposed for slicing purposes in [10] which treated multi-threaded programs, that slicing based on \xrightarrow{ntscd} preserves observable behavior, in

¹Some of the new definitions have been implemented in Indus [21], a publicly available Java slicer whose interface allows the user to tune the size of the slice by selecting which dependencies to consider. (In addition to \xrightarrow{ntscd} and \xrightarrow{nticd} , there are several options regarding concurrency, a feature outside the scope of the present paper.)

particular termination, provided the CFG is reducible (with or without a unique end node). To handle also irreducible CFGs (which is needed to model state charts), [20] proposed several notions of “order dependence”, like \xrightarrow{dod} (“decisive”) and \xrightarrow{wod} (“weak”), and proved that for an *arbitrary* CFG, slicing based on \xrightarrow{ntscd} and on \xrightarrow{dod} preserves observable behavior. On the other hand, the work reported in [19, 20] does not attempt to prove the correctness (modulo termination properties) of slicing based on definitions like \xrightarrow{nticd} that yield more manageable slices than does \xrightarrow{ntscd} .

This paper provides a result missing in the literature so far: 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 is based on, and explores, the abovementioned notion of weak order dependence; in Section 3 we shall motivate our choice and observe that this variant of order dependence also captures an essential part of control dependence. To be more precise, it guarantees that each node in the CFG has a unique “next observable”, with an *observable* being a node relevant to the slicing criterion. In Section 4 we shall see that this is a key property which allows a crisp correctness proof, where correctness is expressed as a (one-way) simulation property, stating that if the original program can do some observable action then so can the sliced program. (The reverse does not hold, as the original program may get stuck in some unobservable loop.) To gradually motivate and explain our approach, which as shown in Section 5 coincides with classical approaches when the CFG has a unique end node, we first give a brief summary of concepts important to program slicing, most of which are quite standard (see, e.g., [18, 2]).

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 must be a distinguished *start node* n_s such that n_s has no incoming edges, and such that all nodes are reachable from n_s . (In some of our examples, to save space, the start node *does* have an incoming edge.) 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 has the *unique end node property*.

To relate a procedure (method) body to its CFG we use a *code map*, a function $code$ which maps a CFG node to the code for the program statement that corresponds to that node. The function def maps each node to the set of vari-

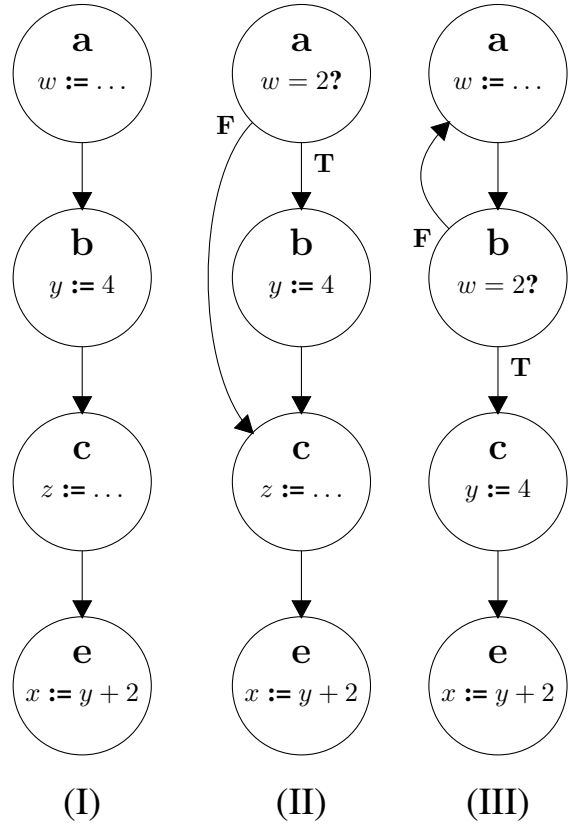


Figure 1. Examples of CFGs (end node e unique), with *code* applied to nodes.

ables defined (*i.e.*, assigned to) at that node (always a singleton or empty set), and ref maps each node to the set of variables referenced at that node. Specifically, an assignment statement $x := E$ is represented as a statement node n with $code(n) = (x := E)$ and $def(n) = \{x\}$ and $ref(n) = fv(E)$ (the free variables of E); a **goto** statement (or a **break** statement) is represented as a statement node n with $def(n) = ref(n) = \emptyset$; a branching statement, branching on the boolean expression B , is represented as a predicate node n with $code(n) = B?$ and $def(n) = \emptyset$ and $ref(n) = fv(B)$.

Given G , a *path* π in G from n_1 to n_k , written $[n_1..n_k]$, is a sequence of nodes n_1, n_2, \dots, n_k such that for all $i \in 1 \dots k - 1$, there is an edge in G from n_i to n_{i+1} . Here $k \geq 1$; if $k > 1$ we say that the path is non-trivial. If there is a path $[n_1..n_k]$ from n to m where $n = n_1$ and $m = n_k$ but no shorter path, we say that $dist^G(n, m) = k$. Note that for all n , $dist^G(n, n) = 1$.

To illustrate the standard notions of dependence, we now consider the CFGs in Fig. 1 which all have a unique end node e . In all cases, we shall choose e as our slicing criterion.

In (I), e is *data dependent* on b , according to the following definition.

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

Referring back to (I), neither b or e are data dependent on a or c . That is, the assignments in a and in c are irrelevant to the slicing criterion, so we may safely update *code* so that $\text{code}(a) = \text{code}(c) = \text{skip}$. This is essentially what slicing does; the resulting program and the original program behave identically on e .

In (II), it holds that $b \xrightarrow{dd} e$, so b must be included in the slice set. We also need to include the conditional a in the slice set, since otherwise slicing would update $\text{code}(a)$ to either *true?* or *false?*. In the former case, b might be improperly executed, and in the latter case, b might be improperly bypassed; in all cases, y might end up with a wrong value. Intuitively, b is control dependent on a . For a CFG with unique end node, the standard way of formulating that n is control dependent on m is: $m \xrightarrow{cd} n$ holds if m is not strictly postdominated by n , but there exists a non-trivial path $[n_1..n_k]$ with $n_1 = m$ and $n_k = n$ such that for all $i \in 2 \dots k - 1$, n_i is postdominated by n . Here we have employed the concept of postdomination: we say that m postdominates n if all paths from n to the unique end node contain m ; if $n \neq m$ the postdomination is strict.

In (III), we have $c \xrightarrow{dd} e$, but we do not have $b \xrightarrow{cd} c$ since b is strictly postdominated by c . Therefore, we might consider slicing away a and b , by updating $\text{code}(a)$ to **skip**, and updating $\text{code}(b)$ to *true?*. (Equivalently, we could update $\text{code}(b)$ to **skip** and remove the edge from b to a , but it is technically convenient to let the original program and the sliced program have the same CFG.) This will change termination properties: the sliced program will always reach e , while the original program may never reach e .

If we want to preserve termination properties, we must express that even though control from b never bypasses c , b may cause c to be infinitely delayed. This is captured by weak control dependence [17], written $b \xrightarrow{wcd} c$. Using this notion, the slice set will include b , and hence even a , cf. the previous discussion of how slices may be large if termination is to be preserved.

3 A Suitable Dependence Relation

To motivate our variant of control dependence, we now consider the CFGs in Figs. 2 and 3, none of which has an end node (as is typical for reactive system). In all cases, we shall choose e as our slicing criterion, where $\text{code}(e) = \text{out } x$ so as to emphasize that the value of x is observable when control is at node e . In our examples, we shall assume that the initial value of x is 0.

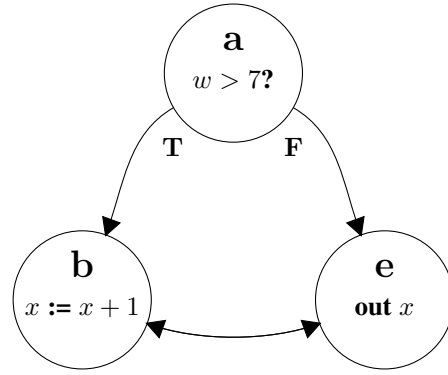


Figure 2. A CFG (irreducible) with no end node (*code* applied to nodes).

First consider Fig. 2 (where the CFG is even irreducible). The original program will output 1, 2, 3, ... if $w > 7$, and output 0, 1, 2, ... otherwise. We have $b \xrightarrow{dd} e$, so the slice set will contain b . But also a needs to be included, since otherwise slicing would update $\text{code}(a)$ to either *true?* or *false?*. In the former case, the sliced program would execute b before it executes e and thus produce observable output 1, 2, 3, ... regardless of the value of w ; in the latter case, a dual conflict might occur.

We would thus like to express that e and b are dependent on a . But clearly there is no data dependence: $a \not\xrightarrow{dd} b$ and $a \not\xrightarrow{dd} e$. Moreover, there seems no obvious way to generalize \xrightarrow{cd} or \xrightarrow{wcd} so that b , or e , becomes control dependent on a . Intuitively, this is because control will *always* hit b as well as e , no matter what happens at a . On the other hand, a determines the *order* in which b and e are reached. This motivates a *ternary* relation, first introduced (in a slightly different version) in [20]:

Definition 2 Nodes n_b and n_c are weakly order dependent on n_a , written $n_a \xrightarrow{wod} n_b, n_c$, iff all of the following holds (implying $n_a \neq n_b \neq n_c$):

1. there exists a path from n_a to n_b not containing n_c ;
2. there exists a path from n_a to n_c not containing n_b ;
3. 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 ;
 - (b) n_c is reachable from n' , and all paths from n' to n_b contain n_c .

In Fig. 2, $a \xrightarrow{wod} b, e$ holds: there exists a path from a to b not containing e ; there exists a path from a to e not containing

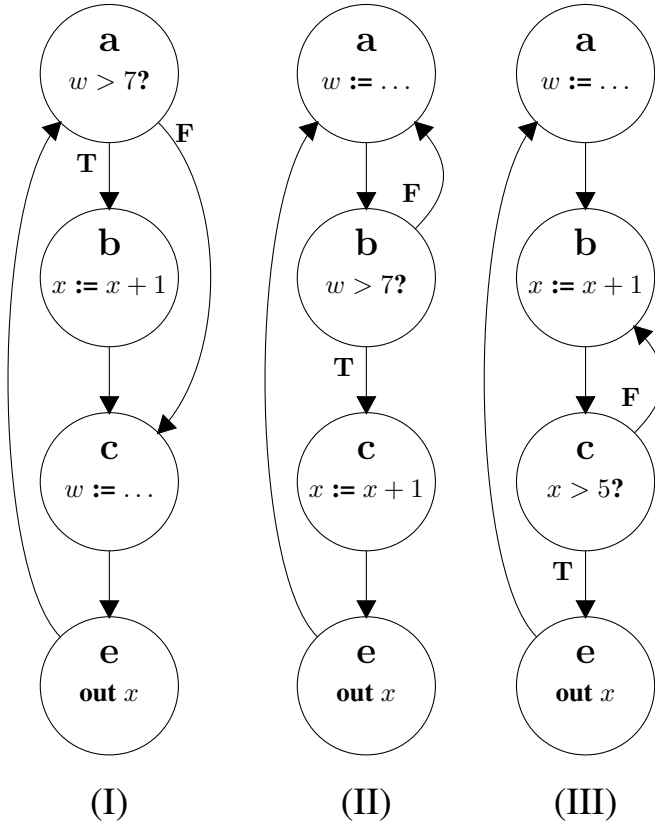


Figure 3. Examples of CFGs (no end node), with $code$ applied to nodes.

b ; b is a successor of a with b reachable from b and all paths from b to e containing b . Since the slice set contains b and e , slicing based on \xrightarrow{wod} will contain also a , as desired.

Now let us examine how Def. 2 works on the CFGs from Fig. 3. In (I), it is easy to check that $a \xrightarrow{wod} b, e$ holds. Since $b \xrightarrow{dd} e$ holds and thus b is in the slice set, we infer that also a is in the slice set. This is as it should be, since otherwise, the observable values produced by the slice would be either $1, 2, 3, \dots$ or $0, 0, 0, \dots$, whereas the original program may produce a sequence like $1, 1, 2, 3, 3, \dots$

In (II), we have $c \xrightarrow{dd} e$. On the other hand, we do not have $b \xrightarrow{wod} c, e$, since there is no path from b to e not containing c . Similarly, $a \not\xrightarrow{wod} c, e$. Therefore, the slice set does not include a or b ; accordingly, $code(a)$ is updated to **skip**, and $code(b)$ is updated to **true?**. The sliced program thus yields observable behavior $1, 2, 3, \dots$, whereas the original program may yield either the same behavior, or a finite *prefix*, like $1, 2, 3, 4$. (To ensure that the sliced program has exactly the same behavior as the original program, we would need also a and b to be in the slice.)

In (III), first note that $b \xrightarrow{dd} e$ holds, and that the original program produces observable behavior $6, 7, 8, \dots$. It is easy

to check that $c \xrightarrow{wod} b, e$ holds, and thus c is in the slice set. This is as it should be, since otherwise $code(c)$ would be updated to **true?**. (If it were to be updated to **false?**, slicing would produce *smaller* observable behavior, contrary to our aims.) But then the observable behavior of the sliced program would be $1, 2, 3, \dots$

In the following, we shall show that to ensure that the observable behavior of the original program is a prefix of the observable behavior of the sliced program, for a slice set S_C which contains the slicing criterion C , it is *sufficient* to demand that S_C is closed under \xrightarrow{dd} and \xrightarrow{wod} . That is, if $n \xrightarrow{dd} m$ and $m \in S_C$, or if $n \xrightarrow{wod} n_a, n_b$ and $n_a, n_b \in S_C$, then also $n \in S_C$.

The key property of weak order dependence is that a slice set closed under \xrightarrow{wod} has at most one “next observable”, as formalized in Def. 3 and Lemma 1. That is, for a given (unobservable) node n , there cannot be two distinct observables m_a and m_b such that there is a path from n to m_a where all nodes but m_a are unobservable, and a path from n to m_b where all nodes but m_b are unobservable. The importance of this property was recognized already in [20] (and less explicitly in [19]).

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 $i \in 1 \dots k - 1$, $n_i \notin S$.

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

Lemma 1 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, in order to arrive at 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 exists a path $\pi = [n_1..n_k]$ with $n_k = n$ such that for all $i \in 1 \dots k - 1$, $n_i \notin S$. Since $obs_S^G(n) = \{n\}$, we infer that there exists $j \in 1 \dots k - 1$ 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 1 and 2 obviously hold; concerning requirement 3, we 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 that $n_j \in S$, yielding the desired contradiction. ■

4 Formalizing Slicing and its Correctness

The technical development in this section is implicitly parameterized wrt. a given CFG G , and a slice set S_C which

contains the slicing criterion C . Slicing is defined only if all nodes have at most one next observable (a sufficient condition for which is that S_C is closed under \xrightarrow{wod} , cf. Lemma 1); the sliced program has the same CFG as the original program but different code maps:

Definition 4 Assume that the original program has code map $code_1$, and assume that for all n in G , $|obs_{S_C}^G(n)| \leq 1$. The result of slicing is a program with CFG G and code map $code_2$ given by

1. if $n \in S_C$, then $code_2(n) = code_1(n)$;
2. if $n \notin S_C$ and n is a statement node, then $code_2(n) = \mathbf{skip}$;
3. 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 there exists m such that $obs_{S_C}^G(n) = \{m\}$, then
 - i. if m is reachable from n_T with $dist^G(n_T, m) < dist^G(n, m)$, then $code_2(n) = \mathbf{true?}$;
 - ii. otherwise, if m is reachable from n_F with $dist^G(n_F, m) < dist^G(n, m)$, then $code_2(n) = \mathbf{false?}$;
 - (b) if $obs_{S_C}^G(n) = \emptyset$, then $code_2(n) = \mathbf{true?}$.

It is easy to see that the clauses 3(a)i and 3(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. Of course, an implementation would probably let $code_2(n)$ be an unconditional jump to m , but we leave this optimization to a post-processing phase so as to keep the core of the correctness proof conceptually simple. In 3b, we could as well have chosen $code_2(n) = \mathbf{false?}$.

Example 1 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

$$\begin{aligned} code_2(a) &= \mathbf{skip} & code_2(b) &= \mathbf{true?} \\ code_2(c) &= x := x + 1 & code_2(e) &= \mathbf{out } x \end{aligned}$$

Note that it would clearly be incorrect to have $code_2(b) = \mathbf{false?}$, as then the sliced program would not produce any observable output.

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 program variables to values. For $i = 1, 2$, we write $i \vdash s \rightarrow s'$ if the code map $code_i$ transforms state s into state s' . The definition is natural: if $code_i(n) = x := E$ then $i \vdash (n, \sigma) \rightarrow (n', \sigma')$ where n' is the successor of n and $\sigma' = \sigma[x \mapsto v]$

with $v = \llbracket E \rrbracket \sigma$ (the value of E in σ); if $code_i(n) = B?$ then $i \vdash (n, \sigma) \rightarrow (n', \sigma)$ where n' is the T -successor of n if $\llbracket B \rrbracket \sigma = \mathbf{true}$, and the F -successor otherwise. The semantics are deterministic: if $i \vdash s \rightarrow s'$ and $i \vdash s \rightarrow s''$ then $s' = s''$.

Using the 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 $i \vdash (n, \sigma) \rightarrow s'$ and $n \in S_C$ (observable move);
- $i \vdash (n, \sigma) \xrightarrow{\tau} s'$ if $i \vdash (n, \sigma) \rightarrow s'$ and $n \notin S_C$ (silent move);
- $i \vdash s \xrightarrow{\tau} s'$ for the reflexive transitive closure of $i \vdash s \xrightarrow{\tau} s'$;
- $i \vdash s \xrightarrow{n} s'$ if there exists s_1 such that $i \vdash s \xrightarrow{\tau} s_1$ and $i \vdash s_1 \xrightarrow{n} s'$.

Note that in $i \vdash s \xrightarrow{n} s'$, we do not allow silent moves *after* the observable move.

Thanks to Def. 4, in particular clauses 3(a)i and 3(a)ii, an easy induction in $dist(n, m)$ yields

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

Definition 5 (Weak Simulation) A binary relation ϕ is a weak simulation if whenever

$$s_1 \phi s_2 \text{ and } 1 \vdash s_1 \xrightarrow{n} s'_1$$

then there exists s'_2 such that

$$s'_1 \phi s'_2 \text{ and } 2 \vdash s_2 \xrightarrow{n} s'_2.$$

That is, if the original program can perform an observable action then so can the sliced program, but not necessarily vice versa. We shall shortly define a relation R and prove it to be a weak simulation. For that purpose, we need the notion of relevant variables:

Definition 6 (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 $i \in 1..k-1$, $v \notin def(n_i)$.

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 [20, Sect. 5, Lemma 7] shows that we can safely ignore the subscripts:

Lemma 3 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 , $rv_1(n) = rv_2(n)$.

Next a couple of auxiliary results.

Lemma 4 Assume that S_C is closed under \xrightarrow{dd} , and that each $obs(n)$ is at most a singleton. For n_a, n_b such that $obs(n_a) = obs(n_b)$ we have $rv(n_a) = rv(n_b)$.

Proof: If $obs(n_a) = obs(n_b) = \emptyset$, no node in S_C is reachable from n_a or from n_b , and therefore $rv(n_a) = rv(n_b) = \emptyset$. Otherwise, there exists a node m such that $obs(n_a) = obs(n_b) = \{m\}$; it will be sufficient (due to symmetry) to prove $rv(n_a) = rv(m)$.

First let $v \in rv(n_a)$ be given; there exists $m_a \in S_C$ with $v \in ref(m_a)$, and a (repetition-free) path π_a from n_a to m_a , such that if n occurs in π_a and $n \neq m_a$ then $v \notin def(n)$. Since $obs(n_a) = \{m\}$, m must occur somewhere in π_a ; we now infer that $v \in rv(m)$.

Next let $v \in rv(m)$ be given; there exists $m_a \in S_C$ with $v \in ref(m_a)$, and a (repetition-free) path π_a from m to m_a , such that if n occurs in π_a and $n \neq m_a$ then $v \notin def(n)$. Since $obs(n_a) = \{m\}$, there exists a (repetition-free) path π from n_a to m such that if n occurs in π and $n \neq m$ then $n \notin S_C$. To establish $v \in rv(n_a)$, it is sufficient to show that if n occurs in π and $n \neq m$ then $v \notin def(n)$. Assume the contrary, then π contains a last such n , but for that n we would have $n \xrightarrow{dd} m_a$ and hence $n \in S_C$, yielding the desired contradiction. ■

Lemma 5 Assume that S_C is closed under \xrightarrow{dd} , and that each $obs(n)$ is at most a singleton. If with $i = 1$ or $i = 2$, $i \vdash (n, \sigma) \rightarrow (n', \sigma')$ with $n \notin S_C$ and with $obs(n') \neq \emptyset$, then for all $v \in rv(n)$ it holds that $\sigma(v) = \sigma'(v)$.

Proof: From $n \notin S_C$ we infer that $obs(n') \subseteq obs(n)$, and from $obs(n') \neq \emptyset$ our assumptions now entail that there exists m such that $obs(n') = obs(n) = \{m\}$. By Lemma 4, $rv(n) = rv(n')$. Now let $v \in rv(n)$; since $n \notin S_C$ we infer that $v \notin def(n)$. But this shows that $\sigma(v) = \sigma'(v)$. ■

We are now ready to define a relation R which, as stated in Theorem 8, is a weak simulation if S_C is closed under \xrightarrow{dd} and \xrightarrow{wod} .

Definition 7 We define $(n_1, \sigma_1) R (n_2, \sigma_2)$ to hold iff

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

Note that by Lemma 4, $rv(n_2) = rv(n_1)$.

Example 2 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 naturally split into two subparts: Lemma 7 talks about the situation where the original program makes an observable move, in which case also the sliced program must make an observable move (perhaps preceded by some silent moves); Lemma 6 talks about the situation where the original program makes a silent move, in which case the sliced program does not need to move.

Example 3 In Examples 1 and 2, assume that $(n_1, \sigma_1) R (n_2, \sigma_2)$ and $1 \vdash (n_1, \sigma_1) \xrightarrow{\tau} (n'_1, \sigma'_1)$ with $n_1 = b$. Then $\sigma'_1 = \sigma_1$, and either $n'_1 = a$ or $n'_1 = c$; also, $n_2 \in \{a, b, c\}$. But then also $(n'_1, \sigma'_1) R (n_2, \sigma_2)$.

In general, we have:

Lemma 6 Assume that S_C is closed under \xrightarrow{dd} and \xrightarrow{wod} . Whenever

$$s_1 R s_2 \text{ and } 1 \vdash s_1 \xrightarrow{\tau} s'_1$$

where with $s'_1 = (n'_1, \sigma'_1)$, $obs(n'_1) \neq \emptyset$, 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 that $obs(n_1) = \{n\}$. From $s_1 R s_2$ we see that $obs(n_2) = \{n\}$, and by Lemma 4 we have $rv(n_1) = rv(n_2) = rv(n'_1) = rv(n)$. For all $v \in rv(n)$ we have $\sigma_1(v) = \sigma_2(v)$, since $s_1 R s_2$, and even $\sigma'_1(v) = \sigma_2(v)$, by Lemma 5. This establishes $s'_1 R s_2$. ■

Example 4 In Examples 1 and 2, assume that $(c, \sigma_1) R (b, \sigma_2)$ and $1 \vdash (c, \sigma_1) \xrightarrow{c} (e, \sigma'_1)$. Thus $\sigma'_1 = \sigma_1[x \mapsto \sigma_1(x) + 1]$ and $\sigma_2(x) = \sigma_1(x)$. With $\sigma'_2 = \sigma_2[x \mapsto \sigma_2(x) + 1]$ we thus have $\sigma'_2(x) = \sigma'_1(x)$ and hence $(e, \sigma'_1) R (e, \sigma'_2)$; we also have $2 \vdash (b, \sigma_2) \xrightarrow{c} (e, \sigma'_2)$ since $code_2(b) = true?$.

This is an instance of a general result:

Lemma 7 Assume that S_C is closed under \xrightarrow{dd} and \xrightarrow{wod} . Whenever

$$s_1 R s_2 \text{ and } 1 \vdash s_1 \xrightarrow{n} s'_1$$

then there exists s'_2 such that

$$s'_1 R s'_2 \text{ and } 2 \vdash s_2 \xrightarrow{n} s'_2.$$

Proof: Let $s_i = (n_i, \sigma_i)$ ($i = 1, 2$); from $1 \vdash (n_1, \sigma_1) \xrightarrow{n} s'_1$ 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 by Lemma 2 hence $2 \vdash (n_2, \sigma_2) \xrightarrow{\tau} (n, \sigma_2)$. We also infer that

$$\forall v \in rv(n) : \sigma_1(v) = \sigma_2(v) \quad (1)$$

and in particular $\sigma_1(v) = \sigma_2(v)$ for all $v \in ref(n)$. Therefore the sliced program agrees with the original program on

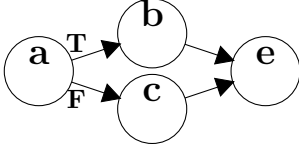


Figure 4. Motivating $n_e \in S$ in Prop. 10.

the outcome of any test in n (if n is a predicate node), so with $s'_1 = (n', \sigma'_1)$ there exists σ'_2 such that $2 \vdash (n, \sigma_2) \xrightarrow{n} (n', \sigma'_2)$, and hence $2 \vdash (n_2, \sigma_2) \xrightarrow{n} (n', \sigma'_2)$. Our only remaining proof obligation is to show that, given $v \in rv(n')$, it holds that $\sigma'_1(v) = \sigma'_2(v)$. Two cases:

- if $v \in \text{def}(n)$, then there exists E such that $\text{code}_1(n) = \text{code}_2(n) = (v := E)$; thus $\sigma'_1(v) = \llbracket E \rrbracket \sigma_1$ and $\sigma'_2(v) = \llbracket E \rrbracket \sigma_2$. From (1) we infer that $\llbracket E \rrbracket \sigma_1 = \llbracket E \rrbracket \sigma_2$ and thus $\sigma'_1(v) = \sigma'_2(v)$.
- if $v \notin \text{def}(n)$, then $v \in rv(n)$, and the claim again follows from (1) since $\sigma'_1(v) = \sigma_1(v) = \sigma_2(v) = \sigma'_2(v)$.

Finally, we are in position to establish the desired correctness result:

Theorem 8 *Assume that S_C is closed under \xrightarrow{dd} and \xrightarrow{wod} . Then R , as defined in Def. 7, is a weak simulation, cf. Def. 5.*

Proof: Our assumption is that $s_1 R s_2$ and $1 \vdash s_1 \xrightarrow{n} s'_1$, that is there exists $s_1^1 \dots s_1^k$ ($k \geq 1$) with $s_1^1 = s_1$ such that $1 \vdash s_1^k \xrightarrow{n} s'_1$, and $1 \vdash s_1^i \xrightarrow{\tau} s_1^{i+1}$ for all $i \in 1 \dots k-1$. With $s_1^i = (n_1^i, -)$ we have $\text{obs}(n_1^i) = \{n\}$ for all $i \in 1 \dots k$, so we can repeatedly apply Lemma 6 to infer that $s_1^k R s_2$. By Lemma 7 we from $1 \vdash s_1^k \xrightarrow{n} s'_1$ now infer that there exists s'_2 such that $s'_1 R s'_2$ and $2 \vdash s_2 \xrightarrow{n} s'_2$. ■

5 Conservative Extension

In this section, we consider the case where the CFG has a unique end node n_e (reachable from all other nodes). We shall show that if a node set S_C is closed under \xrightarrow{cd} then S_C is also closed under \xrightarrow{wod} (Proposition 13), and that if S_C is closed under \xrightarrow{wod} and contains n_e then S_C is also closed under \xrightarrow{cd} (Proposition 10). For an example showing why $n_e \in S_C$ is needed for the latter result, consider the CFG in Fig. 4 with $S_C = \{c\}$ which is trivially closed under \xrightarrow{wod} , but not under \xrightarrow{cd} since $a \xrightarrow{cd} c$. Note that as predicted by our theory, the original program has an observable behavior (either c or the empty sequence) which is a prefix of the observable behavior (c) of the program (which has $\text{code}_2(a) = \text{false?}$) resulting from slicing wrt. \xrightarrow{wod} .

For both propositions, some auxiliary results are needed.

Lemma 9 *Assume that G has a unique end node n_e . If $n \xrightarrow{cd} m$ with $m \notin \{n, n_e\}$ then $n \xrightarrow{wod} m, n_e$.*

Proof: from the definition of \xrightarrow{cd} , we know that

- m does not strictly postdominate n , which since $m \neq n$ implies that m does not postdominate n ; hence there is a path from n to n_e not containing m .
- there is a path from n to m , and since $m \neq n_e$ and n_e has no outgoing edges, this path does not contain n_e .
- n has a successor n_1 such that m postdominates n_1 , that is, all paths from n_1 to n_e contain m ; in particular, since n_e is reachable from all nodes, m is reachable from n_1 .

But this establishes $n \xrightarrow{wod} m, n_e$. ■

Proposition 10 *Assume that G has a unique end node n_e . If S contains n_e and is closed under \xrightarrow{wod} , then S is also closed under \xrightarrow{cd} .*

Proof: Assuming that $n \xrightarrow{cd} m$ with $m \in S$ and $m \neq n$, we must prove $n \in S$. Also, $m \neq n_e$ (since otherwise, we from $n \xrightarrow{cd} n_e$ would have that n_e does not strictly postdominate n which since n_e postdominates all nodes implies $n = n_e$ which is a contradiction since n_e has no outgoing edges). We can now apply Lemma 9 to infer that $n \xrightarrow{wod} m, n_e$. But now $n \in S$ follows from our assumptions (S contains m, n_e and is closed under \xrightarrow{wod}). ■

Lemma 11 *Assume that $n \xrightarrow{wod} n_a, n_b$. Then n is a predicate node. Also, there exists repetition-free paths π_a from n to n_a , and π_b from n to n_b , such that π_a does not contain n_b , π_b does not contain n_a , and the second element of π_a is different from the second element of π_b .*

Proof: First note that $n \neq n_a \neq n_b$. From the definition of \xrightarrow{wod} , we see that there exists a path π'_a from n to n_a not containing n_b , and a path π'_b from n to n_b not containing n_a ; we can clearly assume that π'_a and π'_b are repetition-free. We also see that n has a successor n' where, wlog., we can assume that n_b is reachable from n' and that all paths from n' to n_a contain n_b . We infer that n' does not occur in π'_a , showing that n has two successors and thus is a predicate node, and that there is a repetition-free path π' from n' to n_b such that π' does not contain n_a ; also, π' does not contain n (since otherwise there would be a path from n' to n_a not containing n_b). Hence π' allows us to construct a repetition-free path π_b , with n' as second element, from n to n_b , where π_b does not contain n_a . This yields the claim, with $\pi_a = \pi'_a$. ■

Lemma 12 Assume that $n \xrightarrow{wod} n_a, n_b$. Then there exists repetition-free paths π_a from n to n_a , and π_b from n to n_b , such that if m belongs to both π_a and π_b then $m = n$.

Proof: Lemma 11 gives us repetition-free paths π_a from n to n_a not containing n_b , and π_b from n to n_b not containing n_a ; also, $m_a \neq m_b$ where m_a (one successor of n) is the second element of π_a , and m_b (the other successor of n) is the second element of π_b . From the definition of \xrightarrow{wod} , n has a successor n' such that either all paths from n' to n_b contain n_a , or all paths from n' to n_a contain n_b ; wlog., we can assume that $n' = m_a$, and infer that

$$\text{all paths from } m_a \text{ to } n_b \text{ contain } n_a. \quad (2)$$

Now assume, to get a contradiction, that m belongs to π_a and to π_b (so $m \neq n_a$) but $m \neq n$. Thus π_a contains a subpath from m_a to m not containing n_a , and π_b contains a subpath from m to n_b not containing n_a . The path formed by concatenating those subpaths will violate (2), yielding the desired contradiction. ■

Proposition 13 Assume that G has a unique end node n_e . If S is closed under \xrightarrow{cd} , then S is also closed under \xrightarrow{wod} .

Proof: Assuming that $n \xrightarrow{wod} n_a, n_b$ with $n_a, n_b \in S$, we must show $n \in S$. Lemma 12 tells us that there exists repetition-free paths π_a from n to n_a , and π_b from n to n_b , such that if m belongs to both π_a and π_b then $m = n$. At least one of those paths must have the property that there exists a path from one of its nodes to n_e that does not intersect with the other path; wlog., we can assume that π_a has that property. Let π_b be of the form $[m_1..m_k]$, with $m_1 = n$ and $m_k = n_b$; we infer that for $i \in 2..k$, m_i does not postdominate n . Since $m_k \in S$ and our goal is to prove that $m_1 \in S$, it will suffice to prove

$$\forall i \in 1..k : \text{if } m_i \in S \text{ then } m_1 \in S$$

and we shall do so by induction in i , where the base case ($i = 1$) is trivial. For the inductive case, assuming that $m_i \in S$ with $i > 1$, we must prove $m_1 \in S$. We saw that m_i does not postdominate m_1 but since m_i postdominates itself, there exists i_0 with $1 \leq i_0 < i$ such that m_i does not postdominate m_{i_0} , but m_i postdominates m_j for all j with $i_0 < j \leq i$. Hence $m_{i_0} \xrightarrow{cd} m_i$, so since S is closed under \xrightarrow{cd} we have $m_{i_0} \in S$. But the induction hypothesis, applied to i_0 , now tells us that $m_1 \in S$, as desired. ■

6 Discussion

We have provided the theoretical foundation, culminating in a crisp correctness proof, for an approach to intraprocedural slicing which is able to handle arbitrary control flow

graphs, even those without end nodes. The correctness criterion allows loops that do not influence relevant variables to be sliced away, resulting in more manageable slices. On the other hand, if certain *liveness* properties must hold, as demonstrated by Hatcliff et al. [11], a more conservative approach is needed, as established by Ranganath et al. [20] which demands the slices to contain *all* loops. A more flexible approach would be to annotate loops depending on whether they are known to terminate or not, as does the approach of Chen & Rosu [4] which presents a parametric approach to control dependence, able to use the standard definition for a loop which is known to terminate, and weak control dependence otherwise.

A key ingredient of our approach is the notion of “weak order dependence”, first introduced in [20], which turns out to generalize classical notions of control dependence in that it—unlike them—deals properly even with graphs that have no end nodes. In [20, Sect. 6], algorithms for computing several kinds of dependencies are given, including decisive order dependence (\xrightarrow{dod}); we expect that an algorithm for computing weak order dependence can be constructed along similar lines but leave that to future work.

Future work also includes extending the approach to an interprocedural [13, 22, 1] or even multi-threaded [14] setting. Also, inspired by the bridge-building work of Hammer et al. [8], we may investigate how our framework relates to information flow analysis.

The development in this paper models backwards static slicing; the relationship between various kinds of slicing has been formalized by Binkley et al [3] (whose correctness criterion considers only finite trajectories). Our correctness criterion is expressed using simulation; it is easy to see that simulation entails the standard approach by Ball & Horwitz [2] which demands that at each node in the slicing criterion, the sequence of observed values (values of the variables referenced) for the original program and the sequence of observed values for the sliced program are identical, or—in the case where the termination properties are different—one is a prefix of the other. Note that simulation, while arguably more elegant as it can be 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. In future work, we hope to give (perhaps inspired by Godefroid’s work [7]) a variant of simulation which permits independent observables to be swapped. This would help to generate even smaller slices if our work were to be extended to a trace-based setting; if n_a is a conditional deciding whether to execute the trace $n_a n_b n_c$ or the trace $n_a n_c n_b$ but neither of n_b, n_c is data dependent on the other, a slice set including n_b, n_c would not need to include also n_a (cf. the proposal in [20, Def. 15]).

The correctness of slicing has been addressed in several

other works. An early correctness result, by Horwitz et al., is [12] which uses a multi-layered approach to reason about slicing in the realm of data dependence: starting with a standard semantics, they develop an instrumented semantics which labels each location with the program point that last defined its value; then they derive a collecting semantics and finally an abstract semantics. The correctness result of Ball & Horwitz [2], referred to above, considers control flow graphs that are arbitrary (could be irreducible) except that they are assumed to have the unique end node property. Also Hatcliff et al. [11] assume that control flow graphs have the unique end node property; the authors provide a very detailed correctness proof but only consider the case where *both* executions terminate. For how to deal with non-termination, [11] refers to [10] which proposes a correctness property based on a variant of bisimulation but does not work out the details—this was later done (in a sequential setting) in [20]. A main contribution of this paper is to argue that for practical applications of slicing, an asymmetric notion of bisimulation is needed.

Acknowledgments

The author, who is supported in part by grants from AFOSR and from Rockwell Collins, would like to thank John Hatcliff and Venkatesh Prasad Ranganath for many interesting discussions, comments on a draft of this paper, and general encouragement, and to thank some of the anonymous referees from SCAM 2007 for useful suggestions.

References

- [1] M. Allen and S. Horwitz. Slicing Java programs that throw and catch exceptions. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '03)*, pages 44–54. ACM, June 2003.
- [2] T. Ball and S. Horwitz. Slicing programs with arbitrary control flow. In *1st International Workshop on Automated and Algorithmic Debugging*, volume 749 of *LNCS*. Springer-Verlag, May 1993.
- [3] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, Oct. 2006.
- [4] F. Chen and G. Rosu. Parametric and termination-sensitive control dependence. In *SAS'06*, 2006.
- [5] 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, 1987.
- [6] M. A. Francel and S. Rugaber. The relationship of slicing and debugging to program understanding. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99)*, pages 106–113, 1999.
- [7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, Nov. 1994.
- [8] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *Second International Symposium on Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2006)*, Paphos, Cyprus, pages 136–145, Nov. 2006.
- [9] M. Harman, A. Lakhotia, and D. Binkley. Theory and algorithms for slicing unstructured programs. *Information and Software Technology*, 48(7):549–565, July 2006.
- [10] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *6th International Static Analysis Symposium (SAS'99)*, volume 1694 of *LNCS*, pages 1–18. Springer-Verlag, Sept. 1999.
- [11] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000. Special issue containing selected papers from the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.
- [12] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *PLDI'89 (Programming Language Design and Implementation)*, pages 28–40, 1989.
- [13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
- [14] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Montreal, Canada, pages 35–42, June 1998.
- [15] L. I. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):343–349, Mar. 2000.
- [16] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [17] A. Podgurski and 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*, pages 168–178, 1989.
- [18] 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, Sept. 1990.
- [19] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In M. Sagiv, editor, *ESOP 2005, Edinburgh*, volume 3444 of *LNCS*, pages 77–93. Springer-Verlag, Apr. 2005.
- [20] 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*, 2007. To appear.
- [21] SAnToS Laboratory. Indus, a toolkit to customize and adapt Java programs. This software is available at <http://indus.projects.cis.ksu.edu>.

- [22] J. A. Stafford. *A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis*. PhD thesis, University of Colorado, 2000.
- [23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.