Semantic Correctness of Dependence-Based Slicing for Interprocedural, Possibly Nonterminating Programs

ABU NASER MASUD, Mälardalen University, Sweden
BJÖRN LISPER, Mälardalen University, Sweden

Existing proofs of correctness for dependence-based slicing methods are limited either to the slicing of intraprocedural programs [2, 39], or the proof is only applicable to a specific slicing method [4, 41]. We contribute a general proof of correctness for dependence-based slicing methods such as Weiser [50, 51], or Binkley et al. [7, 8], for interprocedural, possibly nonterminating programs. The proof uses well-formed weak and strong control closure relations, which are the interprocedural extensions of the generalised weak/strong control closure provided by Danicic et al. [13], capturing various nonterminating insensitive and nontermination sensitive control dependence relations that have been proposed in the literature. Thus, our proof framework is valid for a whole range of existing control dependence relations.

We have provided a definition of semantically correct (SC) slice. We prove that SC slices agree with Weiser slicing, that deterministic SC slices preserve termination, and that nondeterministic SC slices preserve the nondeterministic behavior of the original programs.

CCS Concepts: • **Theory of computation** → **Operational semantics**; **Program analysis**; • **Software and its engineering** → *Maintaining software*;

Additional Key Words and Phrases: Correctness of slicing, static slicing, semi-equivalence, simulation, bisimulation, nontermination, nondeterminism, interprocedural program

ACM Reference Format:
Abu Naser Masud and Björn Lisper. 2020. Semantic Correctness of Dependence-Based Slicing for Interprocedural, Possibly Nonterminating Programs. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (2020), 55 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Program slicing is a program transformation technique that selects the parts of a program affecting certain computations, usually specified by a *slicing criterion*, of the program. Since the seminal work of program slicing by Weiser [50], there have been many contributions on different aspects and applications of program slicing.

Slicing was originally considered for deterministic, terminating programs. Today it is being used for a variety of software and systems, e.g., programs with modern language constructs such as exception handling [1, 4], nonterminating reactive systems such as web services [11, 29], distributed real-time systems [2, 38], or software architecture models [23]. It is often desirable that the slice maintains the nontermination property of reactive systems, or the possible nondeterminism in models.

Authors' addresses: Abu Naser Masud, School of Innovation, Design and Engineering, Mälardalen University, Sweden, masud.abunaser@mdh.se; Björn Lisper, School of Innovation, Design and Engineering, Mälardalen University, Sweden, bjorn.lisper@mdh.se.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
0164-0925/2020/0-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

There are many kinds of slicing [43]. Given a *slicing criterion*, i.e., a set of pairs of program points and program variables, they aim to compute a subset of the program consisting of the statements that can possibly affect the values of the variables in the criterion (backward slicing), or the statements that might be affected by these values (forward slicing). Here we consider *static backward* slicing.

Methods for this kind of slicing are often *dependence-based*. Commonly they work on the control-flow graph (CFG) of the program. A so-called *program dependency graph* (PDG) [16] is formed on top of the CFG as the union of a *data dependency relation*, connecting CFG nodes where values computed by the first node might be used by the second node, and a *control dependency relation* between (typically) conditional nodes, and the nodes whose execution might be decided by the outcome of the conditional. The slice is then computed as the set of nodes reachable in the PDG, from the nodes in the slicing criterion, in the forward or backward direction [20].

Early work on program slicing assumed that the CFG is deterministic, has a unique end node, and that there is a path from each node to the end node. Under these assumptions, the control dependencies are accurately captured by the *standard control dependency* relation. However, nonterminating or non-deterministic programs can have CFG's with multiple end nodes, no end node, or nodes with no path to the end node. Various control dependency relations, extending the standard control dependency relation to deal with such CFG's, have been proposed [39]. Danicic et al. [13] presented two generalisations of control dependence, called weak and strong control closure, which guarantee that the resulting slices are *non-termination insensitive* or *non-termination sensitive*, respectively. A non-termination sensitive slice always preserves any possible nontermination of the original program, whereas a non-termination insensitive slice might fail to do this. All previously defined control dependence relations are specialisations of these weak/strong control closure relations. However, weak/strong control closed relations by Danicic et al. are defined for directed graphs that can only represent CFG's of intraprocedural programs. In this article we have extended them to interprocedural programs, and we have provided correctness theorems and proofs of dependence-based slicing in which the control dependence relation is captured by the weak or strong control closure relation, respectively.

The theoretical basis for the correctness criteria of any slicing approach should be some appropriate program slicing semantics. Weiser's finite trajectory semantics [50] informally states that the slice P' of any program P with respect to some slicing criterion is an executable program where P' is obtained from P (by deleting zero or more statements), it computes the same sequence of values for the variables specified by the slicing criterion, and whenever P halts on an input state then P' also halts on that state. Reps and Yang [41] defined the semantics according to Weiser with two additions: (i) P' and P compute the same sequence of values at each program point of P' (note that Weiser considered only the program point specified in slicing criteria), and (ii) a program terminates on some states if it can be decomposed [1] into multiple slices such that all the slices terminate on those states. However, this semantic model is incomplete for slicing non-terminating systems (e.g., reactive systems, or software models used in model checking) as the slice of a non-terminating program may slice out infinite loops and the semantic relationship between the program and its slice is unspecified, i.e., the execution behavior of the slice is not defined for those states where the original program does not terminate. Various attempts have been made to provide the slice semantics for both the terminating and nonterminating programs such as the non-standard semantics of Danicic et al. [14], the lazy program slicing semantics of Cartwright and Felleisen [10], the transfinite semantics of Giacobazzi and Mastroeni [17] and Nestra [32], and

[1]Reps and Yang never explained the term "decomposition of a program". However, we infer the meaning of decomposition by looking into their termination theorem as follows: Program P is decomposed into programs Q and R if the PDF of P can be obtained by combining the PDGs of Q and R.

the finite trajectory semantics of Barraclough et al. [5]. Ward and Zedan [48] illustrated that these semantics do not agree with the original semantics of Weiser in case of nonterminating programs, and allow a nonterminating program as a valid slice of a terminating program. They provided a slice semantics for nonterminating programs that is nontermination insensitive in the sense that a nonterminating loop in the original code may not be preserved in the sliced code. In this article, we have provided a slice semantics for possibly nonterminating programs for both nontermination insensitive and -sensitive slices (Section 3.3). Our semantics agree with that of Weiser's when the slice is nontermination insensitive but not nontermination sensitive (Section 3.4.1), a deterministic nontermination insensitive slice according to our semantics preserves termination (Section 3.4.2), and the semantics allows slicing of nondeterministic programs such that the nondeterministic behavior of the original code is preserved in the slice (Section 3.4.3).

The semantics of slicing describes the semantic relationship between a program and its slice. However, we need a proof of correctness that the slicing approach under consideration produces correct slices according to our slice semantics. The theoretical foundations for the correctness of slicing developed earlier considered finite execution traces. For instance, Binkley et al. [7] formalizes the relationship between forms of program slicing, and their correctness arguments are based on finite trajectories. Some slicing correctness arguments do not specify if the nontermination property is preserved by the slicing algorithms. For instance, Reps and Yang [41] provided a termination relation between the original program and its slices. However, it did not tell anything about nontermination of the sliced code. Recent foundational work on the correctness of slicing [2, 38, 39] for modern program structures are based on establishing a simulation and/or bisimulation relation between the original and the sliced program. The proof principle for the correctness of slicing based on such relations works for programs having infinite execution traces, and is thus able to prove the nontermination property. However, the proof of correctness developed in these works is limited to intraprocedural programs. Here, we provide the correctness arguments and proof techniques for dependency-based slicing methods targeting interprocedural, possibly nonterminating programs. In particular, we have contributed the following:

(1) We provide a definition of *semi-equivalence* that describes the semantic relationship between an interprocedural, possibly nonterminating program, and its slice. Instead of using a trace-based denotational semantics, an operational semantics defined on the CFG level is used to define the semi-equivalence. Thus, we take into account infinite traces originating from nonterminating programs and characterize the relationship between the original program and its correct slice.
(2) We provide the *correctness condition* theorem (Theorem 6.6) specifying the correctness criteria for dependence-based slicing methods. The theorem states that any correct slice obtained from dependence-based slicing produces either a *bisimulation*, or a *simulation* relation with the original program. We obtain the bisimulation relation when nontermination is preserved; otherwise it is a simulation relation. These relations are constructed for interprocedural programs that obey the semi-equivalence semantics. We provide a detailed proof of the theorem in Section 7.
(3) We have defined well-formed weak and strong control closure relations, the nontrivial interprocedural extension of the weak and strong control closure relations provided by Danicic et al. [13] , which are nontermination insensitive and nontermination sensitive.
(4) We have discussed and proved different properties of the semi-equivalence semantics (in Section 3.4). In particular, the semi-equivalence semantics agree with the Weiser slicing, all semi-equivalent deterministic slices preserve the termination property and all semi-equivalent nondeterministic slices preserve the nondeterministic behavior of the original programs.

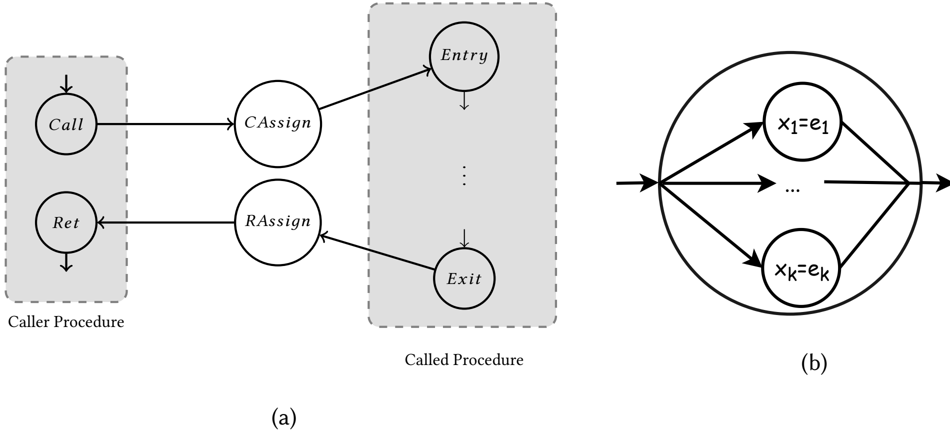

Fig. 1. (a) Control flow among interprocedural CFG nodes, (b) simultaneous data flow among the assignments in a *CAssign* or *RAssign* node

2 PRELIMINARIES

2.1 Control Flow Graph and Related Concepts

All proof arguments, and analyses in this paper are based on the *Control-Flow Graph* (CFG) representation of interprocedural programs. We define an interprocedural CFG as follows:

Definition 2.1 (Control Flow Graph). A *Control-Flow Graph* (CFG) $(N, E, N_e, start)$ is a directed graph where

(1) N is the set of nodes of the following types:
- *Start*: the unique node $start \in N$ represents the beginning of the execution,
- *End*: the node in the set N_e, which is at most a singleton set, represents the normal termination of the execution,
- *Skip*: "no-op" instructions.
- *Assign*: simple assignments $x = e$ where x is a program variable, and e is an expression over program variables,
- *Cond*: boolean conditions,
- *Entry*, *Exit*: procedure entry and exit, respectively,
- *Call*, *Ret*: procedure calls, and returns from called procedures,
- *CAssign*, *RAssign*: the assignments to formal input parameters $x_1 = e_1, \ldots, x_k = e_k$, and actual output parameters $x'_1 = e'_1, \ldots, x'_l = e'_l$ in which e_i and e'_i are expressions over the actual input and formal output parameters, respectively,

(2) $E \subseteq N \times N$ is the relation describing the possible flow of execution in the graph. An *End* node has no successors, and an *Start* node has no predecessors. An *Exit* node may have multiple successors. A *Cond* node n has at most one *true* successor $tsucc(n)$ and at most one *false* successor $fsucc(n)$. In all other cases, the nodes have at most one successor.

Interprocedural control flow is illustrated in Fig. 1(a). A concrete example of procedure calls from multiple locations is illustrated in Fig. 4 on page 11, where procedure add is called from two different locations L_1 and L_2. Each call to procedure add has distinguished *CAssign* and *RAssign* nodes clearly separating contexts for different calls with assignments between formal and actual parameters.

For the brevity and completeness of representation, we allow a single *CAssign* (or *RAssign*) node to include multiple simultaneous assignments representing all the assignments from actual

input parameters (resp. formal output parameters) to formal input parameters (resp. actual output parameters) related to a procedure call. This representation should affect neither the correctness nor the precision of any slicing algorithm. Any such node can be considered to include multiple nodes each representing a single assignment and have simultaneous data flow as illustrated in Fig. 1(b). System dependency graph (SDG) based slicing methods [20] can create SDGs by considering all direct dependencies to/from each of the inner nodes of the *CAssign* or *RAssign* nodes having single assignments. Slicing methods based on solving data flow equations [26, 49] can use a special equation to handle the simultaneous data flow in the *CAssign* or *RAssign* nodes [26]. This representation thus allows slicing relevant parameters (see Def. 4.10 in Section 4.5).

If the set of End nodes N_e is empty, then the CFG has no terminating execution. Like Danicic et al. [13], we allow Cond nodes having no *true* or *false* successors, or other types of nodes having no successors. If the execution reaches such a node, the program is silently nonterminating. However, if the execution reaches an End node in N_e, then it is a terminating execution. We shall sometimes denote a CFG by the tuple (N, E) instead of $(N, E, N_e, start)$. We consider C-like interprocedural code that can be represented by a CFG according to Def. 2.1. For simplicity we only consider simple assignments of statically known program variables. We now introduce some standard concepts and notation.

- Given any program P and its CFG G, we assume a function *code* that maps G to P, i.e., $P = code(G)$. We shall sometimes abuse this notation and write $code(n)$ for the program statement that the CFG node $n \in N$ represents. If the same CFG G is used to represent an original program P_1 and its slice P_2, we use distinguished mapping functions $code_1$ and $code_2$ such that $P_1 = code_1(G)$ and $P_2 = code_2(G)$.
- The *label* of the statement $code(n)$ is given by the function $\ell(n)$ which, with one exception, uniquely identifies the statement (e.g., $\ell(n)$ can be a unique numeric identifier). The exception is that a *Call* node n_1 and its matching *Ret* node n_2 that originate from a single procedure call instruction will have the same label (i.e. $\ell(n_1) = \ell(n_2)$).
- $n : T$ states that the CFG node n has the type T, with T according to Definition 2.1.
- Any CFG node n is associated with two sets of program variables $def(n)$, and $ref(n)$, containing the variables that are *defined*, and *referenced*, respectively, in the $code(n)$ statement. For a node n of type *Assign*, *CAssign*, or *RAssign*, $def(n)$ and $ref(n)$ may be nonempty. If n is a *Cond* node, then $ref(n)$ may be nonempty but $def(n)$ has to be empty. For all other kinds of nodes n holds that $def(n)$ and $ref(n)$ are empty.
- The set of successor and predecessor nodes of any node n is denoted by $succ(n)$ and $pred(n)$. It is sometimes convenient to attach the edge labels $\{T\}$, $\{F\}$, or $\{T, F\}$ to the edge (n, n') from any *Cond* node n when $n' = tsucc(n)$, $n' = fsucc(n)$, or $n' = tsucc(n) = fsucc(n)$ respectively.
- A *final* CFG node is either a Cond node not having both a true and a false successor, or a non-Cond node having no successor.

2.2 CFG paths and walks

Definition 2.2 (CFG path). A finite CFG path is a sequence of nodes $n_1, n_2, \ldots, n_k$ denoted by $[n_1..n_k]$ such that $n_{i+1} \in succ(n_i)$ for all $1 \le i \le k-1$ and $k \ge 1$. An infinite path $n_1, n_2, \ldots$ is denoted by $[n_1..]$.

A path is *non-trivial* if it contains at least two nodes. A maximal path is either an infinite path $[n_1..]$ or a finite path $[n_1..n_k]$ such that n_k is a final node. The minimum distance from n_i to n_j over all paths $[n_i..n_j]$ is denoted by $dist^G(n_i, n_j)$. Sometimes we write $dist(n_i, n_j)$ instead of $dist^G(n_i, n_j)$ if G is understood from the context. We write $n \in [n_1..n_k]$ when n is some node n_i in this path.

Paths are sequences of CFG nodes. For sequences π, π' in general, $\pi.\pi'$ denotes their concatenation. ϵ is the empty sequence.

A matching pair of *Call* and *Ret* nodes are those nodes that have the same label. Thus, a matching *Call* and *Ret* nodes originate at the same call site. Similarly, a matching *CAssign* and *RAssign* nodes have a predecessor and a successor node, respectively, that have the same label. A *valid path* is a path that can be extended to a sequence of nodes that is balanced with respect to matching *Call* and *Ret* nodes, and *CAssign* and *RAssign* nodes, by either prepending the proper *Call* and *CAssign* nodes or appending the proper *Ret* and *RAssign* nodes. More formally, consider a CFG G with l call sites where, for $1 \leq i \leq l$, (c_i, r_i) is the matching pair of *Call* and *Ret* nodes, and (ca_i, ra_i) is the matching pair of *CAssign* and *RAssign* nodes for call site i. A sequence of *Call*, *CAssign*, *RAssign*, and *Ret* nodes is *balanced* when it is generated by the following context-free grammar (for $1 \leq i \leq l$):

$$\begin{array}{lcl} S & \rightarrow & A \mid ca_i\ A\ ra_i \\ A & \rightarrow & c_i\ ca_i\ A\ ra_i\ r_i \mid A\ A \mid \epsilon \end{array}$$

Here S is the start symbol, A is the nonterminal symbol, and c_i, r_i, ca_i, ra_i are the terminal symbols of the grammar. For example, the sequences $c_1\ ca_1\ c_2\ ca_2\ ra_2\ r_2\ ra_1\ r_1\ c_3\ ca_3\ ra_3\ r_3$ and $c_1\ ca_1\ c_2\ ca_2\ ra_2\ r_2\ ra_1\ r_1$ are balanced as it can be generated from the above grammar. The sequence $c_1\ ca_1\ c_2\ ca_2\ ra_1\ r_1$ is not balanced as it is not generated by the aforementioned grammar.

A path in a CFG is similarly called balanced if its maximal subsequence of *Call*, *CAssign*, *RAssign*, and *Ret* nodes is balanced. An unbalanced path π may contain *isolated Call* or *CAssign* nodes, for which no matching *Ret* or *RAssign* nodes exist in π, or *isolated Ret* or *RAssign* nodes for which no matching *Call* or *CAssign* nodes exist. Paths with isolated *Call* and *CAssign* nodes can appear during program execution, where procedures are called but not yet exited. Similarly, during a backward traversal of the CFG (say, during a backwards data flow analysis) a path may contain isolated *Ret* or *RAssign* nodes.

Now consider a path $\pi = [n_1..n_t]$ whose maximal subsequence of isolated *Call* and *CAssign* nodes is $m_k, \ldots, m_1$, and consider the sequence of *RAssign* and *Ret* nodes $m^1, \ldots, m^k$ where each m^i matches m_i. Let $[n_t..m^1]$ and $[m^j..m^{j+1}]$ be finite paths for all $1 \leq j < k$. Then the *R-extension* of π is the sequence $\pi.[m^1, \ldots, m^k]$. Similarly, if the maximal subsequence in π of isolated *RAssign* and *Ret* nodes is $m^k, \ldots, m^1$, the sequence of matching *Call* and *CAssign* nodes $[m_1, \ldots, m_k]$ is such that there exist finite paths $[m_j..m_{j+1}]$ for all $1 \leq j < k$, and there is a finite path $[m_k..n_1]$, then $[m_1, \ldots, m_k].\pi$ is the *C-extension* of π.

Definition 2.3 (Valid Path). A path in a CFG is *valid* if the path itself, its *C-extension*, or *R-extension* is balanced.

Consider the path $[n_{13}..n_6]$ in the CFG G in Fig. 4. It is a valid path since its *C-extension* $[n_3, n_4].[n_{13}..n_6]$ is balanced. An interprocedural analysis that considers only valid paths is usually more precise since it excludes some kinds of infeasible paths.

Definition 2.4 (Element). Let G be a CFG. An *element* is a pair (n, l) such that n is a CFG node, $l = T$ or $l = F$ if n is a Cond node, and $l = \epsilon$ otherwise.

Suppose *code* is any mapping function of the CFG G, Then, $l = T$ and $l = F$ represent true and false evaluation of the condition $code(n)$, respectively. We use the following definition of *walk* adapted from [13] in order to form logical judgments on CFG paths.

Definition 2.5 (Walk). A *walk* in G is a sequence $(n_1, l_1), \ldots, (n_i, l_i) \ldots$ of elements such that

(1) $n_1, \ldots, n_i, \ldots$ is a valid path in G, and
(2) for any two consecutive elements $(n_i, l_i), (n_{i+1}, l_{i+1})$ for $i \geq 1$, if $l_i = T$ or $l_i = F$, then $n_{i+1} = tsucc(n_i)$ or $n_{i+1} = fsucc(n_i)$, respectively.

If $\omega = (n_1, l_1), \ldots, (n_i, l_i), \ldots$ is a walk, then $\bar{\omega} = n_1, \ldots, n_i, \ldots$ denotes the valid path for that walk. Danicic et al. [13] provided the definition of *walk* for intraprocedural CFG. They have made an informal semantic assumption that for any elemnet (n, l) in a walk ω such that n is a Cond node, l always represent the result of the evaluation of the conditional expression at n. The path $\bar{\omega}$ is always a feasible path for intraprocedural CFG due to this assumption. Def. 2.5 considers interprocedural CFG and $\bar{\omega}$ is also a feasible interprocedural path as we consider the same informal semantic assumption and the fact that $\bar{\omega}$ is also a valid path. We write n for (n, ϵ) within walks. $\overrightarrow{G}$ denotes the set of all walks in G. Let ω be a walk in G, and $N' \subseteq N$. The *restriction* of ω to N', denoted $\omega \downarrow N'$, is the subsequence of ω obtained by removing all elements (n, l) of ω where $n \notin N'$. Similarly, for paths π, we define $\pi \downarrow N'$ as the subsequence of π where all $n \in \pi$ that are not in N' are removed. If ω is a walk in G such that $\omega = \omega_1, \omega_2$ then ω_1 is a *prefix* of ω, and if $\omega \neq \omega_1$ then ω_1 is a *proper prefix* of ω. A walk ω is *maximal* in G when it is not a proper prefix of any other walk.

Example 2.6. Let us consider the CFG G_a in Fig. 3 on page 10. We obtain the maximal walk $start, n_1, n_2, (n_3, F), n_6, end$ from the finite terminating path $start, n_1, n_2, n_3, n_6, end$ in G_a. The walk $(n_3, T), n_4, n_5, (n_3, T), n_4, n_5, \ldots$ is also maximal since the path $n_3, n_4, n_5, n_3, \ldots$ is infinite nonterminating path. Note that a maximal walk either ends in a final node or is infinite. The sequence $(n_3, T), n_6, end$ is not a walk since $n_6 \neq tsucc(n_3)$.

2.3 Program Dependence and Slicing

Dependence-based slicing requires computing the *data* and the *control dependency* relations among CFG nodes.

Definition 2.7 (Data Dependency [39]). Node n is *data dependent* on node m (written $m \stackrel{dd}{\rightarrow} n$) in the CFG G if there is a program variable v such that: (1) there exists a nontrivial path π in G from m to n such that for every node $m' \in \pi - \{m, n\}$, $v \notin def(m')$, and (2) $v \in def(m) \cap ref(n)$.

Intuitively, $m \stackrel{dd}{\rightarrow} n$ specifies that the value of variable v that is assigned at node m may be used at n without being redefined by any node in the path $[m..n]$. Note that Def. 2.7 is for the intraprocedural case: if applied to programs with procedures, it will yield a very inexact data dependency relation since it will include many infeasible CFG paths. In Section 4.2 we will provide a more exact definition, suitable for the interprocedural setting.

For control dependency, *postdominators* [37] play an important role. Assume that the CFG G has a single End node n_e, and that there is a path from each node n in G to n_e. Node n is then said to *postdominate* node m if and only if every path from m to n_e goes through n. n *strictly postdominates* m if n postdominates m and $n \neq m$. There are many algorithms to compute postdominators, as well as *postdominator trees* which can be used to efficiently represent sets of postdominators for different nodes. The standard, *postdominator-based control dependency relation* can then be defined as follows:

Definition 2.8 (Control Dependency [16, 39]). Node n is *control dependent* on node m (written $m \stackrel{pcd}{\rightarrow} n$) in the CFG G if (1) there exists a nontrivial path π in G from m to n such that every node $m' \in \pi - \{m, n\}$ is postdominated by n, and (2) m is not strictly postdominated by n.

The relation $m \stackrel{pcd}{\rightarrow} n$ implies that there must be two branches of m such that n is always executed in one branch and may not execute in the other branch. However, in the presence of nonterminating loops, slicing based on the $\stackrel{pcd}{\rightarrow}$ relation will not always yield a satisfactory result: in particular, it will not always preserve nontermination. Consider the CFG fragment in Fig. 2, which contains an infinite loop between nodes c and s. The condition at c determines if s' will be executed or the

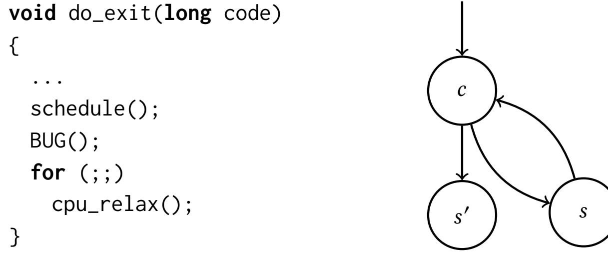

Fig. 2. Linux kernel code containing an infinite loop (left) and CFG fragment containing an infinite loop (right)

execution gets stuck between c and s due to the infinite loop. Nevertheless, the relation $c \stackrel{pcd}{\rightarrow} s'$ does not hold as c is strictly postdominated by s'. Thus, c will not be sliced even if s' is sliced.

This situation appears frequently in practice. For instance, consider the code from the Linux kernel in Fig. 2. Here the infinite `for`-loop at the end may be entered in some abnormal situations, and yet a slicing method based on $\stackrel{pcd}{\rightarrow}$ might not include this code in the slice. This may cause problems, for instance, if slicing is used in conjunction with formal verification of certain properties of the kernel. Another typical example is the embedded programs in which an embedded program consists of a number of tasks and each task is built by a nonterminating function. For instance, Engblom [15] have studied the static properties of a large sample of commercial and real-time embedded programs and identified 44 nonterminating loops in those sample code.

For this reason, a number of different control dependency relations, conservatively extending the standard relation above, have been defined [2, 34–36, 39]. For example, Podgurski and Clarke [36] provided definitions of strong and weak control dependence that are nontermination insensitive and nontermination sensitive. These relations consider that the CFG has a unique end node. In order to handle CFGs having infinite loops, no end node, or multiple end nodes, other kinds of control dependencies are proposed such as the relation $\stackrel{ntscd}{\longrightarrow}$ (Def. 6 in [39]) together with $\stackrel{dod}{\longrightarrow}$ (Def. 12 in [39] for handling irreducible CFGs) preserving nontermination, the relation $\stackrel{nticd}{\longrightarrow}$ (Def. 7 in [39]) ignoring nontermination, and the *weakly order dependency* relation $\stackrel{wod}{\longrightarrow}$ (Def. 2 in [2]). Danicic et al. [13] presented two generalisations of control dependence called *weak* and *strong control closure* which are non-termination insensitive and non-termination sensitive. Many existing control dependencies are specialisations of the above two kinds. For example, $\stackrel{wod}{\longrightarrow}$ and $\stackrel{ntscd}{\longrightarrow}$ are the *weak* and *strong* form of control dependencies not preserving and preserving non-termination respectively.

The rich set of control dependency relations is motivated by the various different uses of slicing, which give different demands on the properties of the slice. For instance preservation of nontermination is not always desirable, as it tends to yield significantly larger slices. This motivates a formal treatment of slicing that is valid over a whole range of control dependency relations.

In the rest of the paper $\stackrel{cd}{\rightarrow}$ will stand for any control dependency relation. We shall extend the *weak* and *strong control closure* relations of Danicic et al. for interprocedural programs capturing nontermination insensitive and sensitive control dependencies. Slicing based upon the *weak* and *strong* form of control dependencies make the slices nontermination insensitive and sensitive, respectively. In what follows, we define the concepts of *weak* and *strong projections* that are used to define nontermination insensitive and sensitive slice.

Definition 2.9 (Weak and Strong Projections [13]). Let $G = (N, E)$ be a CFG. A CFG $G' = (N', E')$, where $N' \subseteq N$, is a *weak projection* of G if and only if every walk of G, when restricted to N', is a walk of G'. That is,

$$\omega \in \overrightarrow{G} \Rightarrow \omega \downarrow N' \in \overrightarrow{G'}.$$

G' is a *strong projection* of G if and only if all maximal walks of G, when restricted to N', give maximal walks of G'. That is,

$$\omega \in \overrightarrow{G} \text{ is maximal} \Rightarrow \omega \downarrow N' \in \overrightarrow{G'} \text{ and is maximal.}$$

Example 2.10. The CFG G_a in Fig. 3 (a) has the following three kinds of maximal walks. All the suffixes of the following walks are maximal:

$$\begin{array}{ll} start, n_1, n_2, (n_3, T), n_4, n_5, (n_3, T), \ldots & \text{(infinite maximal)} \\ start, n_1, n_2, (n_3, T), n_4, n_5, (n_3, T), \ldots, (n_3, F), n_6, end & \text{(finite maximal)} \\ start, n_1, n_2, (n_3, F), n_6, end & \text{(finite maximal)} \end{array}$$

The maximal walks of G_b and G_c (grey nodes) in Fig. 3 (b,c) are the following:
Maximal walks of G_b:

$$\begin{array}{ll} start, n_1, n_6, end & \text{(finite maximal)} \\ n_1, n_6, end & \text{(finite maximal)} \\ n_6, end & \text{(finite maximal)} \\ end & \text{(finite maximal)} \end{array}$$

The maximal walks of G_c are all the suffixes of the following walks:

$$\begin{array}{ll} start, n_1, (n_3, T), n_4, (n_3, T), \ldots & \text{(infinite maximal)} \\ start, n_1, (n_3, T), n_4, (n_3, T), \ldots, (n_3, F), n_6, end & \text{(finite maximal)} \\ start, n_1, (n_3, F), n_6, end & \text{(finite maximal)} \end{array}$$

Any walk of a CFG is a prefix of a maximal walk. G_b is a weak projection of G_a since any walk of G_a when restricted to the set $\{start, n_1, n_6, end\}$ is also a walk of G_b. G_c is a strong projection of G_a since any maximal walk of G_a when restricted to the set $\{start, n_1, n_3, n_4, n_6, end\}$ is also a maximal walk of G_c. The maximal walk $start, n_1, n_2, (n_3, T), n_4, n_5, (n_3, T), \ldots$ of G_a when restricted to $\{start, n_1, n_6, end\}$ gives the walk n_1 of G_b which is not maximal, and hence G_b is not a strong projection of G_a.

See [13] for more examples of weak and strong projections. Note that the set of edges E' in Def. 2.9 is not restricted. But, we require that $G' = (N', E')$ is a valid CFG formed according to Def. 2.1. Regarding the relation between weak and strong projections, a strong projection is also a weak projection since every walk is the prefix of a maximal walk (Lemma 26 in [13]). To see an example, G_c is also a weak projection of G_a in Fig. 3 as any walk in G_a is also a walk in G_c when it is restricted to the set of nodes $\{start, n_1, n_3, n_4, n_6, end\}$.

In order to define nontermination (in)sensitive slices, we need to formally define the slices obtained from dependence-based slicing. A program slice is computed relative to a *slicing criterion* C, which is a set of pairs (n, V) where n is a CFG node, and V is a set of program variables of interest at n. For notational convenience, we consider C to be a partial function where the domain is the set of nodes and the codomain is the power set of the set of all variables. Thus, we write $C(n) = V$ instead of $(n, V) \in C$. We also write *nodes*(C) for the set of all nodes that belong to C. Static backward slicing (on CFG level) then selects a part of the CFG (called the *slice set*) that includes all nodes in the CFG that possibly can affect the memory locations for the respective CFG nodes in the slicing criterion. The slice set then defines the subset of the original CFG that constitutes the slice. For dependence-based slicing we define the slice set and slice as follows:

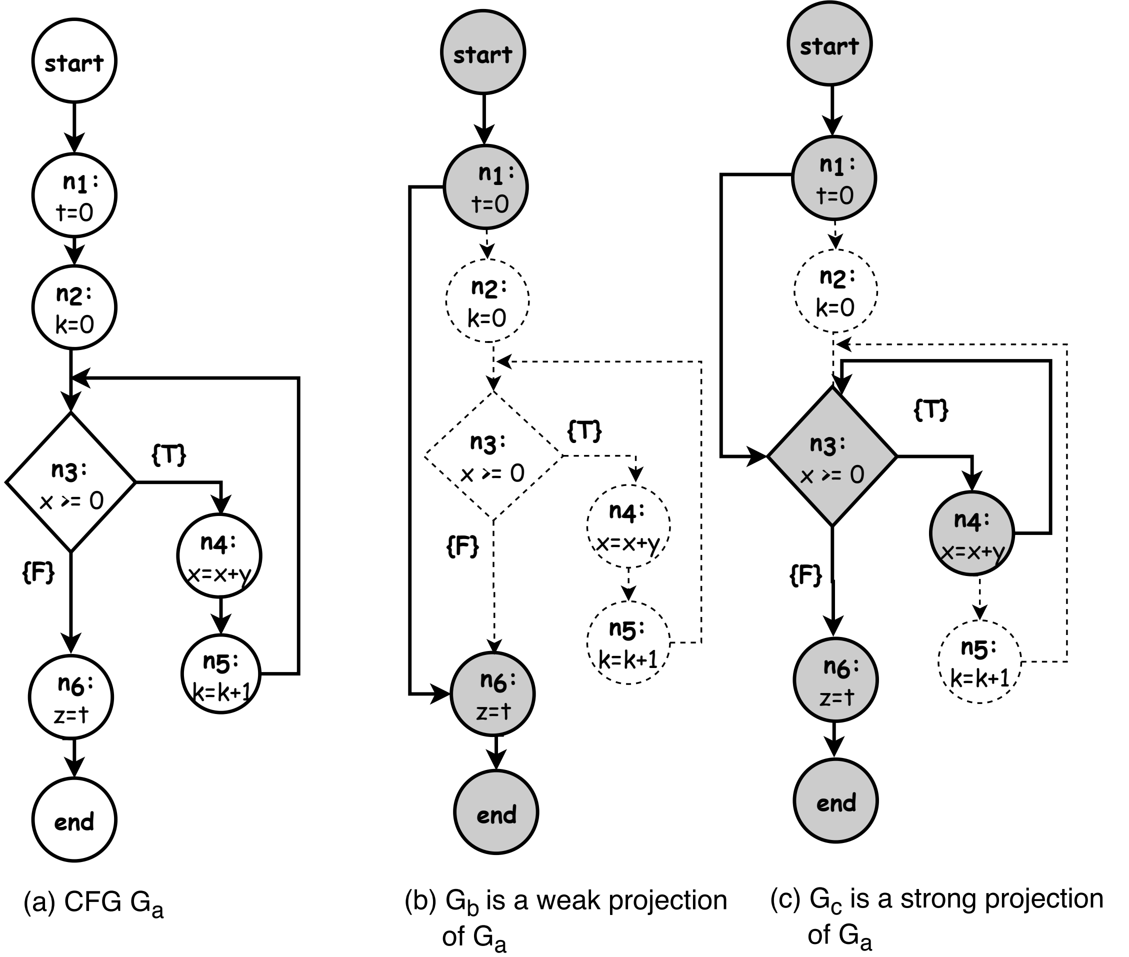

Fig. 3. Weak and strong projections.

Definition 2.11 (Slice Set). For any CFG $G = (N, E)$, control dependency relation $\xrightarrow{cd}$, and slicing criterion C, the *slice set* $S_C(G, \xrightarrow{cd})$ of G with respect to C is defined by

$$S_C(G, \xrightarrow{cd}) = \bigcup_{n \in nodes(C)} \{m : m(\xrightarrow{cd} \cup \xrightarrow{dd})^* n\}$$

where $\rightarrow^*$ denotes the transitive-reflexive closure of $\rightarrow$.

The above definition of slice set assumes, for simplicity, that $C(n) = ref(n)$ for all $n \in nodes(C)$. We shall write S_C for $S_C(G, \xrightarrow{cd})$ when G, and $\xrightarrow{cd}$ are clear from the context. Def. 2.11 defines the slice set for backward slicing. A similar definition can be done for forward slicing, using the relation $(\xrightarrow{cd} \cup \xrightarrow{dd})^*$ in the forward direction. For the CFG G in Fig. 4, the slicing criterion $C = \{(n_{12}, \{c\})\}$, and a post-dominator based control depenednecy relation $\xrightarrow{cd}$ capturing interprocedural control dependency, $S_C = \{n_0, n_8, \ldots, n_{15}, n_E, n_X\}$ is the slice set according to Def. 2.11.

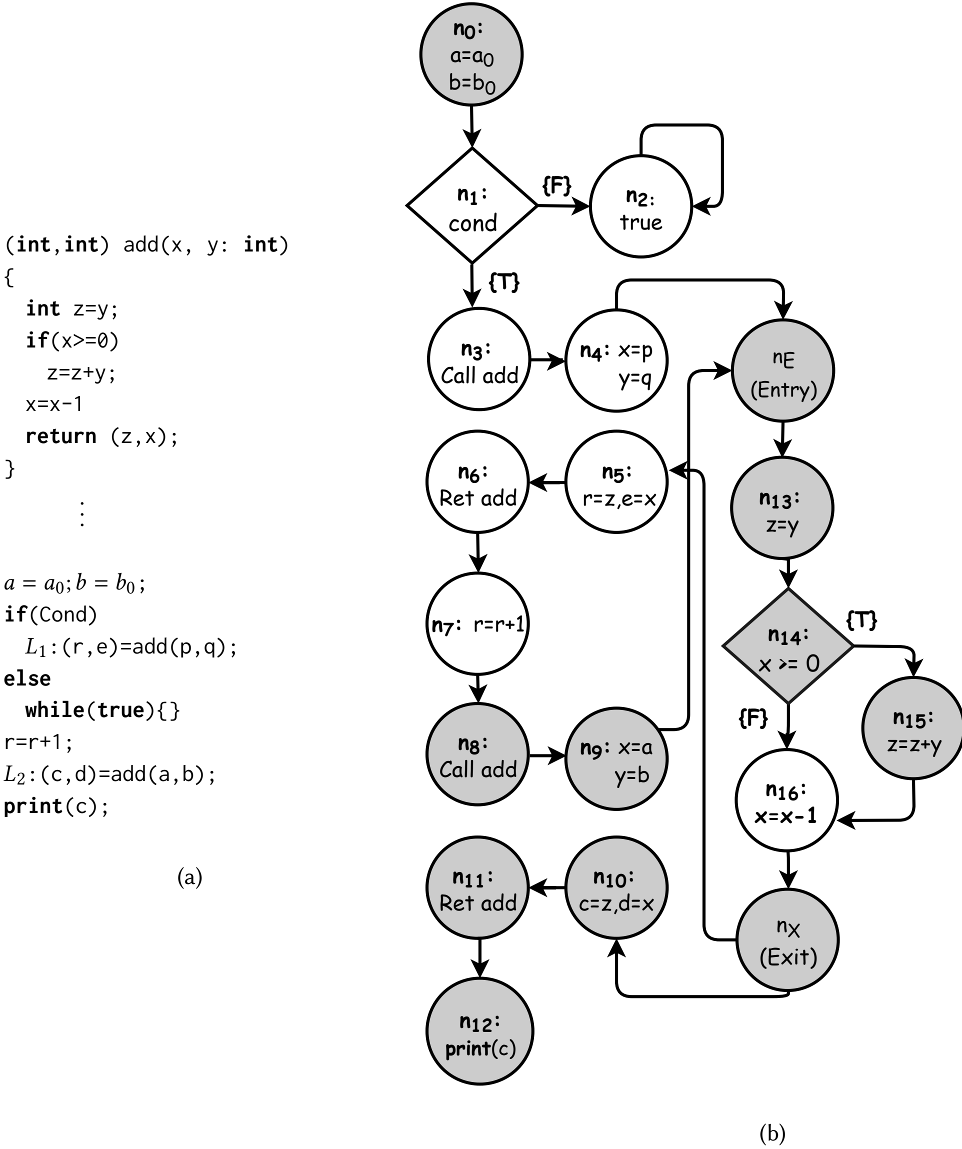

Fig. 4. (a) Interprocedural code and (b) CFG G of (a). Node n representing statement S is denoted by $n : S$ in G, $C = \{(n_{12}, \{c\})\}$ is the slicing criterion, and S_C includes all the grey nodes. $L_1 = \ell(n_3)$ and $L_2 = \ell(n_8)$ are the calling contexts

Given the slice set S_C of $G = (N, E)$ with respect to C and $\xrightarrow{cd}$, the *slice* of G is a CFG $slice_C(G, \xrightarrow{cd}) = (S_C, E')$ where E' captures the "rewiring" of control flow edges that has to be done due to nodes being removed in the slice. Note that if $\xrightarrow{dd}$ is defined by Def. 2.7, then Def. 2.11 will yield a very inexact overapproximation of the slice sets for interprocedural code. With the version of $\xrightarrow{dd}$

provided in Def. 4.6, Def. 2.11 will yield much tighter slice sets for such code. Our results are valid for this more precise definition.

We now define what a nontermination (in)sensitive slice is.

Definition 2.12 (Nontermination (In)Sensitive Slice). Let G be the CFG of program P_1, and let P_2 be the slice of program P_1 with respect to the slicing criterion C and the control dependency relation $\xrightarrow{cd}$. Program P_2 is a nontermination insensitive (or sensitive) slice of program P_1 if the CFG $slice_C(G, \xrightarrow{cd})$ of P_2 is a weak projection (resp. strong projection) of the CFG G of P_1.

In Section 5 we define *well-formed weak/strong control-closure* for interprocedural programs, which are necessary and sufficient conditions for the slice to be a weak/strong projection respectively. Our definitions extend the generalized weak/strong control-closure for intraprocedural programs provided by Danicic et al. [13].

3 THE SEMANTICS OF PROGRAM SLICING

Program semantics [33] is a natural tool to reason about the specification and properties of programs. Semantic equivalence is a notion used to understand if the result of some program transformation applied to code P preserves certain properties of P. In what follows, we give a structural operational semantics for interprocedural programs (Section 3.1), we develop some semantic relations such as *derivation sequence* and *C-derivation sequence* on the semantic transitions of programs (Section 3.2), we define two semantic relations called *nontermination (in)sensitive semi-equivalence* between a program and its slice, reflecting the program slicing semantics for possibly nonterminating programs (Section 3.3), and finally we compare our slice semantics provided by the semi-equivalence relations with the Weiser semantics, the lazy, transfinite, non-standard, finite trajectory, and semirefinement semantics for terminating, nonterminating, and nondeterministic programs (Section 3.4).

In what follows $G = (N, E, N_e, start)$ is the CFG of program P_1, S_C is its slice set computed according to Definition 2.11, and $code_1$ and $code_2$ are two functions mapping the CFG G to code P_1 and the sliced code P_2, respectively.

3.1 Program Semantics

Suppose **Var** is the set of all program variables, **Val** is the set of all values that variables in **Var** can take, and $\mathcal{N}$ is the set of all CFG nodes. A *store* σ is an element of the set **Store** = **Var** $\rightarrow$ **Val** of partial functions that map program variables to the corresponding values. A program state is an element of the set **State** = $\mathcal{N} \times$ **Store**, that is: a pair (n, σ) where n is a CFG node, and σ is a store. **Config** = $(\textbf{State})^*$ is the set of all *program configurations*, where a program configuration can be seen as an abstract *call stack* $(n_0, \sigma_0) \cdot \ldots \cdot (n_m, \sigma_m)$ of program states where the top is (n_m, σ_m). The operational semantics of an interprocedural program can be expressed as transitions on program configurations. The transition relation $\rightarrow \subseteq$ **Config** $\times$ **Config** is summarized in Table 1. For $i = 1, 2$, we write $i \vdash \Gamma \cdot s \rightarrow \Gamma' \cdot s'$ if the program obtained through $code_i$ transforms (input) configuration $\Gamma \cdot s$ into (output) configuration $\Gamma' \cdot s'$ where s, s' range over program states. For any configuration $\Gamma \cdot (n, \sigma)$, we define $node(\Gamma \cdot (n, \sigma)) = n$ and $store(\Gamma \cdot (n, \sigma)) = \sigma$ (i.e., the currently visited node n and its corresponding store σ). $[\![e]\!]\sigma$ denotes the evaluated value of expression e in σ, $\sigma[x \mapsto v]$ denotes the store where $\sigma[x \mapsto v](x) = v$, and $\sigma[x \mapsto v](y) = \sigma(y)$ for $y \neq x$, and $\sigma[\sigma']$ is the store for which $\sigma[\sigma'](x) = \sigma'(x)$ when $x \in dom(\sigma')$, and $\sigma[\sigma'](x) = \sigma(x)$ otherwise. All variables in a store are local variables, or formal input parameters. The scoping rules of variables and how to model global variables are discussed later in this section.

The semantic rules in Table 1 describe the changes in configurations based on the top state (n, σ) in the input configuration $\Gamma \cdot (n, \sigma)$. Intuitively, the rules are as follows:

Table 1. Semantic rules defining transitions

$$\frac{code_i(n) = \ x{=}e \quad n'{\in}succ(n) \quad v{=}[\![e]\!]\sigma}{i \vdash \Gamma \cdot (n,\sigma) \rightarrow \Gamma \cdot (n', \sigma[x \mapsto v])} \text{ (ASSIGN)}$$

$$\frac{code_i(n){=}b \;\; tsucc(n) \neq \bot \;\; [\![b]\!]\sigma{=}true}{i \vdash \Gamma{\cdot}(n,\sigma) \rightarrow \Gamma{\cdot}(tsucc(n),\sigma)} \text{ (TCOND)} \qquad \frac{code_i(n){=}b \;\; fsucc(n) \neq \bot \;\; [\![b]\!]\sigma{=}false}{i \vdash \Gamma{\cdot}(n,\sigma) \rightarrow \Gamma{\cdot}(fsucc(n),\sigma)} \text{ (FCOND)}$$

$$\frac{n : Call \quad n' \in succ(n)}{i \vdash \Gamma \cdot (n,\sigma) \rightarrow \Gamma \cdot (n,\sigma) \cdot (n',\sigma)} \text{ (CALL)}$$

$$\frac{code_i(n) = \ x_1 = e_1, \ldots x_k = e_k \quad n : CAssign \quad n' \in succ(n) \quad v_j = [\![e_j]\!]\sigma}{i \vdash \Gamma \cdot (n,\sigma) \rightarrow \Gamma \cdot (n', \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k\})} \text{ (PARAMIN)}$$

$$\frac{code_i(n) = \ x'_1 = e'_1, \ldots x'_l = e'_l \quad n : RAssign \quad n' \in succ(n) \quad v'_j = [\![e'_j]\!]\sigma}{i \vdash \Gamma \cdot (n'',\sigma') \cdot (n,\sigma) \rightarrow \Gamma \cdot (n', \sigma'[\{x'_j \mapsto v'_j \mid 1 \leq j \leq l\}])} \text{ (PARAMOUT)}$$

$$\frac{n : T \;\; T \in \{Entry, Start, Skip, Ret\} \;\; n' \in succ(n)}{i \vdash \Gamma \cdot (n,\sigma) \rightarrow \Gamma \cdot (n',\sigma)} \text{ (SKIP)}$$

$$\frac{n : Exit \;\; n'' \in succ(n) \;\; m \in succ(n'') \;\; \ell(n') = \ell(m)}{i \vdash \Gamma \cdot (n',\sigma') \cdot (n,\sigma) \rightarrow \Gamma \cdot (n',\sigma') \cdot (n'',\sigma)} \text{ (EXIT)}$$

$$\frac{n : Call \;\; code_2(n) = skip \;\; n' : Ret \;\; \ell(n') = \ell(n)}{2 \vdash \Gamma \cdot (n,\sigma) \rightarrow \Gamma \cdot (n',\sigma)} \text{ (CALL-SKIP)}$$

- ASSIGN: If the CFG node n in the top state (n, σ) represents an assignment $x = e$, then the store σ is updated by assigning $[\![e]\!]\sigma$ to x.
- TCOND and FCOND: These rules are applicable when the CFG node n in the input configuration $\Gamma \cdot (n, \sigma)$ is a Cond node with a boolean expression. The resulting configuration is $\Gamma \cdot (n', \sigma)$ where n' is the immediate successor of n for the *true* or *false* branch (i.e. $n' = tsucc(n)$ or $n' = fsucc(n)$), respectively, depending on the evaluation of the conditional expression. The condition $tsucc(n) \neq \bot$ (or $fsucc(n) \neq \bot$) indicates that there exists a true (resp. false) successor of n.
- CALL: Here, n in the top state (n, σ) represents a call statement. The input configuration is augmented with a new top state (n', σ), where n' is the immediate successor of n, preparing for the change of context where the assignments from actual parameters to formal input parameters are done. Node n' will be a CAssign node representing these assignments, and the store σ copied to the top state allows subsequent evaluation of the actual parameters: see the PARAMIN rule.
- PARAMIN: The top states (n, σ) in the input configuration correspond to CFG nodes representing simultaneous assignments to formal input parameters. The top state (n', σ') in the output configuration corresponds to the CFG node n' which is the only successor of n, and σ' is the corresponding store of n' containing values for formal input parameters.
- PARAMOUT: The top state (n, σ) in the input configuration corresponds to CFG nodes representing simultaneous assignments to actual output parameters before a return from a procedure call. According to the CALL rule, the second top state (n'', σ') corresponds to the call of that procedure. The output configuration is obtained by discarding (n'', σ'), and forming a new top state $(n', \sigma'[M])$ where n' is the Ret node representing a return from the

procedure call, M contains the mappings from the actual output parameters to their values evaluated in the store σ, σ' is the store before the procedure call, and $\sigma'[M]$ is equivalent to the store before the procedure call but updated with the new values of the actual output parameters.

- Skip: This rule is applicable when the CFG node is a Start, Skip, Entry, or Ret node. The output configuration $\Gamma \cdot (n', \sigma)$ is formed from the input configuration $\Gamma \cdot (n, \sigma)$ such that the stores in the top states are not changed and n' is the successor node of n.
- Exit: Here, the CFG node n represents a procedure exit node. There can be more than one successor node of n returning to different call sites. The second top state (n', σ') in the configuration stack indicates the current caller of this procedure. This rule ensures the return to the correct call site by finding the reachable Ret node m (from n), and verifying that n' and m have the same label.
- Call-skip: This rule is only applicable for P_2. The top state (n, σ) in the input configuration corresponds to a Call node, but $code_2(n) = skip$ instead of a procedure call. This situation appears if the call is not sliced: then it is replaced by a *skip* statement in the sliced code. The output configuration is formed from the input configuration in which (n, σ) is replaced by (n', σ), where n' represents the return from the call. Thus, P_2 avoids entering the procedure and moves on to the next instruction.

We consider simple static scoping of variables as follows:

- All variables defined or referenced in a procedure are either local variables or formal parameters of that procedure.
- Global variables are modeled by using the actual parameters of procedure calls and formal procedure parameters explained below.
- We do not consider nested scoping of variables. So, variable names are unique in a procedure. The scope of all local variables and formal parameters are the entire body of that procedure.

The semantic rules in Table 1 define the scope of program variables (see Call, Ret, ParamIn, ParamOut rules). For these semantic rules, the current variables in scope are the variables that are defined by the store in the top state. Initially, at procedure entry, these variables are the formal input parameters as defined by the ParamIn rule. Other variables will be undefined unless explicitly assigned during transitions, and an attempt to evaluate an expression containing such variables will yield an undefined result. Global variables x_g can be modeled by turning them into formal input parameters and actual output parameters, for all procedures in the program, in the following way: for each procedure call, the CAssign node holds an assignment $x_g = x_g$ passing the current value of x_g to itself within the new top store, and the RAssign node at procedure exit holds a similar assignment passing the possibly updated value of x_g to the new top store of the caller. The initial definition of local variables (different from formal input parameters) in the top store are modeled by ordinary assignment (Assign rule) due to initializations (if any).

Without loss of generality we assume that $(start, ref(start))$, and $(n_e, ref(n_e))$ for an End node $n_e \in N_e$, belong to the slicing criterion C. An *initial configuration* is a configuration $(start, \sigma_0)$, where σ_0 is its *initial store*. If $n_e \in N_e$ then (n_e, σ) is a *final terminating configuration*, for any store σ. If $\Gamma \cdot (n, \sigma)$ is any configuration such that $n \notin N_e$ but n is a final node from where no transition $i \vdash \Gamma \cdot (n, \sigma) \to \Gamma'$ exists for $i \in \{1, 2\}$, then $\Gamma \cdot (n, \sigma)$ is a *final nonterminating configuration* (i.e., the program is silently nonterminating).

3.2 Derivation Sequences

Definition 3.1. A *derivation sequence* of program P_i for $i \in \{1, 2\}$ is either

- a finite sequence of configurations $\Gamma_0, \ldots, \Gamma_k$ such that $k \geq 0$ and $i \vdash \Gamma_j \longrightarrow \Gamma_{j+1}$ for all $0 \leq j < k$,
- or an infinite sequence of configurations $\Gamma_0, \Gamma_1, \ldots$ such that $i \vdash \Gamma_j \longrightarrow \Gamma_{j+1}$ for $j \geq 0$.

For any set of CFG nodes X we define the transition relation $i \vdash \Gamma_0 \longrightarrow_X \Gamma_k$, capturing the steps between the configurations in a derivation sequence that visit nodes in X, as follows:

Definition 3.2 ($i \vdash \Gamma_0 \longrightarrow_X \Gamma_k$). The relation $i \vdash \Gamma_0 \longrightarrow_X \Gamma_k$ holds iff there is a finite derivation sequence $\Gamma_0, \ldots, \Gamma_k$ such that $node(\Gamma_0) \in X$, $node(\Gamma_k) \in X$, and $node(\Gamma_j) \notin X$ for all $1 \leq j \leq k-1$.

Thus, $\longrightarrow_X$ captures derivation sequences starting from and ending in X without visiting X in-between. In the following we will consider the cases where $X = nodes(C)$, and $X = S_C$. For brevity we will write $\longrightarrow_C$ for $\longrightarrow_{nodes(C)}$.

Definition 3.3. A (finite or infinite) *C-derivation sequence* of program P_i for $i \in \{1, 2\}$ is either

- a finite sequence of configurations $\Gamma_0, \ldots, \Gamma_k$ such that $k \geq 0$, $node(\Gamma_0) \in nodes(C)$, and $i \vdash \Gamma_j \longrightarrow_C \Gamma_{j+1}$ for $0 \leq j < k$, or
- an infinite sequence of configurations $\Gamma_0, \Gamma_1, \ldots$ such that $i \vdash \Gamma_j \longrightarrow_C \Gamma_{j+1}$ for $j \geq 0$.

Example 3.4. Consider the CFG G_a in Fig. 3, the slicing criterion C containing the pairs $(n, ref(n))$ for $n \in \{start, n_6, end\}$, and the initial store $\sigma_0 = \{x \mapsto -1, y \mapsto 0\}$. We obtain the derivation sequence

$$(start, \sigma_0), (n_1, \sigma_0), (n_2, \sigma_1), (n_3, \sigma_2), (n_6, \sigma_2), (end, \sigma_3)$$

from G_a where $\sigma_1 = \sigma_0[t \mapsto 0], \sigma_2 = \sigma_1[k \mapsto 0]$ and $\sigma_3 = \sigma_2[z \mapsto 0]$. We thus have a transition $1 \vdash (start, \sigma_0) \longrightarrow_C (n_6, \sigma_2)$ and $1 \vdash (n_6, \sigma_2) \longrightarrow_C (end, \sigma_3)$. The derivation sequence $(start, \sigma_0), (n_6, \sigma_2), (end, \sigma_3)$ is thus a C-derivation sequence.

We use the notation $\Gamma_1..\Gamma_k(..)$ to represent that it is either a finite derivation sequence $\Gamma_1, \ldots, \Gamma_k$ or an infinite derivation sequence $\Gamma_1, \Gamma_2, \ldots$. If there exists a derivation sequence $\Gamma_0, \ldots, \Gamma$ from an initial configuration Γ_0 of P_i for $i \in \{1, 2\}$, then Γ is a *valid configuration* of P_i. A finite C-derivation sequence $\Gamma_0, \ldots, \Gamma_k$ of P_i is *maximal* if Γ_k is a a final configuration. Also, all infinite C-derivation sequences are maximal.

3.3 Semi-equivalence Relations

We now define two *semi-equivalence* relations between a program and its slice. These are relations, defined with respect to a slicing criterion, that (i) agree with the standard program slice semantics of Weiser, and (ii) extend to infinite execution traces handling possibly nonterminating programs. The first relation captures correctness of nontermination insensitive slices, and the second captures correctness of nontermination sensitive slices.

First, we define equivalence of two stores with respect to a set of variables. For any store σ, let $var(\sigma)$ denote the set of all variables x for which $\sigma(x)$ is defined.

Definition 3.5 (Equivalence of stores). Let σ_1 and σ_2 be stores. The store σ_1 is *equivalent to σ_2 with respect to a set of variables* X (denoted $\sigma_1 =_X \sigma_2$) iff

$$\forall x \in X. x \in (var(\sigma_1) \cap var(\sigma_2)) \implies \sigma_1(x) = \sigma_2(x).$$

Thus, two stores are not equivalent only when there exists a variable in X that is defined in both stores but mapped to different values. In the following, we first define the the relation $\gtrsim_C$ and then use it to define the semi-equivalence relations between programs P_1 and P_2.

Definition 3.6 ($\gtrsim_C$). The relation $P_1 \gtrsim_C P_2$ holds between the programs P_1 and P_2 iff for all C-derivation sequences $\Gamma_0..\Gamma_k(..)$ of P_1, there exists a C-derivation sequence $\Gamma'_0..\Gamma'_k(..)$ of P_2 such that $node(\Gamma_i) = node(\Gamma'_i)$ and $store(\Gamma_i) =_{C(node(\Gamma_i))} store(\Gamma'_i)$ for all $i \geq 1$.

Definition 3.7 (Semi-equivalences). Program P_2 is *nontermination-insensitively semi-equivalent* to P_1 with respect to C iff $P_1 \gtrsim_C P_2$ holds. Program P_2 is *nontermination-sensitively semi-equivalent* to P_1 with respect to C (denoted $P_1 \simeq_C P_2$) iff $P_1 \gtrsim_C P_2$ and $P_2 \gtrsim_C P_1$.

Thus, the slice P_2 is nontermination-insensitively semi-equivalent to the program P_1 if it holds that for any execution trace of P_1 there exists an execution trace of P_2 that visits the nodes in $nodes(C)$ in the same sequence, and whenever such a node $n \in nodes(C)$ is visited the top stores of the corresponding configurations agree on the values of the variables in $C(n)$. Thus, this semi-equivalence agrees with the standard semantics of Weiser and Binkley. For the nontermination-sensitive semi-equivalence, the symmetric case must hold. Then, both P_1 and P_2 are either terminating or nonterminating, they visit the nodes in $nodes(C)$ in the same sequence, and compute the same values for the variables in $C(n)$ for all $n \in nodes(C)$.

3.4 Relative comparison of the slicing semantics

Numerous attempts have been made by different authors to define the semantics of slicing for terminating, nonterminating, and nondeterministic programs. Ward and Zedan in [48] have identified a number of problems that are latent in the definition of those proposed slice semantics. In this section, we shall illustrate that the semantics provided by the semi-equivalence in Def. 3.7 does not suffer from those problems.

3.4.1 Weiser slicing. The Weiser slicing [50] as illustrated by Ward and Zedan consists of two parts:

(1) A *syntactic part*: the slice P_2 must be formed from the program P_1 by deleting statements, or equivalently by replacing statements by *skip* statements.
(2) A *semantic part*: the slice P_2 must preserve the values of the variables of interest at the points of interest.

The authors explained Weiser's intention: "any code which does not involve the variables of interest can be deleted, regardless of whether or not that code terminates" [48]. The semantic part is further characterized by the *semi-refinement* relation that allows any program as a valid slice when the original program does not terminate. Thus, Weiser slicing allows us to delete irrelevant code, nonterminating code, and code that appears after a nonterminating loop.

Let us consider a program P_1 and its nontermination insensitively semi-equivalent slice P_2 (i.e. $P_1 \gtrsim_C P_2$). We dissect the conditions for the relation $P_1 \gtrsim_C P_2$ as follows:

(1) for all C-derivation sequences $\Gamma_0..\Gamma_k(..)$ of P_1, there exists a C-derivation sequence $\Gamma'_0..\Gamma'_k(..)$ of P_2, and
(2) for the C-derivation sequences in condition (1) above, it holds that $node(\Gamma_i) = node(\Gamma'_i)$ and $store(\Gamma_i) =_{C(node(\Gamma_i))} store(\Gamma'_i)$ for all $i \geq 1$.

From conditions (1) and (2) we conclude that:

- if the original program P_1 visits the statement $code_1(n)$ where $n \in nodes(C)$, then there exists an execution of P_2 that visits $code_2(n)$,
- if P_1 visits $code_1(n)$ for any $n \in nodes(C)$ and its execution is finite, then P_2 visits $code_2(n)$ the same number of times, and
- both of them compute the same values for the variables in $C(n)$.

However, if the execution of P_1 is nonterminating then P_2 visits $code_2(n)$ at least as often as P_1. Program P_2 may have infinite loops removed that do not affect the slicing criterion, and any slicing point after an infinite loop will be visited by P_2 more often than P_1. Conditions (1) and (2) also imply that

- if P_1 visits $code_1(n)$ infinitely many times for any $n \in nodes(C)$, then P_2 also visits $code_2(n)$ the same number of times, and
- both compute the same values for the variables in $C(n)$.

If the original program P_1 is such that any statement $code_1(n)$ is inside an infinite loop, and $n \in nodes(C)$, then $code_2(n)$ must be in an infinite loop in P_2. We can conclude from the above that a slice that is nontermination insensitively semi-equivalent to the original program preserves the values of all the variables of interest specified in the slicing criterion. Since the relation $P_1 \gtrsim_C P_2$ also holds for a nontermination sensitively semi-equivalent slice P_2 of program P_1, a nontermination sensitively semi-equivalent slice also preserves the values of all the variables of interest at the points of interest. So, the semantic condition of Weiser slicing is met by the semi-equivalence relations when the slice is nontermination (in)sensitive. Any statement of P_1 that does not affect the values of the variables in the slicing criterion, and does not affect the nontermination behavior, can be deleted in P_2 (or equivalently replaced by a **skip** statement or **true**/**false** conditions, see Section 4.5 for the formal discussion).

Now, let us consider a nontermination insensitively but not nontermination sensitively semi-equivalent slice P_2 of P_1. Thus, $P_1 \gtrsim_C P_2$ holds, but $P_2 \gtrsim_C P_1$ does not hold and the slice is not nontermination sensitive. In this case, the semi-equivalence relation allows deleting nonterminating code that does not include any statement as part of the specification in the slicing criterion, and any unreachable code even if it is part of the specification of the slicing criterion! Let us explain with examples.

In program Q_1 (Fig. 5), the statement `X = 1` at L_2 affects the value of the variable X at the end of the program. Thus, neither Weiser slicing nor semi-equivalence allows it to be deleted. In program Q_2, Weiser slicing allows deleting the nonterminating loop at label L_3, and the statement `X = 1` at L_4 after the loop as well, as this statement is unreachable and thus cannot affect the value of the variable X in the slicing criterion. If the execution of Q_2 reaches L_3, then there is no C-derivation sequence from the **then** branch of the **if** statement due to the infinite loop. So, there is no obligation to include the infinite loop into a slice which is not nontermination sensitively semi-equivalent to the original program. Moreover, the unreachable statement `X = 1` at L_4 neither affects the value of any variable in the slicing criterion nor does it contribute to making a transition to form a C-derivation sequence by Q_2. So, a nontermination insensitively but not nontermination sensitively semi-equivalent slice allows deleting unreachable statements.

In summary, the original program P_1 may include the following kinds of statements: (i) nonterminating loops that are reachable and contain slicing point(s), (ii) nonterminating loops that contain no slicing point, and (iii) statements that are unreachable. It is always incorrect (i.e. $P_1 \gtrsim_C P_2$ and $P_1 \simeq_C P_2$ do not hold) to delete nonterminating loops of case (i), always correct to delete statements of case (iii) from P_2, and loops of case (ii) can be deleted unless P_2 is nontermination sensitively semi-equivalent to P_1.

The syntactic part of the semi-equivalence relation is less strict than that of Weiser slicing. There is no strict requirement that the slice has to be formed from the original program by deleting statements or replacing irrelevant statements by **skip** statements. However, the semi-equivalence relations assume the same CFG G for both the original program P_1 and its slice P_2 such that $P_1 = code_1(G)$ and $P_2 = code_2(G)$. Programs P_1 and P_2 thus have the same CFG structure, but a CFG node n may represent different code, i.e. we may have that $code_1(n) \neq code_2(n)$. In order to understand its implication, consider the code in Fig. 6. Suppose we are interested in the value of the variable Z at L_5. The value of Z at L_5 is always 2. So, according to Def. 3.7, Q_4 is a correct slice of Q_3. This slice is not formed syntactically from Q_3 according to the rules for Weiser slicing.

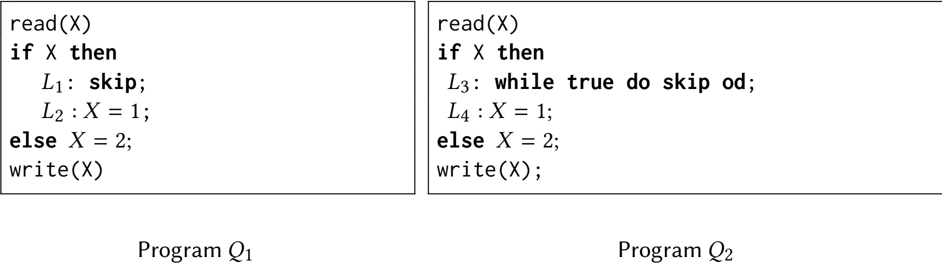

Fig. 5. The slicing criterion consists of the variable X in the statement *write*(X). Programs Q_1 and Q_2 are used to illustrate Weiser slicing in [48].

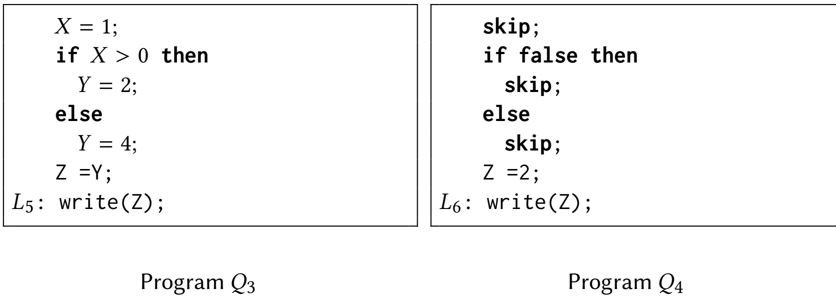

Fig. 6. The slicing criterion consists of the variable Z at the program point L_5. According to the semi-equivalence relation, Q_4 is a valid slice of Q_3.

3.4.2 Termination property of the slice. There exist lazy, transfinite, non-standard, and finite trajectory semantics that allow a nonterminating program as a valid slice of a terminating program. As Ward and Zedan uncovered in [48], the lazy program slicing semantics of Cartwright and Felleisen [10], the transfinite semantics of Giacobazzi and Mastroeni [17] and Nestra [32], the non-standard semantics of Danicic et al. [14], and the finite trajectory semantics of Barraclough et al. [5] all suffer from this problem. The problem is illustrated by the programs Q_5 and Q_6 in Fig. 7. Program Q_6 is formed from Q_5 by deleting the statement labeled L_1. The while loop in Q_5 always terminates, but the loop in Q_6 does not when the initial value of Y is greater than 0. All these semantics allow Q_6 to be a valid slice of Q_5 as it is allowed to remove part of the irrelevant code regardless of whether the termination property is maintained or not. See the relevant papers for the details.

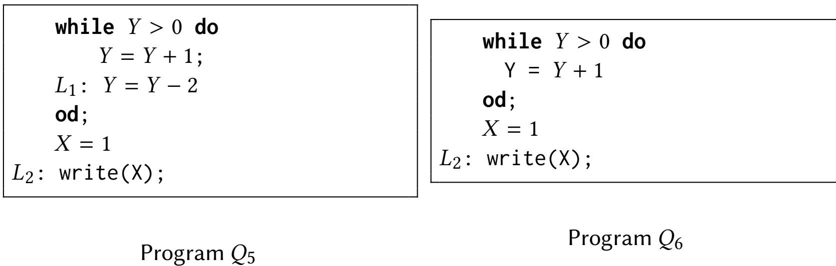

Fig. 7. The slicing criterion consists of the variable X at the program point L_2. Q_6 is a valid slice of Q_5 under the lazy, transfinite, non-standard, and finite trajectory semantics [48].

However, there always exists a C-derivation sequence in Q_5 that visits the statement labeled L_2, but there may not have a C-derivation sequence that can visit that statement in Q_6. All derivation sequences in Q_6 are diverging when $Y > 0$. Thus, Q_6 is not a valid slice of Q_5 according to the semi-equivalence semantics.

This paper mainly deals with deterministic programs. A program is deterministic if there exist at most one transition from any given configuration. The semi-equivalence relations do not allow any nonterminating code as a valid slice of a terminating code if the slice is deterministic. We prove the following theorem regarding the termination property of a nontermination (in)sensitively semi-equivalent slice:

Theorem 3.8. *For all programs P_1, and any deterministic semi-equivalent slice P_2 of P_1 such that $P_1 \gtrsim_C P_2$, holds that if P_1 is a terminating program, then all executions of P_2 are terminating.*

Proof. Without loss of generality we assume that the Start node, as well as the End node, belong to $nodes(C)$. Note that the End node must exist, since P_1 is terminating. Consider any initial configuration $\Gamma'_0 = (start, \sigma')$ of P_2. Then there exists an initial configuration $\Gamma_0 = (start, \sigma)$ of P_1 such that $\sigma =_{var(\sigma')} \sigma'$: we simply pick a σ such that $\sigma(x) = \sigma'(x)$ for all $x \in var(\sigma')$. Now, since P_1 is terminating, all maximal C-derivation sequences of P_1 are finite. Suppose $\Gamma_0, \ldots, \Gamma_k$ is a maximal C-derivation sequence of P_1 where $node(\Gamma_k) \in N_e$. Since $P_1 \gtrsim_C P_2$, there exists a C-derivation sequence $\Gamma'_0, \ldots, \Gamma'_k$ of P_2 where $node(\Gamma'_k) = node(\Gamma_k) \in N_e$. This C-derivation sequence is a projection of a derivation sequence D of P_2 starting in Γ'_0. Since D visits N_e, it is terminating. As we consider deterministic slices only, where there is at most one transition from any configuration, D is the *unique* derivation sequence starting in Γ'_0. Thus, there can only be terminating derivation sequences for P_2. □

Theorem 3.8 holds for both nontermination insensitive and nontermination sensitive slices as $P_1 \gtrsim_C P_2$ is the correctness criteria for both kinds of slices according to the definition of semi-equivalences. The condition that P_2 is deterministic is important for the proof: the theorem does not extend to nondeterministic slices in general. Slicing of nondeterministic programs is further discussed in Section 3.4.3.

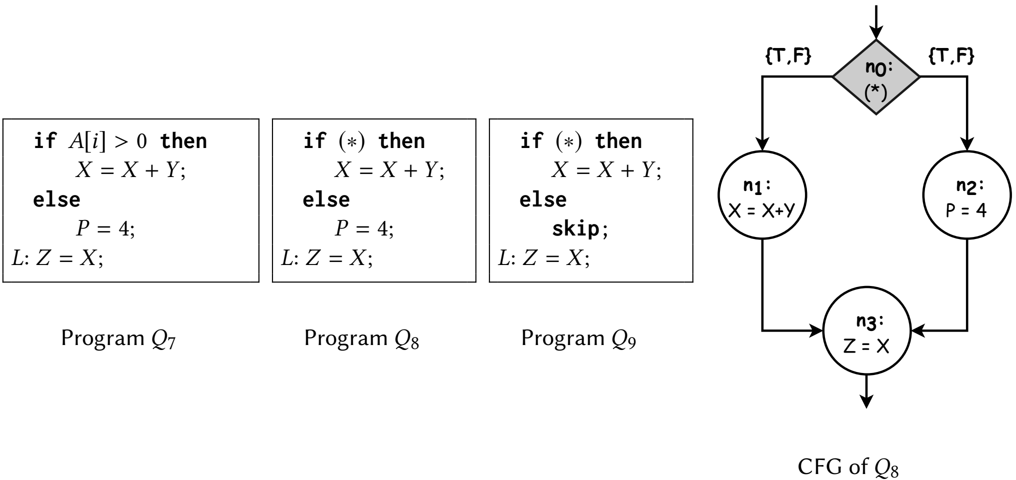

Fig. 8. Program Q_8 is an abstraction of Q_7 in which the condition $A[i] > 0$ in Q_7 is abstracted to the nondeterministic choice condition $(*)$.

3.4.3 Slicing nondeterministic programs. Most practical programs have deterministic execution semantics. However, the concept of nondeterminism is important in computer science in the context of nondeterministic Turing machines, concurrency, or program abstraction. For example, nondeterministic code may appear during the early stage of program analysis where certain parts of the code are abstracted away during the generation of intermediate code representation. Consider the program Q_7 in Fig. 8 where the condition includes an array access. In static program analysis, it is usually difficult to reason about array accesses, and conditions containing array accesses are often represented by nondeterministic choices. Program Q_8 is thus formed from Q_7 by replacing the condition $A[i] > 0$ by the nondeterministic choice condition $(*)$. Node n_0 represents this nondeterministic choice in the CFG of Q_8. Both branches of n_0 have the edge label $\{T, F\}$ meaning that execution might continue nondeterministically to node n_1 or n_2 from n_0. Consider the slicing criterion $C = \{(n_3, \{X\})\}$ and the initial store $\{X \mapsto x_0, Y \mapsto y_0\}$. As the execution is nondeterministic, the value of X at n_3 can be either $x_0 + y_0$ due to executing the **then** branch, or the value x_0 due to the **else** branch.

The definition of slicing provided by Binkley and Gallagher [9] does not consider nondeterministic programs. Their definition requires that both the original program and its slice compute exactly the same value for the variable specified in the slicing point whenever the execution reaches there. Thus, program Q_8 is not a slice of itself according to this definition as different executions of Q_8 might provide different results for X at n_3. Slicing based on *reduction* and *refinement* [46] does not consider nondeterministic programs either. *Reduction* is a syntactic relation: program Q' is a reduction of Q if Q' can be constructed from Q by substituting some of its statements by **skip** statements. *Refinement* is a semantic relation that uses the semantic function f such that if a program starts its execution from an initial state s, $f(s)$ consists of the set of all the possible final states of the program. If Q and Q' have the semantic functions f and f', then Q' is a refinement of Q if $f'(s) \subseteq f(s)$ for all initial states s. So, according to this definition, $Z = X$; is a valid slice of program Q_8 in Fig. 8 as (i) replacing the **if-then-else** block by the **skip** statement is a valid reduction which can be deleted afterwards, and (ii) $Z = X$; is a valid refinement of Q_8 since the final value of X is x_0 and $\{x_0\} \subseteq \{x_0 + y_0, x_0\}$. However, this slice does not preserve the nondeterministic behavior of the original program. This problem was later fixed by using *semirefinement* [45] that basically requires the equality $f(s) = f'(s)$ for all the initial states s when both Q and Q' are terminating. Ward and Zedan later extended *semirefinement* in [48] to handle the slicing of potentially nondeterministic nonterminating programs where the slicing point can be in the middle of the code.

The behavior of a nondeterministic program in some state can be represented by a *computation tree* [18] as follows:

- The nodes are labeled with states, the root is labeled with the start state, and the leaves are labeled with final states (i.e. states in which the program terminates) or failure states (special states representing failures).
- A branching point in the tree indicates a nondeterministic choice point. Each branch of a nondeterministic choice represents different possibilities of computation when the execution starts from the initial state.
- The tree may contain infinite paths indicating potential nonterminating computation of the program.

The computation at any program instruction I can be obtained from the labeled states associated with I at different branches of the computation tree. All those states at different branches represent different possible outcomes of executing I when the execution starts from the initial state. Thus, we get a set of possible values of program variables for executing any program instruction I. The set is finite if all paths of the computation tree are finite. A nontermination (in)sensitively

semi-equivalent slice perfectly preserves the nondeterministic behavior of the original program. Consider the CFG G of a nondeterministic program $P_1 = code_1(G)$, its slice $P_2 = code_2(G)$, and the CFG node $n \in nodes(C)$ such that there exists a nondeterministic choice condition in G that decides if n will be executed at all, or not. Suppose the execution of P_1 and P_2 starts from an equivalent initial state, i.e., for all initial stores σ_0 of P_1 and σ'_0 of P_2, $\sigma_0 =_{var(\sigma')} \sigma'$ holds. The following definition describes what we mean by preserving the nondeterministic behavior of P_1 by the slice P_2:

Definition 3.9 (Nondeterministic behavior). For any $n \in nodes(C)$, and any $x \in C(n)$, let $V_i(n, x)$ denote the set of possible values for x computed by P_i at n, for $i = 1, 2$. P_2 *preserves the nondeterministic behavior of* P_1 *at* n iff it holds, for all $x \in C(n)$, that $V_1(n, x) \subseteq V_2(n, x)$.

Note that we require $V_1(n, x) \subseteq V_2(n, x)$ in the above definition since a nontermination insensitive slice, which does not preserve nontermination, may execute n more often than the original program, and thus a nontermination insensitively semi-equivalent slice may compute a value $v \in V_2(n, x)$ which is not present in $V_1(n, x)$.

Theorem 3.10. *A slice* P_2 *preserves the nondeterministic behavior of a nondeterministic program* P_1 *at all nodes* $n \in nodes(C)$ *if* $P_1 \gtrsim_C P_2$ *holds.*

Proof. Suppose P_1 and P_2 start execution from the initial configurations Γ_0 and Γ'_0 such that $\sigma_0 = store(\Gamma_0)$, $\sigma'_0 = store(\Gamma'_0)$, and $\sigma_0 =_{var(\sigma'_0)} \sigma'_0$ holds. Because of nondeterministic choice(s), starting from the initial configuration Γ_0, P_1 may either have (A) multiple finite or infinite C-derivation sequences visiting a node $n \in nodes(C)$, or (B) no C-derivation sequence exists due to having infinite loop(s) such that node n is unreachable.

Case (A): Suppose $\Gamma^i_0, \Gamma^i_1, \ldots, \Gamma^i_{k_i}$ are the finite truncated C-derivation sequences of P_1 for all $0 \leq i \leq MAX$ and $k_i \geq 1$ such that $\Gamma^i_0 = \Gamma_0$ and $n = node(\Gamma^i_{k_i})$. $MAX = \infty$ if n can be visited infinitely many times during the operational transition, otherwise, MAX is a finite number. The derivation sequence is truncated at a point where node n is visited. For any $x \in C(n)$, if V_1 is the set of possible values of x at n when P_1 executes from an initial configuration Γ_0, then, for any $v \in V_1$, there exists $i \geq 0$ and a configuration $\Gamma^i_{k_i}$ such that $v = store(\Gamma^i_{k_i})(x)$. According to Def. 3.7, for all C-derivation sequences $\Gamma^i_0, \Gamma^i_1, \ldots, \Gamma^i_k$ of P_1, there exists a C-derivation sequence $\Gamma'_0, \Gamma'_1, \ldots, \Gamma'_k$ of P_2 such that $node(\Gamma^i_j) = node(\Gamma'_j)$ and $store(\Gamma^i_j) =_{C(node(\Gamma'_j))} store(\Gamma'_j)$ for all $0 \leq j \leq k$. So, we have that $v = store(\Gamma'_k)(x)$. Thus, if V_2 is the set of possible values of x at n when P_2 executes from an initial configuration Γ'_0, for any value $v \in V_1$ of x, we have that $v \in V_2$ for all $x \in C(n)$ and for all $n \in nodes(C)$.

Case (B): As no C-derivation sequence exists visiting node n, for any $x \in C(n)$, its set of possible values $V_1 = \emptyset$, and $V_1 \subseteq V_2$ trivially holds. P_2 thus preserves the nondeterministic behavior of P_1 at all nodes $n \in nodes(C)$ for all $x \in C(n)$. □

Since the relation $P_1 \gtrsim_C P_2$ also holds for nontermination sensitively semi-equivalent slices, these slices also preserve the nondeterministic behavior of the original program.

Theorem 3.8 states that a deterministic semi-equivalent slice preserves the termination behavior of the original program. If the original program terminates, then a semi-equivalent deterministic slice also terminates. However, Theorem 3.8 cannot be extended to nondeterministic semi-equivalent slices. Let us explain the reason with an example. Consider the program Q_{10} in Fig. 9 such that the slicing criterion consists of the variables X and Z at program point L_1. Program Q_{10} is terminating as the nondeterministic **if-else** statement is terminating. The **while** loop in program Q_{11} is not terminating when $Y > 0$. But, Q_{11} is a valid nontermination insensitively semi-equivalent slice of Q_{10}. This is because for every finite C-derivation sequence $\Gamma_{start}, \Gamma_{L_1}, \Gamma_{end}$ obtained from the

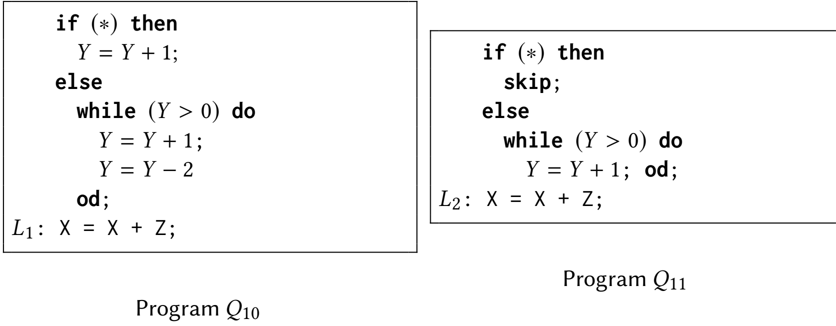

Fig. 9. The slicing criterion consists of the variables X and Z at program point L_1. Q_{11} is a semi-equivalent nontermination insensitive slice of Q_{10}.

CFG of Q_{10}, there exists a finite C-derivation sequence $\Gamma'_{start}, \Gamma'_{L_2}, \Gamma'_{end}$ obtained from the CFG of Q_{11} such that $store(\Gamma_{L_1}) = store(\Gamma'_{L_2})$ and $store(\Gamma_i) = store(\Gamma'_i)$ for $i \in \{start, end\}$. Even though the **else**-branch of Q_{11} is not terminating when $Y > 0$, the existence of the terminating **if**-branch at the nondeterministic choice is enough to prove the existence of the finite C-derivation sequence $\Gamma'_{start}, \Gamma'_{L_2}, \Gamma'_{end}$. If we modify the program Q_{11} in which the nonterminating loop is replaced by the **skip** statement, then the modified Q_{11} is also a valid nontermination insensitively semi-equivalent slice because such a nonterminating loop affects neither the slicing criterion nor the C-derivation sequence.

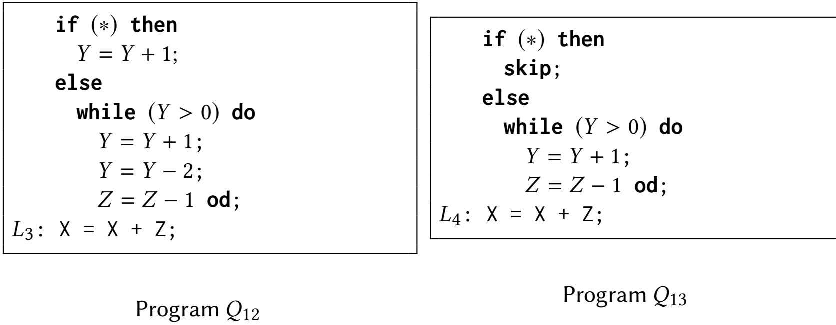

Fig. 10. The slicing criterion consists of the variables X and Z at program point L_3. Q_{13} is not a semi-equivalent nontermination insensitive slice of Q_{12}.

However program Q_{13}, which may not be terminating, is not a valid nontermination insensitively semi-equivalent slice of the terminating program Q_{12}. If the initial store of Q_{12} is $\{x \mapsto x_0, Y \mapsto 1, Z \mapsto z_0\}$, then we may get the C-derivation sequence $\Gamma_{start}, \Gamma_{L_3}, \Gamma_{end}$ of Q_{12} such that $store(\Gamma_{L_3}) = \{x \mapsto x_0, Y \mapsto 0, Z \mapsto z_0 - 1\}$. But no equivalent C-derivation sequence is possible for Q_{13} due to the nonterminating loop in the **else**-branch of the nondeterministic **if-else** statement. The existence of the terminating **if**-branch in Q_{13} is enough to form the finite C-derivation sequence $\Gamma'_{start}, \Gamma'_{L_4}, \Gamma'_{end}$, but not enough to ensure that $store(\Gamma_{L_3})(Z) = store(\Gamma'_{L_4})(Z)$.

Figs. 9 and 10 illustrate the boundary between allowing and not allowing nontermination in a nontermination (in)sensitively semi-equivalent nondeterministic slice P_2 of a terminating program P_1. In these examples, a nonterminating loop L_p is allowed in P_2 if there exists a nondeterministic choice such that one of the branches is terminating, the loop L_p appears in one of the other nondeterministic branches, and the computation in L_p does not affect the slicing criterion.

4 INTERPROCEDURAL DEPENDENCE-BASED SLICING

In this section we discuss the dependence-based slicing for interprocedural programs. Interprocedural slicing should take into account the context information in order to be precise. In Section 4.1, we define *valid contexts*, *derivation of contexts*, and *contextually valid paths*. Derivation of contexts is used to obtain the valid context of a node from the valid context of another node when there exists a path between them. Contextually valid paths are feasible paths that can be followed in an execution of an interprocedural program. Interprocedural data dependency is discussed in Section 4.2, and then interprocedural slicing is discussed using the concepts of *relevant variables* (Section 4.3), and *observable behavior* (Section 4.4). These are the properties of CFG nodes with respect to the slice set, which will be used later to prove the correctness of dependence-based slicing. We define the construction of the sliced code $code_2$ from $code_1$ and S_C according to [2] in Section 4.5.

4.1 Calling Contexts

Since we work in an interprocedural setting, we need the concept of *calling contexts*. Let Lab denote the set of labels of all the procedure calls of the given program. Then, $\Delta = Lab^*$ is the set of calling contexts of procedures. Let $\epsilon \in \Delta$ denote the empty context and let $\delta \circ d$ denote the concatenation of the context δ with the label $d \in Lab$.

Procedure calls can be recursive, and nested to any depth. The calling context describes exactly how a particular procedure is reached. For example, consider these two scenarios: (1) procedure P calls procedure Q from node n_P in P and then Q calls procedure R from node n_Q in Q, and (2) P calls R from another node n'_P. The calling context of R is then $\ell(n_P) \circ \ell(n_Q)$ in the first case, and $\ell(n'_p)$ in the latter case.

In general, Δ will also contain contexts that cannot be reached during any traversal of the CFG. A *valid* context describes a nesting of procedure calls that is reachable according to the structure of the CFG:

Definition 4.1 (Valid context).

(1) ϵ is a valid context for the *Start* node $start$.
(2) For any *Call* node m and its successor *CAssign* node n, or any *Ret node* m and its predecessor *RAssign* node n, $\delta' \circ \ell(m)$ is a valid context of n iff $\delta' \in \Delta$ is a valid context of m.
(3) For any RAssign node n and its predecessor Exit node m, $\delta \circ d$ is a valid context of n iff $\delta \circ d$ is a valid context of m and n has a successor n' such that $d = \ell(n')$.
(4) For any node n that is not a *CAssign*, *Ret*, or RAssign node, and any predecessor n' to n, if δ is a valid context of n' then so it is for n.

It is sometimes desirable to know the *relative* context δ' of a node n' reachable from another node n visited at a specific context δ. For example, consider the code and its CFG G in Fig. 4. Node n_{13} can be visited in two possible contexts: L_1 and L_2. If n_{13} is reached from n_3, which has context ϵ, then n_{13} is in the L_1 context, and if it is reached from, say, n_7 then it must be in the L_2 context. Knowing the relative context is also useful in determining the valid path during a walk of the CFG. For example, if n_X in G is being visited in the L_1 context, then the next visited node should be n_5, not n_{10}.

The function $\mathfrak{d}([n..n'], \delta)$ infers the context of n' relative to the context δ of n by walking through the path $[n..n']$:

Definition 4.2 (Derivation of Context ($\mathfrak{d}$)). For any nonempty path π and context δ, $\mathfrak{d}(\pi, \delta)$ is defined by::

$$\begin{array}{rcl} \mathfrak{d}([n], \delta) & = & \delta \\ \mathfrak{d}([n_1, n_2], \delta) & = & \delta \circ \ell(n_1) \quad \text{if } n_1 : \mathit{Call} \\ \mathfrak{d}([n_1, n_2], \delta) & = & \delta_1 \quad \text{if } n_2 : \mathit{Ret} \text{ and } \delta = \delta_1 \circ \ell(n_2) \\ \mathfrak{d}([n_1, n_2], \delta) & = & \delta \quad \text{otherwise} \\ \mathfrak{d}([n_1..n_k], \delta) & = & \mathfrak{d}([n_2..n_k], \mathfrak{d}([n_1, n_2], \delta)) \quad \text{if } k > 2 \end{array}$$

Example 4.3. Consider the CFG G in Fig. 4. We obtain

$$\begin{array}{rcll} \mathfrak{d}([n_3..n_{13}], \epsilon) & = & \mathfrak{d}([n_4..n_{13}], \mathfrak{d}([n_3, n_4], \epsilon)) & \\ & = & \mathfrak{d}([n_4..n_{13}], L_1) & (\text{since } \mathfrak{d}([n_3, n_4], \epsilon) = L_1 \text{ (Def. 4.2)}) \\ & = & \mathfrak{d}([n_E..n_{13}], \mathfrak{d}([n_4, n_E], L_1)) & \\ & & \vdots & \\ & = & L_1 & \end{array}$$

Note that $\mathfrak{d}$ uniquely determines the context for a configuration Γ, given that the initial configuration is assigned the empty context: regardless of how the execution reaches Γ, the contexts for the different possible paths will be the same. We omit the details. Thus, we will sometimes refer to "the context δ of the configuration Γ".

A *contextually valid path* is a valid path that also is consistent with respect to the possible valid contexts that can arise during a traversal of the path:

Definition 4.4 (Contextually valid path). Let n_1 be a CFG node and let δ be a valid context of n_1. A valid path $[n_1..n_k]$ is *contextually valid from n_1 in δ* if $\delta_i = \mathfrak{d}([n_1..n_i], \delta)$ is a valid context of node n_i for all $2 \leq i \leq k$.

Consider Fig. 4. L_2 is not a valid context of node n_4 and n_5, and L_1 is not a valid context of node n_9 and n_{10}. The derivation function $\mathfrak{d}$ in Def. 4.2 does not always provide valid contexts even if we consider valid paths. For example, the path $\pi = [n_{16}..n_6]$ is valid as its C-extension $[n_3, n_4].\pi$ is balanced. Nevertheless, $L_2 = \mathfrak{d}([n_{16}..n_6], L_2)$ is not a valid context of n_6 as the successor of n_X should be n_{10} instead of n_5 in the L_2 context. This is due to the fact that even though π is a valid path, it is a contextually invalid path in this context. However, the path π is contextually valid from n_{16} in the L_1 context.

Sometimes we shall write (n, δ) to denote a CFG node in context δ. Also, we shall represent a finite contextually valid path $\pi = [n_1..n_k]$ from node n_1 in context δ_1 by the sequence of (node,context)-pairs $\hat{\pi} = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$, where $\delta_i = \mathfrak{d}([n_1..n_i], \delta_1)$ for $2 \leq i \leq k$, and we write $(n_i, \delta_i) \in \hat{\pi}$. We will sometimes refer to $\hat{\pi}$ as a contextually valid path, when the meaning is clear from the context, although strictly speaking it is not a CFG path. We say that a contextually valid path $\hat{\pi}$ is *maximal* if π is a maximal path.

4.2 Interprocedural Data Dependency

We say that a procedure P *declares* a variable if there exists a node n containing an assignment to that variable, and one of the following holds: (i) n is an *Assign* node that belongs to the CFG of P, (ii) n is a *CAssign* node, and its successor is the *Entry* node of P, or (iii) n is a *RAssign* node, and its successor (the Ret node) belongs to the CFG of P. Since our scoping rules do not prohibit using the same name for variables in different procedures, it is sometimes important to know if a CFG node is in the scope of a variable in some given context. In the following we define the scope of a variable, in some context, as a set of (node,context) pairs:

Definition 4.5 (Scope of variables). Let v be a program variable declared in a procedure P in some valid context δ. Then, all (n, δ') specified below are in the *scope* of (v, δ):

(1) n belongs to the CFG of procedure P, and $\delta' = \delta$,
(2) either n is a CAssign node which is a predecessor of the Entry node of the CFG of P, and $\delta' = \delta$, or n is a RAssign node and $\delta' = \delta \circ \ell(n')$ where $n' \in succ(n)$ is a Ret node that belongs to the CFG of P, and
(3) either n is a RAssign node which is a successor of the Exit node of the CFG of P, and $\delta' = \delta$, or n is a CAssign node and $\delta' = \delta \circ \ell(n')$ where $n' \in pred(n)$ is a Call node that belongs to the CFG of P.

All (n, δ') pairs that fulfil (1) and (2) above are in the *definition scope* of (v, δ), and those fulfilling (1) and (3) are in the *reference scope* of (v, δ).

Note that we have to differ between definition and reference scope. This is since for CAssign nodes the assigned variables are visible in the callee, and the referenced variables in the caller, and vice versa for RAssign nodes.

We now have the concepts needed to redefine Def. 2.7 to cover also the interprocedural case:

Definition 4.6 (Interprocedural Data Dependency). Node n_2 is *data dependent* on node n_1 (written $n_1 \overset{dd}{\rightarrow} n_2$) in the CFG G if there exists a non-trivial, contextually valid path $\hat{\pi} = [(n_1, \delta_1)..(n_2, \delta_2)]$ for some (n_1, δ_1) and (n_2, δ_2) such that (1) there is a program variable $v \in def(n_1) \cap ref(n_2)$ declared in some valid context δ, and (2) for every $(m, \delta') \in \hat{\pi} - \{(n_1, \delta_1), (n_2, \delta_2)\}$, it holds that $v \notin def(m)$ when (m, δ') is in the definition scope of (v, δ).

With this definition of $\overset{dd}{\rightarrow}$, Def. 2.11 of slice sets becomes extended to the interprocedural case. The sole difference to Def. 2.7 is that only contextually valid paths and scope of variables are considered: this yields a considerably more precise definition of data dependency. For example, let us replace the code `print(c)` by `print(x)` at node n_{12} in Fig. 4 which is a valid change since we allow different procedures to use same variable names. We obtain $n_{16} \overset{dd}{\rightarrow} n_{12}$ according to Def. 2.7. However, this relation is not valid according to Def. 4.6 since the variable $x \in def(n_{16}) \cap ref(n_{12})$ are declared in different contexts violating condition (1) in Def. 4.6. Thus, even though (n_{12}, ϵ) is in the reference scope of (x, ϵ), no valid context δ of n_{16} exists such that (n_{16}, δ) is in the definition scope of (x, ϵ).

4.3 Relevant Variables in an Interprocedural Context

The concept of *relevant variables* (RVs) is used by different theories and techniques of slicing. It is a fundamental idea for understanding how the values of variables in different program points may affect the values of variables specified in the slicing criterion. Informally speaking, a relevant variable is a variable whose value in some node and context may affect the value of some variable in the slicing criterion.

Weiser [49] provided a recursive definition of relevant variables in an intraprocedural setting. We now define them in an interprocedural setting. Without loss of generality, precision, and correctness, we assume that $ref(n) \neq \emptyset$ for any CFG node n. This can be ensured by assuming a dummy variable x_p for each procedure p that is not defined or referenced anywhere in p and that only has scope in procedure p. We set $ref(n) = \{x_p\}$ for any node n not referencing any variable (e.g. Entry or Exit node, Call or Ret node, Cond node with condition true/false, or assignment statement like $x = 10$ etc.). Assume that **Var** includes x_p, its default value is $v_p \in$ **Val**, and we modify the ParamIn rule in Section 3.1 to include the mapping $x_p \mapsto v_p$ as follows:

$$\frac{code_i(n) = \; x_1 = e_1, \ldots x_k = e_k \quad n : CAssign \quad n' \in succ(n) \quad p = proc(n') \quad v_j = [\![e_j]\!]\sigma}{i \vdash \Gamma \cdot (n, \sigma) \rightarrow \Gamma \cdot (n', \{x_1 \mapsto v_1, \ldots, x_k \mapsto v_k, x_p \mapsto v_p\})} (\textsc{ParamIn})$$

where, n' belongs to the procedure $proc(n')$. Since x_p is not defined anywhere in the program code, it creates no additional dependencies and it does not affect the slicing criterion. However, this assumption simplifies our definition of RVs because we can always assume that the RV set of any node n in context δ is non-empty (or empty) if n in δ may affect (resp. does not affect) the slicing criterion. We provide the following definition of RVs:

Definition 4.7 (Relevant Variables). The set of *relevant variables* $rv^G(n, \delta)$ at node n of CFG G in context δ consists of the variables v for which some of the following holds:

(1) $v \in C(n)$ and $n \in nodes(C)$.
(2) there exists a contextually valid path $[\![(n_1 = n, \delta_1 = \delta)..(n_k, \delta_k)]\!]$ with $n_k \in S_C$ such that
  - (n, δ) and (n_k, δ_k) are in the reference scope of a program variable $v \in rv^G(n_k, \delta_k)$ declared in some valid context δ', and
  - for all $1 \leq i \leq k-1$ it holds that $v \notin def(n_i)$ whenever (n_i, δ_i) is in the definition scope of (v, δ').
(3) $v \in ref(n)$ and $n \in S_C$ where v may influence the value of another RV in one of the following ways:
  (a) $def(n) \cap rv^G(m, \delta_m) \neq \emptyset$ for some $m \in succ(n)$ and $\delta_m = \mathfrak{d}([n, m], \delta)$.
  (b) the relation $n \xrightarrow{cd} m$ holds such that $m \in S_C$, $\delta_m = \mathfrak{d}([n..m], \delta)$ and $rv^G(m, \delta_m) \neq \emptyset$.

Def. 4.7 inductively identifies the RVs by visiting the CFG nodes in the backward direction from the nodes in the slicing criterion. The first sets of RVs for nodes in the slicing criterion are identified from case (1), the CFG is traversed in the backward direction, RVs identified in case (1) are included in the RV sets of visited nodes due to case (2), and new RVs are generated in cases (3a) and (3b). A similar inductive definition for RVs can be provided that visits the CFG in the forward direction.

Def. 4.7 is a nontrivial extension of the definition of relevant variables for intraprocedural programs. The definition of relevant variables [2, 50] for intraprocedural programs can be summarized as follows. Variable v is a RV at a CFG node n (denoted $v \in rv(n)$) if one of the following holds:

A1. $v \in C(n)$ and $n \in nodes(C)$,
A2. there exists $n_k \in S_C$ and a path $[n_1..n_k]$ with $n_1 = n$ such that $v \in rv(n_k)$, but $v \notin def(n_i)$ for all $1 \leq i \leq k-1$,
A3. $v \in ref(n)$ and $def(n) \cap rv(m) \neq \emptyset$ for some $m \in succ(n)$,
A4. $v \in ref(n)$ and $n \xrightarrow{cd} m$ for some $m \in S_C$.

To see the differences in the definitions of interprocedural and intraprocedural RVs, let us compare the above conditions with the conditions in Def. 4.7. Conditions (A1) and (A3) are equivalent to the conditions (1) and (3a) in Def. 4.7. Condition (2) in Def. 4.7 is the refinement of condition (A2) by introducing contexts and scopes to make it suitable for interprocedural programs. It first requires that the path $[n_1..n_k]$ must be contextually valid. The condition $v \notin def(n_i)$ in (A2) is only enforced in condition (2) if n_i is in the definition scope of (v, δ') where v is declared in δ'. This is because node n_i may still define another variable with the same name if n_i and n_k belong to different procedures or n_i may belong to the CFG of a recursive procedure. However, the particular instance of the RV v at n_k cannot be defined at n_i if n_i is outside the definition scope of (v, δ'). Condition (2) additionally requires that both (n, δ) and (n_k, δ_k) are in the reference scope of (v, δ').

In order to illustrate these additional requirements, consider the set of RVs listed in Fig. 11 for the CFG in Fig. 4. Details of how we obtain the RVs are explained shortly in example 4.8 below. The set of RVs at n_8 in the ϵ context is $\{a, b\}$. These variables are declared also in the ϵ context. Now, according to the intraprocedural definition of RVs, a and b are also RVs at node n_5 in L_1 context since there exists a path $[n_5..n_8]$ with $a, b \notin def(n_i)$ for $i = 5, 6, 7$ even though (n_5, L_1) is not in the reference scope of (v, ϵ) for $v \in \{a, b\}$. This would be problematic if, for instance, the procedure `add` would contain local variables a, b. However, Def. 4.7 correctly infers that $rv(n_5, L_1) = \emptyset$.

Indirect relevant variables are identified due to condition (A4) for intraprocedural RVs, or case (3b) in Def. 4.7 for interprocedural RVs[2]. So, case (3b) is similar to (A4) with the additional requirement that $rv^G(m, \delta_m) \neq \emptyset$. In the following, we illustrate the necessity of this additional constraint in the interprocedural case.

Necessity of $\mathbf{rv^G(m, \delta_m) \neq \emptyset}$ for $\delta_m = \mathfrak{d}([\mathbf{n}..\mathbf{m}], \delta)$: In an interprocedural setting, a procedure P may have multiple contexts due to different calls to it. There may exist calling contexts δ_m and δ'_m such that a CFG node $m \in S_C$ in P affects the slicing criterion in δ_m, but not in δ'_m. For example, n_{15} in Fig. 4(b) affects the slicing criterion in the L_2 context, but not in the L_1 context. $rv^G(m, \delta_m) \neq \emptyset$ if $ref(m) \neq \emptyset$ and m affects the slicing criterion in δ_m due to one of the following reasons:

- the slicing criterion is affected by the value of a variable in $def(m)$, and so $rv^G(m, \delta_m)$ includes the variables in $ref(m)$,
- the slicing criterion is affected by m in δ_m due to control dependency, and so $rv^G(m, \delta_m)$ includes the variables in $ref(m)$.

Since $ref(m)$ is never empty, due to the introduction of dummy variables, it is guaranteed that $rv^G(m, \delta_m) \neq \emptyset$ when m affects the slicing criterion in δ_m. If we drop the condition $rv^G(n_m, \delta_m) \neq \emptyset$ from case (3b) in Def. 4.7, the slicing procedure will infer RVs that do not affect the slicing criterion. For example, $rv^G(n_{15}, L_1) = \emptyset$, n_{15} does not affect the slicing criterion in L_1 context, but will infer that $x \in ref(n_{14})$ is a RV at n_{14} in L_1. The far-reaching effect will be that it will infer p to be a RV at n_4 and slice n_4. Ultimately, we shall loose precision by having a larger slice set S_C than necessary. Thus, the condition $rv^G(n_m, \delta_m) \neq \emptyset$ in case (3b) restricts the contexts for considering $ref(n)$ as indirect RVs. If this condition holds then m affects the slicing criterion in δ_m, n indirectly affects the slicing criterion in δ due to the relation $n \xrightarrow{cd} m$, and the variables in $ref(n)$ become RVs at n in δ.

Example 4.8. We illustrate the different cases in Def. 4.7 with the RVs listed in Fig. 11 for the code and its CFG in Fig. 4.

- RVs due to (1). Variable c is a RV at node n_{12} in the ϵ context due to case (1).
- RVs due to (2). Since c is a RV in $rv^G(n_{12}, \epsilon)$, $[\![(n_{11}, \epsilon), (n_{12}, \epsilon)]\!]$ is a contextually valid path, both (n_{11}, ϵ) and (n_{12}, ϵ) are in the reference scope of (c, ϵ), and n_{11} is not defining c, c is also a RV in $rv^G(n_{11}, \epsilon)$ due to case (2). Variables a, b are RVs at node n_9 in the L_2 context (due to case (3a) illustrated below). Case (2) shows that these variables are RVs at all nodes in the path $[n_6..n_8]$ (in the ϵ context), but not relevant at nodes $n_4, n_5, n_{13}, \ldots, n_{16}$ (in the L_1 context) since these nodes are not in the reference scope of (v, L_1) for $v \in \{a, b\}$. However, since (n_1, ϵ) and (n_3, ϵ) are in the reference scope (v, ϵ) for $v \in \{a, b\}$, a, b become relevant again at n_1 and n_3 (in the ϵ context).
- RVs due to (3a). Since c is a RV at n_{11} in the ϵ context and it is defined at n_{10}, the condition $def(n_{10}) \cap rv^G(n_{11}, \epsilon) \neq \emptyset$ in (3a) of Def. 4.7 holds. So, $z \in ref(n_{10})$ becomes a RV at n_{10} in the L_2 context. Similarly, we obtain the RVs y, z at node n_{15} and a, b at n_9 in L_2 context, and a_0, b_0 at n_0 in ϵ context.
- RVs due to (3b). The condition $x \geq 0$ at node n_{14} indirectly affects the computation at node n_{15}. Thus, y, z remain relevant at n_{14} because of case (2), and x is also a RV at n_{14} due to case (3b) in the L_2 context. Note that the condition $rv^G(n_{15}, L_2) \neq \emptyset$ (an instantiation of $rv^G(n_k, \delta_k) \neq \emptyset$ in (3b)) is important for x to become a RV. x is not a RV at n_{14} in the L_1 context as $rv^G(n_{15}, L_1) = \emptyset$.

[2]There exist formal definitions of RVs (for example, in [2]) which do not include indirect relevant variables. Their effects are rather captured implicitly by data and control dependencies.

Node n	Context δ	RVs		$obs(n,\delta)$
		$rv(n,\delta)$	$Case$	
n_{12}	ϵ	$\{c\}$	(1)	$\{n_{12}\}$
n_{11}	ϵ	$\{c\}$	(2)	$\{n_{11}\}$
n_{10}	L_2	$\{z\}$	$(3a)$	$\{n_{10}\}$
n_X, n_{16}	L_2	$\{z\}$	(2)	$\{n_X\}$
n_{15}	L_2	$\{y,z\}$	$(3a)$	$\{n_{15}\}$
n_{14}	L_2	$\{y,z$	(2)	$\{n_{14}\}$
		$x\}$	$(3b)$	
n_{13}	L_2	$\{x,y\}$	$(2,3a)$	$\{n_{13}\}$
n_E	L_2	$\{x,y\}$	(2)	$\{n_E\}$
n_9	L_2	$\{a,b\}$	$(3a)$	$\{n_9\}$
n_1, n_3, n_6, n_7, n_8	ϵ	$\{a,b\}$	(2)	$\{n_8\}$
n_0	ϵ	$\{a_0, b_0\}$	$(3a)$	$\{n_0\}$
n_2	ϵ	$\emptyset$	()	$\emptyset$
$n_4, n_5, n_{13}, \ldots, n_{16}, n_E, n_X$	L_1	$\emptyset$	()	$\{n_8\}$

Fig. 11. RVs and the observable set $obs(n,\delta)$ for all CFG nodes n and contexts δ for the CFG G in Fig. 4

4.4 Observable behavior

In sequences of configurations there are two kinds of visits to CFG nodes: if $n \in S_C$ is visited, affecting the slicing criterion, then it is called an S_C-*observable* move, and otherwise it is a *silent* move. S_C-observable moves observe values of variables at statements belonging to the sliced program. If an original program and its slice have the same S_C-observable moves, observing the same values, then they will have the same C-observable moves and will observe the same values at nodes in $nodes(C)$. The *next observable behavior* defined below will be used to define the sliced program and to prove the correctness of dependence-based slicing.

Definition 4.9 (Next Observable Behavior). Let n be a node in CFG G at context δ, and let S_C be a slice set. The set of *observable nodes* $obs^G_{S_C}(n,\delta)$ contains all nodes $m \in S_C$ such that there exists a contextually valid path $[\![(n_1,\delta_1)..(n_k,\delta_k)]\!]$ with $(n_1,\delta_1) = (n,\delta)$, $n_k = m$, $rv^G(n_k,\delta_k) \neq \emptyset$, and the following properties hold for each n_i for $1 \leq i \leq k-1$:

(1) if $\delta_i = \delta_k$, then $n_i \notin S_C$ and $rv^G(n_i,\delta_i) = rv^G(n_k,\delta_k)$, and
(2) if $\delta_i \neq \delta_k$, then $rv^G(n_i,\delta_i) = \emptyset$.

We write $obs(n,\delta)$ and $rv(n,\delta)$ for $obs^G_{S_C}(n,\delta)$ and $rv^G(n,\delta)$, respectively, if G and S_C are understood from the context. Intuitively, $obs(n,\delta)$ contains all reachable nodes $n_k \in S_C$ from n such that n_k affects the slicing criterion in the δ_k context. In order to ensure that node n_k affects the slicing criterion in δ_k, as illustrated by the cases of interprocedural RVs in Section 4.3, we require that $rv(n_k,\delta_k) \neq \emptyset$. Moreover, we also require that no other node n_i in the path $[n..n_{k-1}]$ affects the slicing criterion in the δ_i context. This requires that either (i) n_i does not belong to the sliced code, and the set of RVs at node n_k is the same as at n_i when n_k and n_i are in the same context, or (ii) the set of RVs at node n_i is empty when n_k and n_i are in different contexts[3].

The sets of observable nodes $obs(n,\delta)$ for all CFG nodes n in the CFG G in Fig. 4 and their relevant contexts δ are listed in Fig. 11. In the following, we illustrate the necessity of condition (2) in Def. 4.9 (i.e. $rv(n_i,\delta_i) = \emptyset$ when $\delta_i \neq \delta_k$) in order for m to be included into the set of observable nodes. We obtain $obs(n_1,\epsilon) = \{n_8\}$ since $n_8 \in S_C$, $rv(n_8,\epsilon) \neq \emptyset$ and the following conditions hold:

[3]In the proofs of Lemmas 7.3 and 7.4 in Section 7.1, we show that $rv^G(n_i,\delta_i) = rv^G(n_k,\delta_k)$ if $\delta_i = \delta_k$, and $rv^G(n_i,\delta_i) = \emptyset$ if $\delta_i \neq \delta_k$.

- $rv(n_8, \epsilon) = rv(n, \epsilon)$ for all $n \in \{n_1, n_3, n_6, n_7\}$ satisfying condition (1) of Def. 4.9, and
- $rv(n, L_1) = \emptyset$ for all $n \in \{n_4, n_5, n_E, n_X, n_{13}, \ldots, n_{16}\}$ satisfying condition (2) of Def. 4.9.

However, the observable set $obs(n_1, \epsilon)$ does not include n_{13} even though $n_{13} \in S_C$. As $rv(n_{13}, L_1) = \emptyset$, n_{13} does not affect the slicing criterion in L_1, there does not exist any observable variable at n_{13}, and n_{13} itself is not observable in the L_1 context. Similarly, no node in $\{n_E, n_X, n_{13}, \ldots, n_{15}\}$ is observable in L_1 even though these nodes belong to S_C: this is since these nodes are included into S_C as they affect (do not affect) the slicing criterion in L_2 (L_1) context, which leads to the condition $rv(n, L_1) = \emptyset$. In order to illustrate further the set of observable nodes at different contexts, we have $obs(n_{13}, L_1) = \{n_8\}$ according to Def. 4.9, but $obs(n_{13}, L_2) = \{n_{13}\}$ trivially holds.

4.5 Sliced Code

Let the slice set S_C be defined according to Def. 2.11. For any CAssign/RAssign node n, let T_n be a function that takes an assignment $x = e$ as input and returns either $x = e$ or an empty sequence ϵ as follows:

$$T_n(x = e) \quad = \quad \begin{cases} x = e & \text{if } x \in rv(m, \delta) \text{ for any valid context } \delta \text{ of } m \in succ(n), \\ \epsilon & \text{otherwise} \end{cases}$$

The following definition of the mapping function $code_2$ (adapted from [2]) provides the sliced code:

Definition 4.10 (Mapping Function $code_2$). The function $code_2(n)$ is obtained from the function $code_1(n)$ and the slice set S_C as follows:

(1) if $n \in S_C$ and n is not a CAssign/RAssign node, then $code_2(n) = code_1(n)$;
(2) If $n \in S_C$, n is a CAssign/RAssign node, and $code_1(n) = (x_1 = e_1, \ldots, x_k = e_k)$, then

$$code_2(n) = T_n(x_1 = e_1), \ldots, T_n(x_k = e_k);$$

(3) if $n \notin S_C$ and n is not a Cond node, then $code_2(n) = \textbf{skip}$;
(4) If $n \notin S_C$, n is a Cond node, and
  (a) $\exists \delta, m. m \in obs(n, \delta) \wedge dist(tsucc(n), m) < dist(n, m)$, then $code_2(n) = \textbf{true}$;
  (b) otherwise, $code_2(n) = \textbf{false}$.

The first and the third cases ensure that the slice does not modify the original code if $n \in S_C$ and n is not a CAssign, RAssign or Cond node. The second case may partially slice the relevant parameters when $n \in S_C$ is a CAssign or RAssign node. If n contains an assignment $x = e$ such that x is a RV in $rv(m, \delta)$ for any successor node $m \in succ(n)$ of n and its valid context δ, then the sliced code contains this assignment since another node in S_C then is data dependent on node n. Otherwise, the assignment is replaced by an empty sequence ϵ that basically removes this assignment. In the last case the conditional node n is not included in the slice set S_C. If the true branch is closer to the observable node m, then $code_2(n)$ is set to **true**; if not, or the observable set $obs(n, \delta)$ is empty for all relevant contexts δ, $code_2(n)$ is set to **false**. We may nondeterministically set $code_2(n)$ to be **true** or **false** if $|obs(n, \delta)| > 1$. The sliced code P_2 obtained through $code_2$ thus has the same CFG structure as the original code P_1, but with nodes outside the slice "blanked out" by assigning them program statements that do not alter the stores.

Note that Def. 4.10 provides a unique slice P_2 of the original code P_1 after computing the slice set S_C. The slice set S_C is computed according to Def. 2.11 where the transitive-reflexive closure relation can be obtained by traversing the system dependence graph representation of the program. Alternatively, we obtain the slice set by traversing the CFG of the program backward (for backward slicing) from the nodes in $nodes(C)$ and computing RVs according to Def. 4.7. For any node $n \in nodes(C)$, $C(n)$ is the set of RVs at n for all valid contexts of n (Case (1) in Def. 4.7), and n is included in S_C. RVs are propagated from any $n \in S_C$ to its ancestor nodes m such that there exists a path $[m..n]$ according to Case (2) in Def. 4.7 during the backward traversal of the CFG.

Whenever Case (3a) or (3b) in Def. 4.7 are applied to generate new RVs during visiting a CFG node m, node m is included in S_C. For example, consider the CFG G in Fig. 4, $C = \{(n_{12}, \{c\})\}$, and the set of RVs listed in Fig. 11. The slice set S_C includes all nodes n listed in Fig. 11 for which Cases (1), (3a), or (3b) are applicable. Thus, S_C includes the nodes $n_0, n_9, n_{10}, n_{12}, n_{13}, n_{14}, n_{15}$ according to Fig. 11. Note that nodes n_8, n_{11}, n_E, and n_X should also be included in S_C due to well-formedness of S_C as explained in the next section.

During the computation of S_C, the CFG nodes of a procedure may be traversed multiple times from multiple call sites, and the CFG nodes of that procedure are included in S_C whenever Cases (1), (3a), or (3b) in Def. 4.7 are applicable. The number of times to traverse a procedure depends on the slicing criterion and the call sites. For example, consider the CFG G in Fig. 4, $C = \{(n_{12}, \{c\})\}$, and the set of RVs listed in Fig. 11. As $rv(n_5, L_1) = \emptyset$, no statement in the `add` procedure will affect n_5 in the L_1 context. This illustrates the empty RVs at the CFG nodes of the `add` procedure in L_1 context as shown in Fig. 11. Thus, it will be more efficient to traverse the CFG node n_4 instead of n_X from n_5 as visiting n_X will not provide any new dependencies. However, if we change the statement of n_{12} in Fig. 4 to **`print`**`(c,e)`, and change the slicing criterion accordingly to $C = \{(n_{12}, \{c, e\})\}$, then we obtain $rv(n_5, L_1) = \{x\}$ according to Def. 4.7. Traversing the CFG nodes of the `add` procedure for the second time will additionally include nodes n_{16} and n_4 in S_C due to Case (3a) in Def. 4.7.

For the CFG G, $C = \{(n_{12}, \{c\})\}$, and the set S_C in Fig. 4, we obtain $code_2(G)$ according to Def. 4.10 as follows:

- $code_2(n) =$ **`skip`** for any $n \in \{n_3, n_4, n_5, n_6, n_7, n_{16}\}$,
- $code_2(n_1) =$ **`true`** and $code_2(n_2) =$ **`false`**,
- $code_2(n_{10}) = (c = z, \epsilon)$, and
- $code_2(n) = code_1(n)$ for all other nodes.

5 WELL-FORMED WEAKLY AND STRONGLY CONTROL-CLOSED SLICE SETS

Danicic et al. [13] provided two generalisations of nontermination insensitive and nontermination sensitive control dependence, called weak and strong control closure, for directed graphs that can represent intraprocedural programs. In this section we extend these generalisations to interprocedural programs, and we call them *well-formed weak/strong control-closure*. We show that well-formed weak/strong control-closure is a necessary and sufficient condition for the interprocedural slice to be a weak/strong projection, and hence non-termination insensitive/sensitive. In particular, well-formed weak and strong control-closure are properties of the CFG nodes in the interprocedural context such that the *well-formed weak/strong control-closed* set N' of any set $N'' \subseteq N'$ of CFG nodes captures all relevant control dependencies of the nodes in N'. Thus, if the slice set S_C is well-formed weak/strong control-closed, the interprocedural slice is a weak/strong projection of the original program, and hence the slice is a nontermination insensitive/sensitive slice of the original program according to Def. 2.12.

In what follows, let $G = (N, E)$ be a CFG, let $\mathrm{N}' \subseteq N$, and let Δ be the set of all calling contexts in G. Let N'_Δ be a set of pairs (n, δ) where $n \in \mathrm{N}'$, and $\delta \in \Delta$ is a valid context of n.

Definition 5.1 (N'_Δ-Path). A N'_Δ*-path* from node n_1 in a valid context $\delta_1 \in \Delta$ in a CFG G is a finite contextually valid path $[\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ in G such that $k > 1$, $(n_k, \delta_k) \in \mathrm{N}'_\Delta$, and $1 < i < k \implies (n_i, \delta_i) \notin \mathrm{N}'_\Delta$.

Note that (n_1, δ_1) may possibly be in N'_Δ in the above definition. We now define well-formed weakly and strongly control-closed sets. We need some auxiliary definitions to define these sets.

Definition 5.2 (N'_Δ-Weakly Committing Vertices). A node n in context δ in a CFG G is N'_Δ*-weakly committing* if all N'_Δ-paths from (n, δ) have the same end point. In other words, all finite contextually valid N'_Δ-paths from (n, δ) meet at a common node n_k in δ_k such that $(n_k, \delta_k) \in \mathrm{N}'_\Delta$.

Example 5.3. Let us consider Fig. 4. Let $N_1 = \{n_9, n_{10}, N_E, N_X, n_{13}, n_{14}, n_{15}\}$, and let $N_2 = \{n_0, n_8, n_{11}, n_{12}\}$. Assume that N'_Δ contains the pairs (n, δ) where $\delta = L_2$ for $n \in N_1$, and $\delta = \epsilon$ for $n \in N_2$.

- The contextually valid path $\hat{\pi} = [\![(n_{13}, L_1)..(n_8, \epsilon)]\!]$ is a N'_Δ-path since all $(n, \delta) \in \hat{\pi}$ except (n_8, ϵ) are not in N'_Δ.
- Since $[\![(n_{13}, L_2)..(n_8, \epsilon)]\!]$ is not a contextually valid path, it is not a N'_Δ-path.
- Since $(n_{14}, L_2) \in N'_\Delta$, $[\![(n_{13}, L_2), (n_{14}, L_2), (n_{15}, L_2)]\!]$ is not a N'_Δ-path.
- Node n_{13} in context L_1 is a N'_Δ-weakly committing vertex since all N'_Δ-paths from node n_{13} in L_1 context meet at node n_8 in ϵ context.

Definition 5.4 (Weakly Control-Closed Sets). The set N'_Δ is *weakly control-closed* in a CFG G if and only if all CFG nodes n in all valid contexts δ such that $(n, \delta) \notin N'_\Delta$ are N'_Δ-weakly committing in G when there exists a contextually valid path $[\![(n', \delta')..(n, \delta)]\!]$ such that $(n', \delta') \in N'_\Delta$.

Example 5.5. The N'_Δ set in Example 5.3 is weakly control-closed since any node n in all valid contexts δ such that $(n, \delta) \notin N'_\Delta$, but reachable by a contextually valid path from any $(n', \delta') \in N'_\Delta$, is N'_Δ-weakly committing in δ. For example, n_1 in ϵ is N'_Δ-weakly committing since there exists a contextually valid path $[\![(n_0, \epsilon)..(n_1, \epsilon)]\!]$ and there exists a single N'_Δ-path $[\![(n_1, \epsilon)..(n_8, \epsilon)]\!]$. Note that n_2 in ϵ is also N'_Δ-weakly committing since a contextually valid path $[\![(n_0, \epsilon)..(n_2, \epsilon)]\!]$ exists and no N'_Δ-path from n_2 in ϵ exists. However, if we exclude the element (n_{14}, L_2) from N'_Δ, then it is not weakly control-closed since node n_{14} in L_2 context is not N'_Δ-weakly committing. We have the contextually valid path $[\![(n_{13}, L_2), (n_{14}, L_2)]\!]$, but there exist two N'_Δ-paths $[\![(n_{14}, L_2), (n_{15}, L_2)]\!]$ and $[\![(n_{14}, L_2)..(N_X, L_2)]\!]$ that do not have the same end point.

In the following, we define strongly control-closed sets by using the auxiliary definitions of N'_Δ*-strongly committing vertices* and N'_Δ*-avoiding vertices*. Informally, a node n in a valid context δ is N'_Δ-strongly committing vertex if all N'_Δ paths from (n, δ) meet at a common node n_k in δ_k such that $(n_k, \delta_k) \in N'_\Delta$ (i.e. (n, δ) is N'_Δ-weakly committing) and all maximal (contextually valid) paths from n go through an element in N'_Δ. In such a case, node n in δ has a unique observable behavior due to the unique meeting point, and all nonterminating paths contain an observable behavior and thus preserve nontermination. Node n in a valid context δ is N'_Δ-avoiding if all maximal (contextually valid) paths do not contain an element in N'_Δ and hence n in δ has no observable behavior.

Definition 5.6 (N'_Δ-Strongly Committing Vertices). A node n in context δ in a CFG G is N'_Δ*-strongly committing* if and only if it is N'_Δ-weakly committing in G and all contextually valid maximal paths from (n, δ) contain an element (n', δ') such that $(n', \delta') \in N'_\Delta$.

Definition 5.7 (N'_Δ-Avoiding Vertices). A node n in context δ in a CFG G is N'_Δ*-avoiding* if and only if no N'_Δ-path exists from n in δ.

Definition 5.8 (Strongly Control-Closed Sets). The set N'_Δ is *strongly control-closed* in a CFG G if and only if for all $(n, \delta) \notin N'_\Delta$, for which there exists a contextually valid path $[(n', \delta')..(n, \delta)]$ from some $(n', \delta') \in N'_\Delta$, node n in context δ is either N'_Δ-strongly committing or N'_Δ-avoiding.

Example 5.9. Consider the N'_Δ set in Example 5.3. Node n_2 in ϵ context is N'_Δ-avoiding. Node n_1 in ϵ context is N'_Δ-weakly committing because $[\![(n_1, \epsilon)..(n_8, \epsilon)]\!]$ is the only N'_Δ-path. Node n_1 in ϵ context is neither N'_Δ-strongly committing nor N'_Δ-avoiding since the maximal path $[\![(n_1, \epsilon)..(n_{12}, \epsilon)]\!]$ contains an element from N'_Δ and the maximal contextually valid path $\hat{\pi} = (n_1, \epsilon), (n_2, \epsilon), \ldots$ does not contain an element from N'_Δ. N'_Δ is thus not a strongly control-closed set. However, $N'_\Delta \cup \{(n_1, \epsilon), (n_2, \epsilon)\}$ is a strongly control closed set.

Weakly and strongly control-closed sets may exclude certain kinds of nodes such as Call, Ret, Entry and Exit nodes as these nodes do not carry any data or control dependence. Thus, if we

compute the slice of a program by capturing the data dependence and using weak/strong control closure to capture the control dependence, then the CFG of the sliced code may not be a proper CFG. In order to show that the interprocedural slice is a weak/strong projection and thus it produces a semi-equivalence relation with the original program, it is important that the CFG of the slice is valid according to Def. 2.1. Thus, in the following, we define *well-formed* sets such that well-formed weakly and strongly control-closed sets are valid CFGs. Consider the CFG node n that either belongs to a procedure P in G or n is a CAssign (or RAssign) node such that the Entry (resp. Exit) node of P is the successor (resp. predecessor) of n. We write $entry(n)$ and $exit(n)$ for the Entry and Exit CFG nodes of P in G.

Definition 5.10 (Well-formed Sets). A *well-formed set* N'_Δ includes the following, for all $(n, \delta) \in \mathrm{N}'_\Delta$: (1) $(entry(n), \delta), (exit(n), \delta) \in \mathrm{N}'_\Delta$, and (ii) if $\delta = \delta' \circ \ell(m)$ where $\delta' \in \Delta$, then $(m, \delta') \in \mathrm{N}'_\Delta$.

For example, the N'_Δ set in Example 5.3 is a well-formed weakly control-closed set since N'_Δ includes the elements (N_E, L_2), (N_X, L_2), (n_8, ϵ) and (n_{11}, ϵ). However, Def. 5.4 and Def. 5.8 do not consider these elements to be included into N'_Δ.

Definition 5.11 (Δ-Augmented Slice Set S_{C_Δ}). Let S_C be a slice set of a CFG G, and let $rv(n, \delta)$ be the set of RVs at node n in context $\delta \in \Delta$ in G. The Δ-*augmented slice set* S_{C_Δ} of S_C is

$$S_{C_\Delta} = \{(n, \delta) : n \in S_C, \delta \in \Delta, rv(n, \delta) \neq \emptyset\}.$$

Definition 5.12 (Weakly and Strongly Control-Closed Slice Sets). Let S_C be a slice set of a CFG G, and let S_{C_Δ} be the Δ-augmented slice set of S_C. The slice set S_C is a *weakly* or *strongly control-closed slice set* if S_{C_Δ} is a weakly or strongly control-closed set in G, respectively.

Consider the N'_Δ set in Example 5.3, the slice set S_C in Fig. 4, and the RVs in Fig. 11. S_C is a weakly control-closed slice set since $S_{C_\Delta} = \mathrm{N}'_\Delta$ and S_{C_Δ} is a weakly control-closed set.

6 CORRECTNESS OF DEPENDENCE-BASED SLICING

We shall provide the main theorem in this section stating that if P_2 is the slice of P_1 obtained from dependence based slicing algorithms such as Weiser [51], or Ottenstein and Ottenstein [34] computing the slice set S_C according to definition 2.11, then P_1 is nontermination (in)sensitively semi-equivalent to P_2, i.e., $P_1 \gtrsim_C P_2$ or $P_1 \simeq_C P_2$ holds. But, it is impractical to check the relation $\gtrsim_C$ or $\simeq_C$ between two given programs. We thus adapt some concepts from [3, 39] in order to prove the semi-equivalence relations between P_1 and P_2. In particular, we shall define a relation $\overset{seq}{\sim}$ and show that $\overset{seq}{\sim}$ is either a *weak bisimulation* or a *weak simulation* relation between the original program and its slice, according to Def. 6.2 given in this section, depending on the preservation of nontermination by the slicing algorithm. If $\overset{seq}{\sim}$ is a weak simulation then $P_1 \gtrsim_C P_2$, and if $\overset{seq}{\sim}$ is a weak bisimulation relation then $P_1 \simeq_C P_2$.

We define labeled transitions $i \vdash \Gamma \cdot (n, \sigma) \xrightarrow{l} \Gamma' \cdot (n', \sigma')$, where the label l is either an observable node n representing an observable move, or the symbol τ representing a silent move.

Definition 6.1 (Labeled Transition). For all configurations $\Gamma_1 = \Gamma \cdot (n, \sigma)$ and Γ_2 such that $i \vdash \Gamma_1 \longrightarrow \Gamma_2$ holds, $i \in \{1, 2\}$, where δ is the context of Γ_1, we define

- $i \vdash \Gamma_1 \xrightarrow{n} \Gamma_2$ if $n \in S_C$ and $rv(n, \delta) \neq \emptyset$
- $i \vdash \Gamma_1 \xrightarrow{\tau} \Gamma_2$ otherwise.

We write:

- $i \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_2$ for the reflexive transitive closure of $i \vdash \Gamma_1 \xrightarrow{\tau} \Gamma_2$
- $i \vdash \Gamma_0 \overset{n}{\Rightarrow} \Gamma_2$ if there exists a configuration Γ_1 such that $i \vdash \Gamma_0 \overset{\tau}{\Rightarrow} \Gamma_1$ and $i \vdash \Gamma_1 \xrightarrow{n} \Gamma_2$

The observable transition $\xrightarrow{n}$ requires that $n \in S_C$ and $rv(n, \delta)$ is non-empty in order to ensure that n affects the slicing criterion in δ. In case of a silent transition $\xrightarrow{\tau}$ we require either $n \notin S_C$, or $rv(n, \delta) = \emptyset$ regardless of whether $n \in S_C$ or not, which ensures that n does not affect the slicing criterion in δ, and so it is a silent transition. Thus, for an observable transition, the values of all variables in the RV set at n are observable whereas, for a silent transition, either n is not part of the sliced code, or the RV set at n is empty and no variable values can be observed.

Definition 6.2 (Weak Simulation and Bisimulation). Consider the following properties for relations Φ:

(i) if $\Gamma_1 \Phi \Gamma_2$ and $1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$, then there exists Γ_2' such that $\Gamma_1' \Phi \Gamma_2'$ and $2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$.
(ii) if $\Gamma_1 \Phi \Gamma_2$ and $2 \vdash \Gamma_2 \overset{n}{\Rightarrow} \Gamma_2'$, then there exists Γ_1' such that $\Gamma_1' \Phi \Gamma_2'$ and $1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$.

Φ is a *weak simulation* if (i) holds, and a *weak bisimulation* if both (i) and (ii) holds.

If there is a weak bisimulation between the configurations of the original program and those of its slice, then if the original program can make some *observable action* (i.e., visit a node in S_C), then the sliced program can do it as well and vice versa. On the other hand, if weak simulation between the configurations of the original program and its slice holds and if the original program can perform some observable action, then the sliced program can do so, but not necessarily the other way around as the original code may contain, for instance, infinite loops that have been skipped in the sliced code.

However, the mere existence of a weak (bi)simulation does not imply much: for instance, the empty[4] relation is a weak bisimulation. Our challenge is thus to find a relation that is either a weak bisimulation or simulation, according to Def. 6.2, and *observes the same values of relevant variables at every C-observable node.* In order to define the relation $\overset{seq}{\sim}$ that is a weak (bi)simulation and observes the same values of observable variables at every C-observable node, we need to ensure that a program and its slice observe the same values of observable variables at every S_C-observable node. This requires defining equivalence of all stores between two configurations with respect to RVs in a pair-wise fashion as follows:

Definition 6.3 (Equivalent Stores upto RVs). Let $\Gamma_{(1,k)}$ and $\Gamma_{(2,l)}$ be two configurations such that

$$\Gamma_{(1,k)} = (n_1, \sigma_1) \cdot \ldots \cdot (n_k, \sigma_k) \text{ and } \Gamma_{(2,l)} = (m_1, \sigma^1) \cdot \ldots \cdot (m_l, \sigma^l).$$

The stores of $\Gamma_{(2,l)}$ are equivalent to that of $\Gamma_{(1,k)}$ upto relevant variables (denoted $stores(\Gamma_{(1,k)}) =_{RV} stores(\Gamma_{(2,l)})$) if $l \leq k$ and the following holds:

(1) the configurations $\Gamma_{(1,i)}$ and $\Gamma_{(2,i)}$ are in the same context δ_i such that $\sigma_i =_{rv(m_i,\delta_i)} \sigma^i$ for all $1 \leq i \leq l$, and
(2) $rv(n_i, \delta_i) = \emptyset$ where δ_i is the context of configuration $\Gamma_{(1,i)}$ for all $l + 1 \leq i \leq k$.

Example 6.4. Consider the CFG G in Fig. 4 and the set of RVs listed in Fig. 11. Let us consider the configurations

$$\Gamma_1 = (n_3, \sigma_1) \cdot (n_E, \sigma_2), \quad \Gamma_2 = (n_3, \sigma_1), \text{ and } \Gamma_3 = (n_8, \sigma_3) \cdot (n_E, \sigma_2)$$

such that the stores σ_1, σ_2, and σ_3 are as follows:

$$\begin{array}{rcl} \sigma_1 & = & \{a \mapsto 8, b \mapsto 5, p \mapsto 7, q \mapsto 9\} \\ \sigma_2 & = & \{x \mapsto 7, y \mapsto 9\} \\ \sigma_3 & = & \{a \mapsto 8, b \mapsto 5, p \mapsto 7, q \mapsto 9, r \mapsto 19\}. \end{array}$$

- $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ holds since the RVs a, b at n_3 have same values in the botom states of Γ_1 and Γ_2, and $rv(n_E, L_1) = \emptyset$.

[4]Note that empty relation is trivially a weak (bi)simulation relation since no Γ_1, Γ_2 exists such that $\Gamma_1 \Phi \Gamma_2$ holds.

- $stores(\Gamma_2) =_{RV} stores(\Gamma_1)$ does not hold since Γ_1 has more states than Γ_2.
- $stores(\Gamma_3) =_{RV} stores(\Gamma_2)$ does not hold since $rv(n_E, L_2) \neq \emptyset$.
- $stores(\Gamma_1) =_{RV} stores(\Gamma_3)$ does not hold since the contexts of node n_E in both configurations are different.

We now define the relation $\overset{seq}{\sim}$ between the configurations of an original program and its slice as follows:

Definition 6.5 ($\overset{seq}{\sim}$). Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 at nodes n_1 and n_2 in contexts δ_1 and δ_2 respectively. The relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ holds if

(1) $obs(n_1, \delta_1) = obs(n_2, \delta_2)$,
(2) $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$, and
(3) $m \in obs(n_1, \delta_1) \implies \mathfrak{d}([n_1..m], \delta_1) = \mathfrak{d}([n_2..m], \delta_2)$.

Theorem 6.6 below states that $\overset{seq}{\sim}$ is either a weak simulation or a weak bisimulation, depending on the slice set S_C computed in Def. 2.11 to be well-formed weakly or strongly control-closed set. If S_C is a well-formed strongly control-closed set, then the slice is nontermination sensitive and $\overset{seq}{\sim}$ is a weak bisimulation, otherwise it is a weak simulation.

Theorem 6.6 (Correctness Condition). *Assume that S_C is closed under $\xrightarrow{dd}$ and $\xrightarrow{cd}$. Then $\overset{seq}{\sim}$ is a weak simulation if S_C is a well-formed weakly control-closed set, and a weak bisimulation if S_C is a well-formed strongly control-closed set.*

Theorem 6.7 stated below ensures that when $\overset{seq}{\sim}$ is a weak (bi)simulation, then the slice P_2 is correct with respect to the original program P_1 if it is obtained from some dependence-based slicing method such as the ones defined in Def. 2.11 and Def. 4.10.

Theorem 6.7 (Correctness). *If $\overset{seq}{\sim}$ is a weak simulation or bisimulation, then $P_1 \gtrsim_C P_2$ or $P_1 \simeq_C P_2$, respectively.*

The proofs of Theorems 6.6 and 6.7 are given in Section 7. Thus, P_2 is a correct slice of program P_1 with respect to C (i.e. $P_1 \simeq_C P_2$) if $\overset{seq}{\sim}$ is a weak bisimulation as the semantic conditions (i.e. conditions in Def. 3.6) for this relation are satisfied by the possibly nonterminating programs. However, if P_1 is possibly nonterminating, and P_2 is terminating as the slicing algorithm is nontermination insensitive, then $\overset{seq}{\sim}$ is a weak simulation, and the relation $P_1 \gtrsim_C P_2$ holds which ensures that P_1 is nontermination insensitively semi-equivalent to P_2.

Fig. 12 illustrates the intuitive idea of weak (bi)simulation between the configurations of an original program P_1 and its slice P_2. CFG nodes are labelled by possible program configurations. In the CFG of the original program P_1, the CFG nodes $n_3, \ldots, n_5$ have two program configurations for two call sites. On the other hand, these nodes in the CFG of the slice P_2 have only one program configuration related to the second call site. This is due to the fact that $code_2(n) =$ **skip** since $n_1 \notin S_C$ and we shall have the transition $2 \vdash \Gamma_1' \to \Gamma_7'$ according to Call-skip semantic rule in Table 1. Thus, there is no flow of execution from n_i to n_j for $(i, j) \in \{(1, 2), (2, 3), (5, 6), (6, 7)\}$. Table 2 lists all the weak (bi)simulation relations for the configurations in programs P_1 and P_2.

It is worthwhile to mention that the (bi)simulation relation $\overset{seq}{\sim}$ is a sufficient condition for P_2 to be the correct slice of P_1, but not a necessary condition. Valid slices can be obtained from original programs for which $\overset{seq}{\sim}$ is not a (bi)simulation relation. For example, P_2 may contain a sequence of statements not affecting the slicing criterion for which the execution path of P_2 is different than P_1 so that the (bi)simulation relation does not hold. However, dependence-based slicing such as Def. 2.11 and 4.10 produces slices from original code for which $\overset{seq}{\sim}$ is a weak (bi)simulation according to the *correctness condition* theorem and the slice is correct according to the *correctness* theorem.

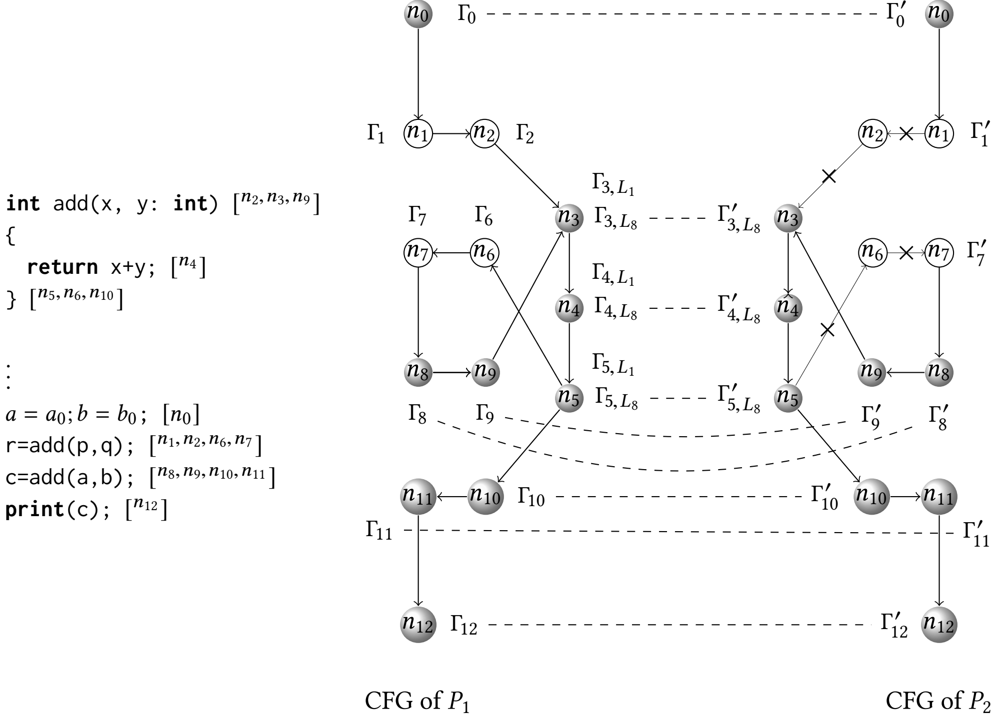

Fig. 12. A simplified version of the code in Fig. 4(a) (left), its CFG (middle), and the CFG of its slice (right) where the slicing criterion is $C = \{(n_{12}, \{c\})\}$. The labels inside $[\cdots]$ beside a program statement denote the CFG nodes representing that statement. The CFGs are labelled by program configurations: Γ_i denotes the configuration at node n_i and $\Gamma_{i,\delta}$ denotes the configuration Γ_i for node n_i in context δ. The contexts are L_1 and L_8 where $L_i = \ell(n_i)$ for $i = 1, 8$. Some (bi)simulation relations are represented by the dashed lines between the configurations of the original program P_1 and the sliced program P_2. Solid circles represent CFG nodes that belong to the slice set.

$\Gamma \overset{seq}{\sim} \Gamma'$	$\Gamma \in \{\Gamma_0, \Gamma_1, \Gamma_2, \Gamma_{3,L_1}, \Gamma_{4,L_1}, \Gamma_{5,L_1}, \Gamma_6, \Gamma_7\}$ $\Gamma' \in \{\Gamma'_0, \Gamma'_1, \Gamma'_7\}$
$\Gamma_i \overset{seq}{\sim} \Gamma'_i$	$i \in \{8, \ldots, 12\}$
$\Gamma_{i,L_8} \overset{seq}{\sim} \Gamma'_{i,L_8}$	$i \in \{3, 4, 5\}$

Table 2. The (bi)simulation relation between the configurations of P_1 and P_2 in Fig. 12

7 PROOFS OF THE CORRECTNESS CONDITION AND THE CORRECTNESS THEOREMS

In what follows, we provide additional lemmas in proving the correctness condition and the correctness theorems (i.e., Theorem 6.6 and 6.7). All lemmas assume that S_C **is a well-formed weakly or strongly control closed slice set**. Some are applicable only when S_C is a well-formed strongly control closed slice set: we will then mention this explicitly. In some lemmas we state that the original program P_1 (with code map $code_1$) has the configuration Γ_1, and the sliced code P_2

(with code map $code_2$) has the configuration Γ_2. The calling contexts δ_i are then derived from the configurations Γ_i for $i = 1, 2$.

7.1 Lemmas on relevant variables

Lemma 7.1. *Let $[\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ be any contextually valid path in the CFG G such that*
(1) $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$, and
(2) for all $1 \leq i \leq k-1$, $n_i \notin S_C$ or $rv(n_i, \delta_i) = \emptyset$.
Then, for all $v \in rv(n_k, \delta_k)$ declared in any valid context δ and for all $1 \leq i \leq k-1$, if (n_i, δ_i) is in the definition scope of (v, δ), then $v \notin def(n_i)$.

Proof. We prove the lemma by induction on i in the sequence $k-1, \dots, 1$ that $v \notin def(n_i)$ if (n_i, δ_i) is in the definition scope of (v, δ). Let $i = k-1$ (base case), and let (n_i, δ_i) is in the definition scope of (v, δ). If $v \in def(n_i)$, we must have $n_i \in S_C$, and $rv(n_i, \delta_i)$ includes the elements of $ref(n_i) \neq \emptyset$ according to Case (3a) in the definition of relevant variables (Def. 4.7). This contradicts the assumption of the lemma, and thus $v \notin def(n_{k-1})$. Assume that $v \notin def(n_i)$ if (n_i, δ_i) is in the definition scope of (v, δ) for all $l \leq i \leq k-1$ such that $l > 1$ (IH). Consider node n_{l-1} such that (n_{l-1}, δ_{l-1}) is in the definition scope of (v, δ). This particular instance of variable v may be defined or referenced at all CFG nodes of a procedure declaring this variable and any CAssign/RAssign node adjacent to the CFG nodes of that procedure. The relative positions of node n_k and n_{l-1} such that v may be defined at n_{l-1}, and both (n_k, δ_k) and (n_l, δ_l) are in the reference scope of (v, δ), are shown in Fig. 13.

In the last two scenarios, n_{k-1} is either an Entry or an Exit node. Since $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$, (n_k, δ_k) is an element of the Δ-augmented slice set S_{C_Δ}. Since S_C is a well-formed set, (n_{k-1}, δ_{k-1}) must be an element of S_{C_Δ} according to the definition of the well-formed sets (see Def. 5.10 and Def. 5.11). This implies that $n_{k-1} \in S_C$ and $rv(n_{k-1}, \delta_{k-1}) \neq \emptyset$, which contradicts the assumption of the lemma. Thus the last two scenarios are not possible.

In the first two scenarios, v is a RV in $rv(n_l, \delta_l)$ due to Case (2) in the definition of relevant variables since this particular v is not defined by any node in the path $[n_l..n_{k-1}]$ according to inductive hypothesis and $\delta_l = \delta_k$. If the particular v is defined at n_{l-1}, we obtain $n_{l-1} \in S_C$ and $rv(n_{l-1}, \delta_{l-1}) \neq \emptyset$ where the RV set $rv(n_{l-1}, \delta_{l-1})$ includes the elements of $ref(n_{l-1})$ according to Case (3a) in the definition of relevant variables. This contradicts the assumption of the lemma, and thus any RV v in $rv(n_k, \delta_k)$ is not defined at n_{l-1}. □

Lemma 7.2. *Let $\hat{\pi}_1 = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ be any contextually valid path such that $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$, and let $\hat{\pi}_2$ be any S_{C_Δ}-path. If the path $\hat{\pi}_2$ meets with the path $\hat{\pi}_1$ at some $(n, \delta) \in \hat{\pi}_i$ for $i = 1, 2$ such that $(n, \delta) \notin S_{C_\Delta}$ and the paths are node-disjoint afterward, then S_C is not weakly or strongly control-closed slice set.*

Proof. Let $\hat{\pi}_2 = [\![(m^1, \delta^1)..(m^l, \delta^l)]\!]$ for any $l > 1$. Since (n, δ) is an element in the path $\hat{\pi}_2$, and $\hat{\pi}_2$ is an S_{C_Δ}-path, $[\![(n, \delta)..(m^l, \delta^l)]\!]$ is also an S_{C_Δ}-path. Since $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$ according to the assumption of the lemma, (n_k, δ_k) is an element in S_{C_Δ}. Let there exists an index $1 \leq j \leq k-1$ such that $(n, \delta) = (n_j, \delta_j)$.

Since the paths $\hat{\pi}_1$ and $\hat{\pi}_2$ are node disjoint after meeting at (n, δ) (see the left graph in Fig. 14), there must exist an S_{C_Δ}-path $[\![(n_j, \delta_j)..(n_t, \delta_t)]\!]$ from (n_j, δ_j) towards (n_k, δ_k) for any $j+1 \leq t \leq k$. This is always possible since either we choose $(n_t, \delta_t) = (n_k, \delta_k)$ if no other element in $[\![(n_j, \delta_j)..(n_k, \delta_k)]\!]$ is in S_{C_Δ} except (n_k, δ_k) or we choose the first reachable element $(n_t, \delta_t) \in S_{C_\Delta}$ from (n_j, δ_j) towards (n_k, δ_k). Then, S_C is not weakly or strongly control-closed slice set since node n in context δ is not weakly committing due to have two S_{C_Δ}-paths. □

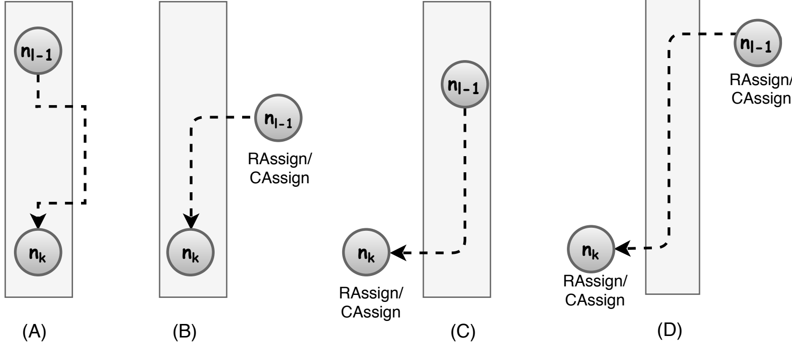

Fig. 13. Relative positions of CFG nodes n_{l-1} and n_k in contexts δ_{l-1} and δ_k when these nodes are in the scope of (v, δ). Nodes inside the solid box belong to the same procedure.

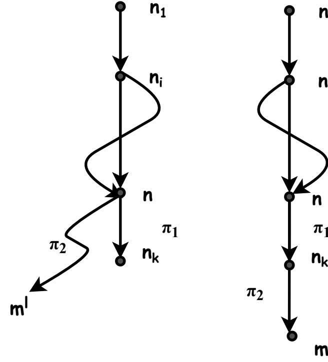

Fig. 14. Paths π_1 and π_2 are node-disjoint (left) and coincident (right) after meeting at n

Lemma 7.3. *Let* $[\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ *be any contextually valid path in the CFG G such that*
(1) $n_k \in S_C$ *and* $rv(n_k, \delta_k) \neq \emptyset$*, and*
(2) $n_i \notin S_C$ *or* $rv(n_i, \delta_i) = \emptyset$ *for all* $1 \leq i \leq k-1$*.*
If $\delta_i = \delta_k$ *for any* $1 \leq i \leq k-1$*, then* $rv(n_k, \delta_k) = rv(n_i, \delta_i)$*.*

Proof. Assume that $\delta_i = \delta_k$ for any $1 \leq i \leq k-1$, and let $\hat{\pi}_1 = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$.

$rv(n_k, \delta_k) \subseteq rv(n_i, \delta_i) \Rightarrow$ Let v be any RV in $rv(n_k, \delta_k)$ declared in any valid context δ. According to Def. 4.5, (n_i, δ_i) is in the definition and reference scope of (v, δ). Moreover, according to Lemma 7.1, this particular v is not defined by any node n_j for all $1 \leq j \leq k-1$ if n_j is in the definition scope of (v, δ). Then, v is a RV in $rv(n_i, \delta_i)$ according to Case (2) in the definition of relevant variables (Def. 4.7). Thus, $rv(n_k, \delta_k) \subseteq rv(n_i, \delta_i)$.

$rv(n_i, \delta_i) \subseteq rv(n_k, \delta_k) \Rightarrow$ Let v be a RV in $rv(n_i, \delta_i)$ declared in any valid context δ. If v is not an element in $rv(n_k, \delta_k)$, then there exists another contextually valid path $\hat{\pi}_2 = [\![(n_i = m^0, \delta_i = \delta^0), (m^1, \delta^1)..(m^l, \delta^l)]\!]$ in the CFG G such that $(m^l, \delta^l) \neq (n_k, \delta_k)$, and according to Case (2) in the definition of relevant variables,

$$m^l \in S_C \text{ and } v \in rv(m^l, \delta^l).$$

Moreover, $v \notin def(m^j)$ for all nodes $m^j \in [n_i..m^{l-1}]$ if (m^j, δ^j) is in the definition scope of (v, δ).

Consider the Δ-augmented slice set S_{C_Δ} (in Def. 5.11). Path $\hat{\pi}_1$ is a S_{C_Δ}-path since either $n_j \notin S_C$ or $rv(n_j, \delta_j) = \emptyset$ and hence $(n_j, \delta_j) \notin S_{C_\Delta}$ for all $1 \leq j \leq k-1$, but $(n_k, \delta_k) \in S_{C_\Delta}$. If the path $\hat{\pi}_2$ meets

with the path $\hat{\pi}_1$ at some $(n, \delta') \in \hat{\pi}_1$ and are node disjoint afterward (see paths in Fig. 14), then S_C is not weakly or strongly control-closed slice set according to Lemma 7.2, and this contradicts with our general assumption that S_C is a weakly or strongly control-closed slice set. If the path $\hat{\pi}_2$ meets with $\hat{\pi}_1$ and they coincide afterward (Fig. 14), then $(n_k, \delta_k) \in \hat{\pi}_2$. Note that path $\hat{\pi}_1$ cannot include (m^l, δ^l) since $\hat{\pi}_1$ does not contain any element from S_{C_Δ} except (n_k, δ_k). However, if v is a RV in $rv(n_i, \delta_i)$, then v is also a RV in $rv(n_k, \delta_k)$ according to Case (2) in the definition of relevant variables since $\delta_i = \delta_k$ and this particular v is not defined by any node m^j in the path $[n_i..m^{l-1}]$ if (m^j, δ^j) is in the definition scope of (v, δ).

Thus, we must have $v \in rv(n_k, \delta_k)$ which proves that $rv(n_i, \delta_i) \subseteq rv(n_k, \delta_k)$, and consequently $rv(n_k, \delta_k) = rv(n_i, \delta_i)$. □

Lemma 7.4. *Let* $[\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ *be any contextually valid path in the CFG G such that*
(1) $n_k \in S_C$ *and* $rv(n_k, \delta_k) \neq \emptyset$*, and*
(2) $n_i \notin S_C$ *or* $rv(n_i, \delta_i) = \emptyset$ *for all* $1 \leq i \leq k-1$.
If $\delta_i \neq \delta_k$ *for any* $1 \leq i \leq k-1$*, then* $rv(n_i, \delta_i) = \emptyset$.

Proof. Let $\hat{\pi}_1 = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$. Consider the Δ-augmented slice set S_{C_Δ} in Def. 5.11. Path $\hat{\pi}_1$ is an S_{C_Δ}-path since either $n_i \notin S_C$ or $rv(n_i\delta_i) = \emptyset$ and hence $(n_i, \delta_i) \notin S_{C_\Delta}$ for all $1 \leq i \leq k-1$. Moreover, S_{C_Δ} includes the element (n_k, δ_k) due to the assumption that $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$. Consider any $i \in \{1, \ldots, k-1\}$ such that $\delta_i \neq \delta_k$.

If $rv(n_i, \delta_i) \neq \emptyset$, we must have $n_i \notin S_C$ due to the second assumption of the lemma. Then, any RV in $rv(n_i, \delta_i)$ is due to Case (2) in the definition of relevant variables. If any RV $v \in rv(n_k, \delta_k)$ declared in any valid context δ is also a RV in $rv(n_i, \delta_i)$ due to this case, and since $\delta_i \neq \delta_k$ and (n_i, δ_i) is in the reference scope of (v, δ), n_k must be a CAssign node according to Case (3) in Def. 4.5. If node $n_k \in S_C$ is a CAssign node, then n_{k-1} is a Call node and (n_{k-1}, δ_{k-1}) is an element in the Δ-augmented slice set S_{C_Δ} according to the definition of the well-formed sets (see Def. 5.10 and Def. 5.11), since S_C is a well-formed set. Thus we obtain $n_{k-1} \in S_C$ and $rv(n_{k-1}, \delta_{k-1}) \neq \emptyset$ which contradicts the assumption of the lemma. Thus, no RV in $rv(n_k, \delta_k)$ is an element in $rv(n_i, \delta_i)$.

If $rv(n_i, \delta_i)$ is a nonempty set, there exists $\hat{\pi}_2 = [\![(n_i = m^0, \delta_i = \delta^0), (m^1, \delta^1)..(m^l, \delta^l)]\!]$ in the CFG G such that $(m^l, \delta^l) \neq (n_k, \delta_k)$, and according to Case (2) in the definition of relevant variables,

$$m^l \in S_C \text{ and } rv(m^l, \delta^l) \neq \emptyset.$$

Moreover, for any RV $v \in rv(m^l, \delta^l)$ declared in any valid context δ, $v \notin def(m^j)$ for all $j \in \{0, \ldots, l\}$ if (m^j, δ^j) is in the definition scope of (v, δ).

If the path $\hat{\pi}_2$ meets with the path $\hat{\pi}_1$ at some $(n, \delta) \in \hat{\pi}_1$ and are node disjoint afterward (Fig. 14), then S_C is not weakly or strongly control-closed slice set according to Lemma 7.2, and this contradicts with our general assumption that S_C is a weakly or strongly control-closed slice set.

If the path $\hat{\pi}_2$ meets with $\hat{\pi}_1$ at (n, δ) and they coincide afterward (Fig. 14), then $(n_k, \delta_k) \in \hat{\pi}_2$. Note that path $\hat{\pi}_1$ cannot include (m^l, δ^l) since $\hat{\pi}_1$ does not contain any element from S_{C_Δ} except (n_k, δ_k). According to Case (2) in the definition of relevant variables, both (m^l, δ^l) and (n_i, δ_i) are in the reference scope of (v, δ). Then, according to Def. 4.5, we obtain either $\delta^l = \delta_i$ or $\delta^l \neq \delta_i$, and m^l is a CAssign node in the latter case such that $\delta^{l-1} = \delta_i$.

So, path $\hat{\pi}_2$ includes an element (n_k, δ_k) such that $\delta_k \neq \delta_i$ but either $\delta_i = \delta^l$ or $\delta_i = \delta^{l-1}$. Due to this change of contexts of nodes in the path $[n_i..m^l]$, there must exist a Call node in $[n_i..n_{k-1}]$ and a Ret node in $[n_k..m^l]$. So, the path $[\![(n_i, \delta_i)..(n_k, \delta_k)]\!]$ must include an element (n_c, δ_c) such that n^c is a Call node and $\delta_k = \delta_c \circ \ell(n_c)$. If there are multiple Call nodes in the path $[n_i..n_k]$, let n^c be the closest Call node to n_k. Then, we must have $(n_c, \delta_c) \in S_{C_\Delta}$ from $(n_k, \delta_k) \in S_{C_\Delta}$ due to well-formedness of S_C according to the second case in the definition of well-formed sets (Def. 5.10), which states that if any (n_k, δ_k) is an element in S_{C_Δ}, then any (n_c, δ_c) such that $\delta_k = \delta_c \circ \ell(n_c)$

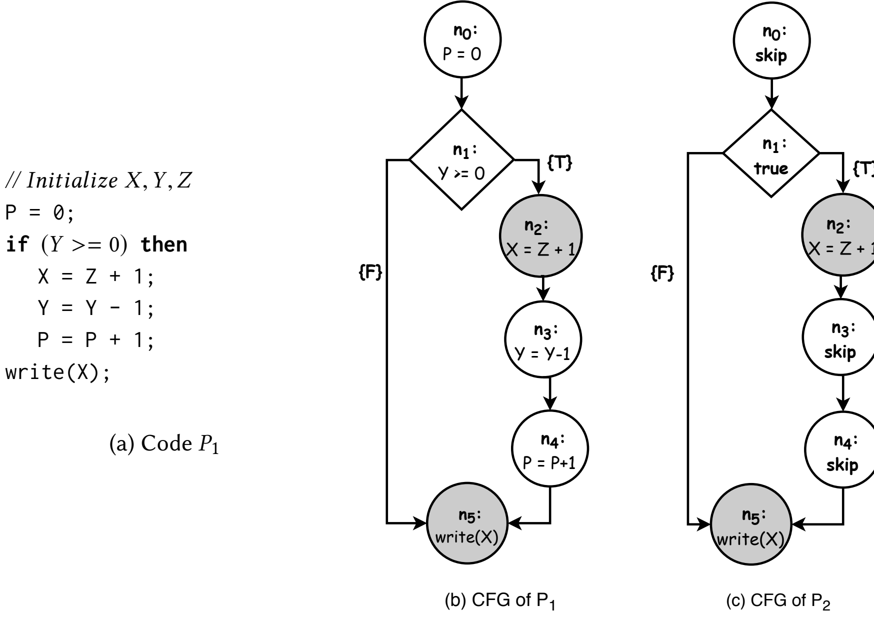

Fig. 15. Slicing criterion $C = \{(n_5, \{X\})\}$ and the slice set $S_C = \{n_5, n_2\}$. P_2 is the slice of P_1 and its CFG is shown in (c).

must be in S_{C_Δ}. Consequently, $n_c \in S_C$ and $rv(n_c, \delta_c) \neq \emptyset$ which contradicts the assumption of the lemma that either $n_c \notin S_C$ or $rv(n_c, \delta_c) = \emptyset$.

Since any possibility of the path $\hat{\pi}_2$ leads to a contradiction, no such path $\hat{\pi}_2$ exists to contribute a RV in $rv(n_i, \delta_i)$, and consequently $rv(n_i, \delta_i) = \emptyset$. □

7.2 Lemma on the singleton property of an observable set

We prove the singleton property of the set of observable nodes $obs(n, \delta)$: $|obs(n, \delta)| \leq 1$ for any CFG node n and context δ. The singleton property of any observable set is a necessary condition for the correctness of any dependence-based slicing method.

Consider the code P_1 and its CFG in Fig. 15 where the slicing criterion contains the variable X at node n_5. Let the slice be computed according to an incorrect control dependency relation that does not capture the control dependency $n_1 \xrightarrow{cd} n_2$. Then the slice set will contain the nodes n_2 and n_5. Fig. 15(c) contains the CFG of the sliced code P_2 according to Def. 4.10. We have $obs(n_0, \delta) = \{n_2, n_5\}$ for any context δ due to the existence of two separate paths $[\![(n_0, \delta)..(n_2, \delta)]\!]$ and $[\![(n_0, \delta)..(n_5, \delta)]\!]$. But, the CFG of the sliced code P_2 is not a valid slice of the CFG of P_1 as X can have values x_0 or $z_0 + 1$ at n_5 in the original code, but it can only have the value $z_0 + 1$ at n_5 in P_2 for any initial values x_0 of X and z_0 of Z. Here, the problem with the slice P_2 is that even though the statement "`X=Z+1`" is sliced, the condition `Y >= 0` on which the statement should be control dependent is not sliced. The problem becomes even more serious if we replace the **if** statement by a **while** statement in P_1. P_1 is then always terminating, but P_2 will be nonterminating (if $code_2(n_1) =$ **true**) or the loop will never be entered (if $code_2(n_1) =$ **false**) and the statement "`X = Z + 1`" will never be executed. Thus, if the singleton property is not satisfied by the observable set, the semi-equivalence relation between P_1 and P_2 cannot be proved. In the following, we prove that the observable set is always at most a singleton if the slice set S_C is a well-formed weakly or strongly control-closed slice set.

Lemma 7.5 (Singleton). *If S_C is a well-formed weakly or strongly control-closed slice set in the CFG G, then $obs(n, \delta)$ is at most a singleton for any CFG node n and its valid context δ.*

Proof. Let S_{C_Δ} be the Δ-augmented slice set according to Def. 5.11. If $(n, \delta) \in S_{C_\Delta}$, then $obs(n, \delta) = \{n\}$ trivially holds. So, assume that $(n, \delta) \notin S_{C_\Delta}$. If S_C is a well-formed weakly or strongly control-closed slice set, then S_{C_Δ} is a weakly or strongly control closed set. If S_{C_Δ} is a weakly control-closed set, node n in δ is S_{C_Δ}-weakly committing. If S_{C_Δ} is a strongly control-closed set, then node n in δ is S_{C_Δ}-weakly committing since a S_{C_Δ}-strongly committing vertex is also S_{C_Δ}-weakly committing and a S_{C_Δ}-avoiding vertex is vacuously S_{C_Δ}-weakly committing (see Def. 5.4 and Def. 5.8). So, if there exist S_{C_Δ}-paths

$$\hat{\pi}_i = [\![(n, \delta)..(n_k, \delta_k)]\!] \text{ for all } i \geq 1,$$

all paths meet at n_k in δ_k such that $(n_k, \delta_k) \in S_{C_\Delta}$, which implies $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$ (Def. 5.11). Moreover, for any $(n', \delta') \in \hat{\pi}_i$ such that $(n', \delta') \neq (n_k, \delta_k)$, we have $(n', \delta') \notin S_{C_\Delta}$ which implies either $rv(n', \delta') = \emptyset$ or $n' \notin S_C$ (Def. 5.11). Furthermore, we obtain $rv(n', \delta') = rv(n_k, \delta_k)$ if $\delta' = \delta_k$ according to Lemma 7.3 and $rv(n', \delta') = \emptyset$ if $\delta' \neq \delta_k$ according to Lemma 7.4. So, conditions (1) and (2) in Def. 4.9 are satisfied for all nodes in the path $[n..n_{k-1}]$ proving that $obs(n, \delta) = \{n_k\}$. If there exists no S_{C_Δ}-path from n in δ, then $obs(n, \delta) = \emptyset$ since no $(n_k, \delta_k) \in S_{C_\Delta}$ exists such that $rv(n_k, \delta_k) \neq \emptyset$. □

7.3 Lemmas related to observable relations

Lemma 7.6. *Let Γ and Γ' be configurations of CFG nodes n and n' in contexts δ and δ', and let $obs(n', \delta') = \{m\}$ for any $m \in S_C$. If there exists a silent transition $i \vdash \Gamma \xrightarrow{\tau} \Gamma'$ for any $i \in \{1, 2\}$, then $obs(n, \delta) = \{m\}$.*

Proof. Since $obs(n', \delta') = \{m\}$, there exists a contextually valid path $[\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ such that $(n_1, \delta_1) = (n', \delta')$, $n_k = m \in S_C$, and $rv(n_k, \delta_k) \neq \emptyset$ according to the definition of next observable behavior (Def. 4.9). Moreover, conditions (1) and (2) in Def. 4.9 are satisfied for CFG node n_i for all $1 \leq i \leq k - 1$:

(1) if $\delta_i = \delta_k$, then $n_i \notin S_C$ and $rv(n_i, \delta_i) = rv(n_k, \delta_k)$, and
(2) if $\delta_i \neq \delta_k$, then $rv(n_i, \delta_i) = \emptyset$.

We also have $n' \in succ(n)$ due to the silent transition $\xrightarrow{\tau}$ in the assumption of the lemma. So, $[\![(n, \delta)..(n_k, \delta_k)]\!]$ is a contextually valid path. In proving $obs(n, \delta) = \{m\}$, it is thus enough to show that conditions (1) and (2) in Def. 4.9 for node n are satisfied.

According to the definition of labeled transition (Def. 6.1), the silent transition $i \vdash \Gamma \xrightarrow{\tau} \Gamma'$ in the assumption of the lemma yields

$$n \notin S_C \vee rv(n, \delta) = \emptyset. \tag{1}$$

If $\delta = \delta_k$, then we obtain $rv(n, \delta) = rv(n_k, \delta_k)$ according to Lemma 7.3. Since $rv(n_k, \delta_k)$ is nonempty, $rv(n, \delta) \neq \emptyset$, and we obtain $n \notin S_C$ due to condition (1) above. If $\delta \neq \delta_k$, then we obtain $rv(n, \delta) = \emptyset$ according to Lemma 7.4. Thus, conditions (1) and (2) in Def. 4.9 are satisfied, and consequently $obs(n, \delta) = \{m\}$. □

Lemma 7.7. *Let n_1 and n_2 be CFG nodes, and let δ be a valid context of n_1 and n_2. If $obs(n_1, \delta) = obs(n_2, \delta)$, and $obs(n_i, \delta) \neq \emptyset$ for $i = 1, 2$, then $rv(n_1, \delta) = rv(n_2, \delta)$.*

Proof. According to Lemma 7.5, $obs(n_i, \delta)$ is at most a singleton for $i = 1, 2$. Moreover, $obs(n_1, \delta) = obs(n_2, \delta) \neq \emptyset$ according to the assumption of the lemma. Thus, there exists a CFG node $m \in S_C$ such that $obs(n_i, \delta) = \{m\}$ for $i = 1, 2$. Let $\delta_m = \mathfrak{d}([n_i..m], \delta)$ be the context of node m for any $i \in \{1, 2\}$. According to the definition of next observable behavior (Def 4.9), the observable relation

$obs(n_i, \delta) = \{m\}$ for $i = 1, 2$ yields

$$rv(n_i, \delta) = rv(m, \delta_m) \text{ if } \delta = \delta_m \text{ and } rv(n_i, \delta) = \emptyset \text{ if } \delta \neq \delta_m$$

Thus, $rv(n_1, \delta) = rv(n_2, \delta)$ always holds. □

Lemma 7.8. *Let G be the CFG of the slice P_2 with the mapping function $code_2$, and let n_1 be a CFG node in a valid context δ_1. If $obs(n_1, \delta_1) = \{n_k\}$ for any $n_k \in S_C$ and $\delta_k = \mathfrak{d}([n_1..n_k], \delta_1)$, then $\delta_1 = \delta_k$.*

Proof. Since $obs(n_1, \delta_1) = \{n_k\}$, there exists a contextually valid path $\hat{\pi} = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$ according to the definition of next observable behavior, where $\delta_k = \mathfrak{d}([n_1..n_k], \delta_1)$. If $k = 1$, then the lemma trivially holds. Thus, assume that $k > 1$. If $\delta_1 \neq \delta_k$, there must be a Call or Ret node in the path $[n_1..n_k]$ to change the contexts. Assume that there exists an element (n_i, δ_i) in $\hat{\pi}$ such that n_i is a Call or Ret node. If there are multiple elements containing Call/Ret nodes, we assume that (n_i, δ_i) is the closest element to (n_k, δ_k). The change of context occurs in one of the following ways:

(1) Node n_i is a Call node for any $1 \leq i < k$. Then, node n_{i+1} is a CAssign node in context $\delta_{i+1} = \delta_i \circ \ell(n_i) = \delta_k$.
(2) Node n_i is a Ret node for any $1 < i \leq k$. Then, node n_{i-1} is a RAssign node in context $\delta_{i-1} = \delta_i \circ \ell(n_i)$, and $\delta_i = \delta_k$.

In the first case, (n_i, δ_i) must be an element in the Δ-augmented slice set S_{C_Δ} according to the definition of the well-formed sets (see Def. 5.10) since S_C is a well-formed set. Thus we obtain $rv(n_i, \delta_i) \neq \emptyset$ which contradicts condition (2) in the definition of next observable behavior. In the second case, $\delta_i = \delta_k$ implies that $n_i \notin S_C$ according to the first condition of the definition of next observable behavior. Then, we obtain $code_2(n_i)$ =**skip** in P_2 by Case 3 in Def. 4.10, and the Ret node is semantically transformed to a Skip node in the CFG of P_2. This gives us the contradiction that the path $\hat{\pi}$ is not a contextually valid path due to node n_i. So, we conclude that no Call/Ret node n_i exists in $\hat{\pi}$ when $k > 1$ (i.e. $n_1 \neq n_k$), and consequently $\delta_1 = \delta_k$. □

7.4 Proving the weak simulation and bisimulation relations

Lemma 7.9 and 7.10 below state that if the relation $\overset{seq}{\sim}$ holds between two configurations Γ_1 and Γ_2 of P_1 and P_2 (i.e. $\Gamma_1 \overset{seq}{\sim} \Gamma_2$), and there is a silent transition from Γ_i to Γ of program P_i for $i = 1, 2$, then the relation $\Gamma_1 \overset{seq}{\sim} \Gamma$ or $\Gamma \overset{seq}{\sim} \Gamma_2$ hold. In other words, since the observable behavior does not change in silent transitions, the relation $\overset{seq}{\sim}$ also holds between the relevant configurations with the same observable behavior.

Lemma 7.9. *Let Γ_1 and Γ_2 be valid configurations of P_1 and P_2 such that $\Gamma_1 \overset{seq}{\sim} \Gamma_2$. If there exists a transition $1 \vdash \Gamma_1 \xrightarrow{\tau} \Gamma$ where $\Gamma = \Gamma \cdot (n, \sigma)$ is any configuration of P_1 in context δ, and $obs(n, \delta) \neq \emptyset$, then $\Gamma \overset{seq}{\sim} \Gamma_2$ holds.*

Proof. Let $\Gamma_1 = \Gamma_1 \cdot (n_1, \sigma_1)$, let $\Gamma_2 = \Gamma_2 \cdot (n_2, \sigma_2)$, and let δ_1 and δ_2 be the contexts of configurations Γ_1 and Γ_2, respectively. In the following, we prove conditions (1), (2), and (3) in the definition of $\overset{seq}{\sim}$ (Def. 6.5) for the relation $\Gamma \overset{seq}{\sim} \Gamma_2$.

(1) $obs(n, \delta) = obs(n_2, \delta_2)$. There exists a CFG node $m \in S_C$ such that $obs(n, \delta) = \{m\}$ since the nonempty observable set $obs(n, \delta)$ is at most a singleton according to the singleton lemma (Lemma 7.5). Then, we obtain $obs(n_1, \delta_1) = \{m\}$ according to Lemma 7.6 due to the silent transition $\xrightarrow{\tau}$ in the assumption of the lemma. The relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields $obs(n_1, \delta_1) = obs(n_2, \delta_2)$ according to the definition of $\overset{seq}{\sim}$. This concludes that $obs(n, \delta) = obs(n_2, \delta_2)$.

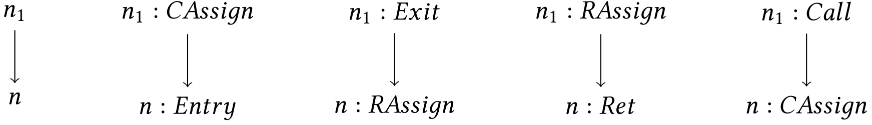

Fig. 16. All possible scenarios where node n is a successor of node n_1. Both n_1 and n belong to the same procedure in the first scenario

(2) $stores(\Gamma) =_{RV} stores(\Gamma_2)$. According to condition (2) in the definition of $\overset{seq}{\sim}$, the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$. The silent transition $1 \vdash \Gamma_1 \overset{\tau}{\longrightarrow} \Gamma$ yields

$$n_1 \notin S_C \vee rv(n_1, \delta_1) = \emptyset \tag{2}$$

according to the definition of labeled transition (Def. 6.1). This silent transition only affects the top store σ of the configuration Γ. As the relation $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ holds, the stores that are beneath the top state of Γ are equivalent to the stores of Γ_2 according to the definition of equivalent stores upto RVs (Def. 6.3). Thus, in order to prove $stores(\Gamma) =_{RV} stores(\Gamma_2)$, it is enough to prove the following conditions:

(C1) if Γ and Γ_2 are in different contexts (i.e. $\delta \neq \delta_2$), then $rv(n, \delta) = \emptyset$, and
(C2) if Γ and Γ_2 are in the same context (i.e. $\delta = \delta_2$), then $\sigma =_{rv(n_2,\delta_2)} \sigma_2$.

We already have proved that $obs(n, \delta) = obs(n_2, \delta_2)$. Let $obs(n, \delta) = obs(n_2, \delta_2) = \{m\}$ for any $m \in S_C$ since $obs(n, \delta) \neq \emptyset$ (assumption of the lemma), and let δ_m be the context of node m. According to Lemma 7.8, $\delta_2 = \delta_m$.

If $\delta \neq \delta_2$ and since $\delta_2 = \delta_m$, we obtain $rv(n, \delta) = \emptyset$ from the relation $obs(n, \delta) = \{m\}$ according to condition (2) in the definition of next observable behavior (Def. 4.9). This proves condition (C1).

Now, in order to prove condition (C2), assume that $\delta = \delta_2$. The relation $obs(n, \delta) = obs(n_2, \delta_2)$ yields

$$rv(n, \delta) = rv(n_2, \delta_2) \text{ (Lemma 7.7).}$$

Since $\delta_2 = \delta_m$ and $obs(n_2, \delta_2) = \{m\}$, we obtain

$$rv(n_2, \delta_2) = rv(m, \delta_m)$$

from the first condition in the definition of next observable behavior. The observable relation $obs(n_2, \delta_2) = \{m\}$ requires that $rv(m, \delta_m) \neq \emptyset$, and thus $rv(n_2, \delta_2)$ is nonempty. Node n is a successor of n_1 due to the silent transition $\overset{\tau}{\longrightarrow}$ in the assumption of the lemma that requires the transition $\longrightarrow$. Since $rv(n, \delta) = rv(n_2, \delta_2)$, we conclude that *any RV* x *in* $rv(n_2, \delta_2)$ *must not be defined at* n_1 as otherwise we obtain $n_1 \in S_C$ and $rv(n_1, \delta_1) \neq \emptyset$ according to Case (3a) in the definition of relevant variables (Def. 4.7) which contradicts condition (2) above. We must have one of the scenarios in Fig. 16 for the relative positions of n_1 and n.

In the first three scenarios, the contexts of nodes n_1 and n are same (i.e. $\delta = \delta_1$). We obtain $\sigma_1 =_{rv(n_2,\delta_2)} \sigma_2$ from the relation $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ according to the definition of equivalent stores upto RV (Def. 6.3). Thus, we infer $\sigma =_{rv(n_2,\delta_2)} \sigma_2$ from the relation $\sigma_1 =_{rv(n_2,\delta_2)} \sigma_2$ and the fact that any RV x in $rv(n_2, \delta_2)$ is not defined at n_1.

In the fourth scenario, configuration $\Gamma_1 = \mathsf{\Gamma}_1 \cdot (n_1, \sigma_1)$ of node n_1 implies that $\mathsf{\Gamma}_1$ is the configuration of a Call node (see the Paramout and the Call rule in Table 1), and n is the matching Ret node of $node(\mathsf{\Gamma}_1)$. Thus, $\mathsf{\Gamma}_1$ and Γ must be in the same context $\delta = \delta_2$, and we obtain $store(\mathsf{\Gamma}_1) =_{rv(n_2,\delta_2)} \sigma_2$ from the relation $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$. The store $store(\mathsf{\Gamma}_1)$ may differ only from the store σ by the values of actual output parameters x that may be assigned at n_1. However, variable x is not a RV in $rv(n_2, \delta_2)$ as any RV in this set is not defined at n_1. Thus, we infer $\sigma =_{rv(n_2,\delta_2)} \sigma_2$ from $store(\mathsf{\Gamma}_1) =_{rv(n_2,\delta_2)} \sigma_2$.

In the fifth scenario, n is the first node in context $\delta = \delta_2$ where $\delta_2 = \delta_1 \circ \ell(n_1)$. Thus, no configuration that is either Γ_1 or a prefix of Γ_1 can be in context δ_2. However, the equality relation $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ requires that Γ_1 or a prefix of Γ_1 must be in the same context as Γ_2 (i.e. δ_2). So, this scenario is impossible.

Thus, if $\delta = \delta_2$, then $\sigma =_{rv(n_2,\delta_2)} \sigma_2$ in all possible scenarios. This proves condition (C2), and thus we conclude that $stores(\Gamma) =_{RV} stores(\Gamma_2)$.

(3) $m \in obs(n,\delta) \implies \mathfrak{d}([n..m],\delta) = \mathfrak{d}([n_2..m],\delta_2)$. Since $obs(n,\delta) \neq \emptyset$, there exists a node $m \in S_C$ such that $m \in obs(n,\delta)$. According to condition (3) in the definition of $\overset{seq}{\sim}$, the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$\mathfrak{d}([n_1..m],\delta_1) = \mathfrak{d}([n_2..m],\delta_2).$$

Since $n \in succ(n_1)$, we have $\delta = \mathfrak{d}([n_1..n],\delta_1)$. Then, we derive the following equalities:

$$\begin{aligned} \mathfrak{d}([n_2..m],\delta_2) &= \mathfrak{d}([n_1..m],\delta_1) \\ &= \mathfrak{d}([n..m],\mathfrak{d}([n_1..n],\delta_1)) \\ &= \mathfrak{d}([n..m],\delta). \end{aligned}$$

□

Lemma 7.10. *Let Γ_1 and Γ_2 be valid configurations of P_1 and P_2 such that $\Gamma_1 \overset{seq}{\sim} \Gamma_2$. If there exists a transition $2 \vdash \Gamma_2 \xrightarrow{\tau} \Gamma$ where $\Gamma = \Gamma \cdot (n,\sigma)$ is any configuration of P_2 in context δ, and $obs(n,\delta) \neq \emptyset$, then $\Gamma_1 \overset{seq}{\sim} \Gamma$ holds.*

Proof. This proof is dual to that of Lemma 7.9. □

The following lemma says that if n_1, in context δ_1, has an observable node $n_k \neq n_1$, then all transitions of the sliced code from n_1 to n_k are silent transitions.

Lemma 7.11. *Let Γ_1 be a valid configuration at CFG node n_1 of program P_2 with the mapping function $code_2$, and let δ_1 be the context of Γ_1. If $obs(n_1,\delta_1) = \{n_k\}$ for any $n_k \in S_C$ and $n_1 \neq n_k$, then there exists a configuration Γ_k at n_k such that $2 \vdash \Gamma_1 \overset{\tau}{\Longrightarrow} \Gamma_k$.*

Proof. There exists a contextually valid path $[\![(n_1,\delta_1)..(n_k,\delta_k)]\!]$ due to the relation $obs(n_1,\delta_1) = \{n_k\}$ according to the definition of next observable behavior (Def. 4.9). Let Γ_i be the configuration of node n_i in context δ_i for all $1 \leq i \leq k$. According to Lemma 7.8, $\delta_1 = \delta_k$. Then, the relation $obs(n_1,\delta_1) = \{n_k\}$ yields $n_1 \notin S_C$ (first condition in Def. 4.9). In the following, we prove for any $i < k$ and $\delta_i = \delta_k$ that there exists an index $i < j \leq k$ such that $2 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_j$ and $\delta_j = \delta_k$. Since $obs(n_1,\delta_1) = \{n_k\}$ and $\delta_i = \delta_k$, we obtain $n_i \notin S_C$ (first condition in Def. 4.9). One of the following holds for node n_i.

- Node n_i is a Cond node: If the minimum distance $dist(tsucc(n_i),n_k)$ is smaller than the minimum distance $dist(n_i,n_k)$, then we get $code_2(n_i) = true$ according to the definition of the mapping function $code_2$ (Def. 4.10) and obtain $n_j = tsucc(n_i)$. We obtain $code_2(n) = false$ otherwise and set $n_j = fsucc(n_i)$. Node n_j is in the path $[n_1..n_k]$, $\delta_j = \delta_k$ in both cases, and we obtain the transition $2 \vdash \Gamma_i \longrightarrow \Gamma_j$ according to the semantic rules in Table 1. Since $n_i \notin S_C$, we obtain the silent transition $2 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_j$ (Def. 6.1).
- Node n_i is a Call node: Since $n_i \notin S_C$, we obtain $code_2(n_i) = skip$ according to the definition of the mapping function $code_2$. Let n' be the matching Ret node in context δ_i such that $\ell(n_i) = \ell(n')$. We obtain $n_j = n'$ and $\delta_j = \delta_i$, and there exists a transition $2 \vdash \Gamma_i \longrightarrow \Gamma_j$ according to the Call-Skip rule in Table 1. Since $n_i \notin S_C$, we obtain $2 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_j$ (Def. 6.1). In the following, we show that $n_j \in [n_1..n_{k-1}]$.

Since $obs(n_1, \delta_1) = \{n_k\}$, either $n_l \notin S_C$ or $rv(n_l, \delta_l) = \emptyset$ for all $1 \le l \le k-1$ according to conditions (1) and (2) in the definition of next observable behavior, and thus (n_l, δ_l) is not an element in the Δ-augmented slice set S_{C_Δ} according to Def. 5.11. However, the relation $obs(n_1, \delta_1) = \{n_k\}$ yields $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$, and thus $(n_k, \delta_k) \in S_{C_\Delta}$. $(n_i, \delta_i) \notin S_{C_\Delta}$ implies $(n_j, \delta_j) \notin S_{C_\Delta}$ due to well-formedness (Defs. 5.10). Since $(n_k, \delta_k) \in S_{C_\Delta}$, but $(n_j, \delta_j) \notin S_{C_\Delta}$ where nodes n_i, n_j, and n_k are in the same contexts and n_j is the immediate subsequent node of n_i in the CFG of the same procedure, n_j must be in the path $[n_1..n_{k-1}]$.

- Node n_i is not a Call or Cond node: Node n_i cannot be a RAssign node as otherwise we will not have any $n_k \in S_C$ such that $\delta_i = \delta_k$. For any other kinds of nodes $n_i \notin S_C$, $code_2(n_i)$ is a *skip* instruction according to the definition of the mapping function $code_2$. Then, we get $n_j \in succ(n_i)$, $\delta_j = \delta_i$, and the transition $2 \vdash \Gamma_i \longrightarrow \Gamma_j$ according to the SKIP rule in Table 1. This implies $2 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_j$ according to the definition of the labeled transition (Def. 6.1).

So, we obtain a subsequence $t_1, \ldots, t_l$ of the sequence of indices $1, \ldots, k$ where $t_1 = 1$, $t_l = k$, and $t_1 \le \ldots \le t_l$ such that $2 \vdash \Gamma_{t_j} \xrightarrow{\tau} \Gamma_{t_{j+1}}$ for all $1 \le j < l$, and consequently, we conclude that $2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$. □

Example 7.12. Let us consider the CFG G and the slice set S_C in Fig. 4, the set of observable nodes $obs(n, \delta)$ in Fig. 11, and the mapping function $code_2$ such that $code_2(n) =$**skip** for any $n \in \{n_3, n_6, n_7\}$. Let $\Gamma_1 = (n_3, \sigma_1)$ be any valid configuration at node n_3. We have $obs(n_3, \epsilon) = \{n_8\}$ where ϵ is the context of configuration Γ_1 and $n_8 \neq n_3$. According to Lemma 7.11, there exists a configuration Γ_k at node n_8 in context ϵ such that $2 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$.

Program P_2 (i.e. $code_2(G)$) has the following sequences of transitions: (i) $2 \vdash \Gamma_1 \longrightarrow (n_6, \sigma_1)$ according to the CALL-SKIP rule, and (ii) $2 \vdash (n_6, \sigma_1) \longrightarrow (n_7, \sigma_1)$ and $2 \vdash (n_7, \sigma_1) \longrightarrow (n_8, \sigma_1)$ according to the SKIP rule in Table 1. Since $n_3, n_6, n_7 \notin S_C$, all these transitions are silent transitions, and thus $2 \vdash (n_3, \sigma_1) \overset{\tau}{\Rightarrow} (n_8, \sigma_1)$.

```
i=1; z=0;
while(z ≥ 0)
   z=z;
y=i+1;
```

```
i=1; skip;
while(false)
  skip;
y=i+1;
```

Fig. 17. Program P_1 (left) and its slice P_2 (right). P_1 is not terminating, but P_2 always terminates

Note that we cannot ensure $1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$ in Lemma 7.11. In order to illustrate this fact, consider the program P_1 and its nontermination insensitive slice P_2 in Fig. 17. The slicing criterion consists of the variable i at the last instruction. If node n represents the condition of the while loop, and node m is the last statement, then there is no transition from n to m in P_1 as the loop condition will be always evaluated to *true*. However, there is a transition from n to m in the sliced code as the condition there is *false*. If S_C is a well-formed strongly control closed set, then we can prove the following lemma since the slice will be non-termination sensitive.

LEMMA 7.13. *Let S_C be a well-formed strongly control-closed slice set, let Γ_1 be a valid configuration at node n_1 of program P_1 with the mapping function $code_1$, and let δ_1 be the context of Γ_1. If $obs(n_1, \delta_1) = \{n_k\}$ for any $n_k \in S_C$ such that $n_1 \neq n_k$, then there exists a configuration Γ_k such that $1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma_k$.*

PROOF. Since $obs(n_1, \delta_1) = \{n_k\}$, there exists a contextually valid path $\hat{\pi} = [\![(n_1, \delta_1)..(n_k, \delta_k)]\!]$. For all $1 \le i \le k-1$, we obtain either $n_i \notin S_C$ or $rv(n_i, \delta_i) = \emptyset$ according to the first and second

conditions in the definition of the next observable behavior (Def. 4.9), and thus, (n_i, δ_i) is not an element in the Δ-augmented slice set S_{C_Δ} according to Def. 5.11. However, the observable relation $obs(n_1, \delta_1) = \{n_k\}$ yields $n_k \in S_C$ and $rv(n_k, \delta_k) \neq \emptyset$, and thus $(n_k, \delta_k) \in S_{C_\Delta}$.

Let Γ_i be the configuration of node n_i in context δ_i for all $1 \leq i \leq k$. In the following, we prove that $1 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_{i+1}$ for any $i < k$ and that any infinite execution from n_1 must go through n_k in δ_k. Consider node n_i for any $1 \leq i \leq k-1$.

If node n_i is a Cond node, then we have either $n_{i+1} = tsucc(n_i)$ or $n_{i+1} = fsucc(n_i)$. If node n_i is not a Cond node, then $n_{i+1} \in succ(n_i)$. In any case, we have a transition $1 \vdash \Gamma_i \longrightarrow \Gamma_{i+1}$ according to the semantic rules in Table 1. Since either $n_i \notin S_C$ or $rv(n_i, \delta_i) = \emptyset$, it is the silent transition $1 \vdash \Gamma_i \xrightarrow{\tau} \Gamma_{i+1}$ (Def. 6.1).

The element (n_i, δ_i) is not S_{C_Δ}-avoiding because there exists a contextually valid path from n_i in context δ_i to n_k in δ_k such that $(n_k, \delta_k) \in S_{C_\Delta}$ (Def. 5.7). Since S_C is a strongly control-closed slice set, all contextually valid maximal paths from (n_i, δ_i) contain an element from S_{C_Δ}. Moreover, the element (n_i, δ_i) is also S_{C_Δ}-weakly committing since S_C is a strongly control-closed slice set implying that all S_{C_Δ}-paths meet at (n_k, δ_k) (see Defs. 5.6, 5.4, 5.8). So, no infinite execution is possible without going through n_k in δ_k.

Thus, we conclude that $1 \vdash \Gamma_1 \overset{\tau}{\Longrightarrow} \Gamma_k$. □

Lemma 7.14 below states that if P_1 and P_2 visit a node n in Γ_1 and Γ_2 configurations in context δ such that the relation $\overset{seq}{\sim}$ holds between these configurations (i.e. $\Gamma_1 \overset{seq}{\sim} \Gamma_2$), and both P_1 and P_2 make an observable move in one step to visit a node in Γ_3 and Γ_4 configurations, then the stores of Γ_3 and Γ_4 are equivalent upto RVs according to Def. 6.3.

Lemma 7.14. *Let Γ_1 and Γ_2 be valid configurations at node n in context δ of programs P_1 and P_2, respectively, such that $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ holds. If there exist transitions $1 \vdash \Gamma_1 \xrightarrow{n} \Gamma_3$ and $2 \vdash \Gamma_2 \xrightarrow{n} \Gamma_4$, then $stores(\Gamma_3) =_{RV} stores(\Gamma_4)$.*

Proof. Let $\Gamma_i = \Gamma_i \cdot (n, \sigma_i)$ for $i = 1, 2$. The relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$stores(\Gamma_1) =_{RV} stores(\Gamma_2) \text{ (condition (2) in Def. 6.5).}$$

Since Γ_1 and Γ_2 are in the same context, the relation $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ implies that the stores underneath the top states of Γ_1 and Γ_2 are equivalent upto RVs according to Def. 6.3. The transitions $1 \vdash \Gamma_1 \xrightarrow{n} \Gamma_3$ and $2 \vdash \Gamma_2 \xrightarrow{n} \Gamma_4$ produce configurations Γ_3 and Γ_4 such that the stores underneath the top states of Γ_3 and Γ_4 are from Γ_1 and Γ_2, and hence they are equivalent upto RVs. Thus, in order to conclude $stores(\Gamma_3) =_{RV} stores(\Gamma_4)$, it is sufficient to prove that $store(\Gamma_3)$ and $store(\Gamma_4)$ are equivalent with respect to the set of RVs at node $node(\Gamma_4)$ in the context of Γ_4.

The transition $1 \vdash \Gamma_1 \xrightarrow{n} \Gamma_3$ or $2 \vdash \Gamma_2 \xrightarrow{n} \Gamma_4$ implies that

$$n \in S_C \text{ and } rv(n, \delta) \neq \emptyset \text{ (Def. 6.1).}$$

Moreover, $ref(n) \subseteq rv(n, \delta)$ according to Case (1) or Case (3) in the definition of relevant variables since $n \in S_C$. So, $\sigma_1(x) = \sigma_2(x)$ for all $x \in ref(n)$. Thus, P_1 and P_2 take the same path from n even if n is a Cond node as both evaluate their conditional expressions to the same value. Thus, both Γ_3 and Γ_4 are the configurations for the same node and context in programs P_1 and P_2. Let Γ_i be the configuration at node m in context δ_m for $i = 3, 4$. In the following, we prove that

$$store(\Gamma_3) =_{rv(m,\delta_m)} store(\Gamma_4).$$

If $\delta = \delta_m$, then $store(\Gamma_3) =_{rv(m,\delta_m)} store(\Gamma_4)$ due to one of the following cases:

- Node n contains an assignment $x = e$ and $x \in rv(m, \delta_m)$: Then, we have $[\![e]\!]\sigma_1 = [\![e]\!]\sigma_2$, and hence $store(\Gamma_3)(x) = store(\Gamma_4)(x)$.

- For any $x \in rv(m, \delta_m)$ and $x \notin def(n)$: We must have $x \in rv(n, \delta)$ according to Case (2) in the definition of relevant variables, and then we obtain $store(\Gamma_3)(x) = store(\Gamma_4)(x)$ from $\sigma_1(x) = \sigma_2(x)$.

If $\delta \neq \delta_m$, then n is either a Call or a RAssign node. Moreover, $m \in succ(n)$. Then, we have one of the following possibilities.

- Node n is a Call node and node m is a CAssign node: According to the CALL rule in Table 1,
$$store(\Gamma_3) = \sigma_1 \text{ and } store(\Gamma_4) = \sigma_2.$$
For any RV x in $rv(m, \delta_m)$ such that $x \in ref(n)$, x is also a RV in $rv(n, \delta)$ according to Case (2) in the definition of relevant variable. Then, we obtain
$$store(\Gamma_3)(x) = store(\Gamma_4)(x) \text{ from } \sigma_1(x) = \sigma_2(x).$$
For any other RV x in $rv(m, \delta_m)$ such that $x \notin ref(m)$ (e.g. x can be a dummy variable), either $store(\Gamma_3)(x) = store(\Gamma_4)(x)$ due to the PARAMIN rule (e.g. both $store(\Gamma_3)$ and $store(\Gamma_4)$ contains same values for dummy variables) or $store(\Gamma_3) =_{\{x\}} store(\Gamma_4)$ vacuously holds. Thus, $store(\Gamma_3) =_{rv(m,\delta_m)} store(\Gamma_4)$.
- Node n is a RAssign node and node m is a Ret node: Since $\Gamma_1 = \mathsf{\Gamma}_1 \cdot (n, \sigma_1)$, $\Gamma_2 = \mathsf{\Gamma}_2 \cdot (n, \sigma_2)$, and Γ_1 and Γ_2 are in the same context, $\mathsf{\Gamma}_1$ and $\mathsf{\Gamma}_2$ are also in the same context. According to the CALL rule in Table 1, $\mathsf{\Gamma}_1$ and $\mathsf{\Gamma}_2$ are configurations for a Call node. Let $m' = node(\mathsf{\Gamma}_1)$ be the Call node in context δ'. According to the PARAMOUT rule in Table 1, m is the matching Ret node of m' such that $m'\ m$ is a balanced sequence of Call and Ret nodes. Then, according to the definition of equivalent stores upto RVs, $stores(\Gamma_1) =_{RV} stores(\Gamma_2)$ implies
$$store(\mathsf{\Gamma}_1) =_{rv(m',\delta')} store(\mathsf{\Gamma}_2) \text{ and } \sigma_1 =_{rv(n,\delta)} \sigma_2.$$
For any RV x in $rv(m, \delta_m)$, either there exists an assignment $x = e$ at n or x is also a RV at $rv(m', \delta')$ according to Case (2) in the definition of RV. In the former case, we have $[\![e]\!]\sigma_1 = [\![e]\!]\sigma_2$ since σ_1 and σ_2 contain same values for any $x \in ref(n) \subseteq rv(n, \delta)$, and consequently $store(\Gamma_3)(x) = store(\Gamma_4)(x)$. According to the PARAMOUT rule in Table 1, $store(\Gamma_3)$ and $store(\Gamma_4)$ contain same values as $store(\mathsf{\Gamma}_1)$ and $store(\mathsf{\Gamma}_2)$, respectively, for all variables that are not updated at n. Thus, in the latter case, we obtain $store(\Gamma_3)(x) = store(\Gamma_4)(x)$ from the relation $store(\mathsf{\Gamma}_1) =_{rv(m',\delta')} store(\mathsf{\Gamma}_1)$.

Thus, $store(\Gamma_3) =_{rv(m,\delta_m)} store(\Gamma_4)$ always holds, and consequently, $stores(\Gamma_3) =_{RV} stores(\Gamma_4)$. □

Lemma 7.15 below states that if the relation $\stackrel{seq}{\sim}$ holds between two configurations Γ_1 and Γ_2 of P_1 and P_2 (i.e. $\Gamma_1 \stackrel{seq}{\sim} \Gamma_2$), and the original program P_1 can make an observable move from Γ_1 in one step, then the sliced program P_2 can make an observable move in one or more steps, and the relation $\stackrel{seq}{\sim}$ also holds between the changed configurations. However, the reverse only holds when S_C is a well-formed strongly control closed set (Lemma 7.16).

LEMMA 7.15. *Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \stackrel{seq}{\sim} \Gamma_2$. If there exists a transition $1 \vdash \Gamma_1 \stackrel{n_1}{\longrightarrow} \Gamma_3$ in P_1, then there exists a transition $2 \vdash \Gamma_2 \stackrel{n_1}{\Longrightarrow} \Gamma_4$ in P_2 such that $\Gamma_3 \stackrel{seq}{\sim} \Gamma_4$.*

PROOF. Let $\Gamma_i = \mathsf{\Gamma}_i \cdot (n_i, \sigma_i)$ be the configuration of node n_i in context δ_i for $1 \leq i \leq 4$, and let $\Gamma = \mathsf{\Gamma} \cdot (n_1, \sigma)$ be another configuration at node n_1 in context δ of program P_2. First, we prove that the labeled transition $2 \vdash \Gamma_2 \stackrel{n_1}{\Longrightarrow} \Gamma_4$ in P_2 exists.

The transition $1 \vdash \Gamma_1 \stackrel{n_1}{\longrightarrow} \Gamma_3$ yields
$$n_1 \in S_C \text{ and } rv(n_1, \delta_1) \neq \emptyset \text{ (Def. 6.1)},$$

and the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$obs(n_1, \delta_1) = obs(n_2, \delta_2) \text{ (Def. 6.5)}.$$

Thus, $obs(n_i, \delta_i) = \{n_1\}$ for $i = 1, 2$ according to the definition of the next observable behavior (Def. 4.9). Moreover, $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$\delta_1 = \mathfrak{d}([n_2..n_1], \delta_2) \text{ (condition (3) in Def. 6.5)}.$$

If $n_2 = n_1$, then $\Gamma_1 \overset{seq}{\sim} \Gamma$ holds such that $\Gamma = \Gamma_2$ and $\delta_2 = \delta_1 = \delta$. If $n_2 \neq n_1$, we show that $\Gamma_1 \overset{seq}{\sim} \Gamma$ also holds.

Assume that $n_2 \neq n_1$. Then, according to Lemma 7.11, we obtain the silent transition $2 \vdash \Gamma_2 \overset{\tau}{\Rightarrow} \Gamma$ from $obs(n_2, \delta_2) = \{n_1\}$. So, there exists a sequence of configurations $\Gamma^1 = \Gamma_2, \ldots, \Gamma^k = \Gamma$ such that for all $1 \leq i \leq k-1$ the silent transition $2 \vdash \Gamma^i \overset{\tau}{\longrightarrow} \Gamma^{i+1}$ holds. Let δ^i be the context of configuration Γ^i for all $1 \leq i \leq k$ such that

$$\delta^1 = \delta_2 \text{ and } \delta^k = \delta = \mathfrak{d}([n_2..n_1], \delta_2).$$

This gives us $\delta = \delta_1$.

Since $obs(n_1, \delta_1) = \{n_1\}$, we infer from the silent transition $2 \vdash \Gamma^i \overset{\tau}{\longrightarrow} \Gamma^{i+1}$ using Lemma 7.6 iteratively for all i in the sequence $k-1, \ldots, 1$ that $obs(node(\Gamma^i), \delta^i) = \{n_1\}$. Since $obs(node(\Gamma^i), \delta^i) \neq \emptyset$ and $\Gamma_1 \overset{seq}{\sim} \Gamma_2$, we infer from the silent transition $2 \vdash \Gamma^i \overset{\tau}{\longrightarrow} \Gamma^{i+1}$ using Lemma 7.10 iteratively for all i in the sequence $1, \ldots, k-1$ that $\Gamma_1 \overset{seq}{\sim} \Gamma^{i+1}$. Thus we infer the relation $\Gamma_1 \overset{seq}{\sim} \Gamma$.

The relation $\Gamma_1 \overset{seq}{\sim} \Gamma$ yields

$$stores(\Gamma_1) =_{RV} stores(\Gamma).$$

As both Γ_1 and Γ are in the same context $\delta = \delta_1$, the top stores of both configurations are equivalent upto RVs, i.e.,

$$\sigma_1 =_{rv(n_1, \delta_1)} \sigma \text{ (according to Def. 6.5)}.$$

As $n_1 \in S_C$ and $rv(n_1, \delta_1) \neq \emptyset$, Case (1) or (3) in the definition of relevant variables (Def. 4.7) are applicable from which we obtain $ref(n_1) \subseteq rv(n_1, \delta_1)$. Thus P_1 and P_2 meet at n_1 in the same context $\delta_1 = \delta$ and agree with the values of all variables in $ref(n_1)$. So, P_1 and P_2 take the same path from n_1 even if n_1 is a Cond node as both evaluate their conditional expressions to the same values. So, if there exists a transition $1 \vdash \Gamma_1 \overset{n_1}{\longrightarrow} \Gamma_3$ in P_1, there exists a configuration $\Gamma_4 = \Gamma_4 \cdot (n_3, \sigma_4)$ such that $n_4 = n_3$ and we obtain the transition $2 \vdash \Gamma \longrightarrow \Gamma_4$ in P_2. Since $n_1 \in S_C$ and $rv(n_1, \delta_1) \neq \emptyset$, $2 \vdash \Gamma \overset{n_1}{\longrightarrow} \Gamma_4$, and consequently, we get the transition $2 \vdash \Gamma_2 \overset{n_1}{\Rightarrow} \Gamma_4$ in P_2.

As both P_1 and P_2 take the same path from n_1 in context $\delta_1 = \delta$ to node n_3, we must have $\delta_3 = \delta_4$ and the relation $obs(n_3, \delta_3) = obs(n_3, \delta_4)$ trivially holds. Also, $\mathfrak{d}([n_3..n], \delta_3) = \mathfrak{d}([n_3..n], \delta_4)$ trivially holds for any $n \in obs(n_3, \delta_3)$. Moreover, since $\Gamma_1 \overset{seq}{\sim} \Gamma$ holds, and the contexts of Γ_1 and Γ are same, we obtain $stores(\Gamma_3) =_{RV} stores(\Gamma_4)$ according to Lemma 7.14. Thus, the relation $\Gamma_3 \overset{seq}{\sim} \Gamma_4$ holds since all three conditions in the definitions of $\overset{seq}{\sim}$ for this relation are satisfied. □

Lemma 7.16. *Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2 such that $\Gamma_1 \overset{seq}{\sim} \Gamma_2$, and let S_C be a well-formed strongly control-closed set. If there exists a transition $2 \vdash \Gamma_2 \overset{n_2}{\longrightarrow} \Gamma_4$ in P_2, then there exists a labeled transition $1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$ in P_1 such that $\Gamma_3 \overset{seq}{\sim} \Gamma_4$ holds.*

Proof. Let $\Gamma_i = \Gamma_i \cdot (n_i, \sigma_i)$ be the configuration of node n_i in context δ_i for $1 \leq i \leq 4$, and let $\Gamma = \Gamma \cdot (n_2, \sigma)$ be another configuration at node n_2 in context δ of program P_1. First, we prove that the labeled transition $1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$ in P_1 exists.

The transition $2 \vdash \Gamma_2 \overset{n_2}{\longrightarrow} \Gamma_4$ yields

$$n_2 \in S_C \text{ and } rv(n_2, \delta_2) \neq \emptyset \text{ (Def. 6.1)}.$$

and the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$obs(n_1, \delta_1) = obs(n_2, \delta_2) \text{ (Def. 6.5).}$$

Thus, $obs(n_i, \delta_i) = \{n_2\}$ for $i = 1, 2$ according to the definition of the next observable behavior (Def. 4.9). Moreover, $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ yields

$$\delta_2 = \mathfrak{d}([n_1..n_2], \delta_1) \text{ (condition (3) in Def. 6.5).}$$

If $n_1 = n_2$, then $\Gamma \overset{seq}{\sim} \Gamma_2$ holds such that $\Gamma = \Gamma_1$ and $\delta_1 = \delta_2 = \delta$. If $n_1 \neq n_2$, we show that $\Gamma \overset{seq}{\sim} \Gamma_2$ also holds.

Assume that $n_1 \neq n_2$. Then, according to Lemma 7.13, we obtain the silent transition $1 \vdash \Gamma_1 \overset{\tau}{\Rightarrow} \Gamma$ from $obs(n_1, \delta_2) = \{n_2\}$. So, there exists a sequence of configurations $\Gamma^1 = \Gamma_1, \ldots, \Gamma^k = \Gamma$ such that for all $1 \leq i \leq k-1$ the silent transition $1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ holds. Let δ^i be the context of configuration Γ^i for all $1 \leq i \leq k$ such that

$$\delta^1 = \delta_1 \text{ and } \delta^k = \delta = \mathfrak{d}([n_1..n_2], \delta_1).$$

This gives us $\delta = \delta_2$.

Since $obs(n_2, \delta_2) = \{n_2\}$, we infer from the silent transition $1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ using Lemma 7.6 iteratively for all i in the sequence $k-1, \ldots, 1$ that $obs(node(\Gamma^i), \delta^i) = \{n_2\}$. Since $obs(node(\Gamma^i), \delta^i) \neq \emptyset$ and $\Gamma_1 \overset{seq}{\sim} \Gamma_2$, we infer from the silent transition $1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ using Lemma 7.9 iteratively for all i in the sequence $1, \ldots, k-1$ that $\Gamma^{i+1} \overset{seq}{\sim} \Gamma_2$. Thus we infer the relation $\Gamma \overset{seq}{\sim} \Gamma_2$.

The relation $\Gamma \overset{seq}{\sim} \Gamma_2$ yields

$$stores(\Gamma) =_{RV} stores(\Gamma_2).$$

As both Γ and Γ_2 are in the same context $\delta = \delta_2$, the top stores of both configurations are equivalent upto relevant variables, i.e.,

$$\sigma =_{rv(n_2,\delta_2)} \sigma_2 \text{ (according to Def. 6.5).}$$

As $n_2 \in S_C$ and $rv(n_2, \delta_2) \neq \emptyset$, Case (1) or (3) in the definition of relevant variables (Def. 4.7) are applicable from which we obtain $ref(n_2) \subseteq rv(n_2, \delta_2)$. Thus, P_1 and P_2 meet at n_2 in the same context $\delta_2 = \delta$ and agree with the values of all variables in $ref(n_2)$. So, P_1 and P_2 take the same path from n_2 even if n_2 is a Cond node as both evaluate their conditional expressions to the same values. So, if there exists a labeled transition $2 \vdash \Gamma_2 \overset{n_2}{\rightarrow} \Gamma_4$ in P_2, there exists a configuration $\Gamma_3 = \Gamma_3 \cdot (n_4, \sigma_3)$ such that $n_4 = n_3$ and we obtain the transition $1 \vdash \Gamma \rightarrow \Gamma_3$ in P_1. Since $n_2 \in S_C$ and $rv(n_2, \delta_2) \neq \emptyset$, $1 \vdash \Gamma \overset{n_2}{\rightarrow} \Gamma_3$ holds, and consequently, we conclude that $1 \vdash \Gamma_1 \overset{n_2}{\Rightarrow} \Gamma_3$.

As both programs P_1 and P_2 take the same path from node n_2 in context $\delta_2 = \delta$ to node n_4, we must have $\delta_3 = \delta_4$ and the relation $obs(n_4, \delta_3) = obs(n_4, \delta_4)$ trivially holds. Also, $\mathfrak{d}([n_4..n], \delta_3) = \mathfrak{d}([n_4..n], \delta_4)$ trivially holds for any $n \in obs(n_4, \delta_4)$. Moreover, since $\Gamma \overset{seq}{\sim} \Gamma_2$ holds, and the contexts of Γ and Γ_2 are same, we obtain $stores(\Gamma_3) =_{RV} stores(\Gamma_4)$ according to Lemma 7.14. Thus, the relation $\Gamma_3 \overset{seq}{\sim} \Gamma_4$ holds since all three conditions in Def. 6.5 for this relation are satisfied. □

Proof of Theorem 6.6. Let Γ_1 and Γ_2 be valid configurations of programs P_1 and P_2.

(1) Assume that $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ holds and there exists a labeled transition $1 \vdash \Gamma_1 \overset{n}{\Rightarrow} \Gamma_1'$. So, there exist a sequence of configurations $\Gamma^1, \ldots, \Gamma^k$ such that $k \geq 1$, $\Gamma^1 = \Gamma_1$, the silent transition $1 \vdash \Gamma^i \overset{\tau}{\rightarrow} \Gamma^{i+1}$ holds for all $i = 1, \ldots, k-1$, and the labeled transition $1 \vdash \Gamma^k \overset{n}{\rightarrow} \Gamma_1'$ holds. Let Γ^i be the configuration of node n_i in context δ_i for all $1 \leq i \leq k$. From the labeled transition $1 \vdash \Gamma^k \overset{n}{\rightarrow} \Gamma_1'$, we obtain $n = n_k \in S_C$ and $rv(n, \delta_k) \neq \emptyset$. Thus, we have $obs(n, \delta_k) = \{n\}$ according to the definition of the next observable behavior (Def. 4.9). We apply Lemma 7.6 iteratively and infer from the above silent transitions for all i in the sequence $k-1, \ldots, 1$

that $obs(n_i, \delta_i) = \{n\}$. Since $obs(n_i, \delta_i)$ is nonempty and $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ holds, we iteratively apply Lemma 7.9 and infer from the above silent transitions for all i in the sequence $1, \ldots, k-1$ that $\Gamma^{i+1} \overset{seq}{\sim} \Gamma_2$ holds. Thus, the relation $\Gamma^k \overset{seq}{\sim} \Gamma_2$ holds. Then, from the labeled transition $1 \vdash \Gamma^k \overset{n}{\longrightarrow} \Gamma_1'$, we infer that there exists Γ_2' such that $\Gamma_1' \overset{seq}{\sim} \Gamma_2'$ and $2 \vdash \Gamma_2 \overset{n}{\Longrightarrow} \Gamma_2'$ holds according to Lemma 7.15.

(2) The proof in (1) is one direction of the bisimulation as it also holds when S_C is a well-formed strongly control closed set. For the other direction, let us assume the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ and the labeled transition $2 \vdash \Gamma_2 \overset{n}{\Longrightarrow} \Gamma_2'$.
Then, there exist a sequence of configurations $\Gamma^1, \ldots, \Gamma^k$ such that $k \geq 1$, $\Gamma^1 = \Gamma_2$, the silent transition $2 \vdash \Gamma^i \overset{\tau}{\longrightarrow} \Gamma^{i+1}$ holds for all $i = 1, \ldots, k-1$, and the labeled transition $2 \vdash \Gamma^k \overset{n}{\longrightarrow} \Gamma_2'$ holds. Let Γ^i be the configuration of node n_i in context δ_i for all $1 \leq i \leq k$. From the labeled transition $2 \vdash \Gamma^k \overset{n}{\longrightarrow} \Gamma_1'$, we obtain $n = n_k \in S_C$ and $rv(n, \delta_k) \neq \emptyset$. Thus, we have $obs(n, \delta_k) = \{n\}$ according to the definition of the next observable behavior. We apply Lemma 7.6 iteratively and infer from the above silent transitions for all i in the sequence $k-1, \ldots, 1$ that $obs(n_i, \delta_i) = \{n\}$. Since $obs(n_i, \delta_i)$ is nonempty and $\Gamma_1 \overset{seq}{\sim} \Gamma_2$ holds, we iteratively apply Lemma 7.10 and infer from the above silent transitions for all i in the sequence $1, \ldots, k-1$ that $\Gamma_1 \overset{seq}{\sim} \Gamma^{i+1}$ holds. Thus, the relation $\Gamma_1 \overset{seq}{\sim} \Gamma^k$ holds. Then, from the labeled transition $2 \vdash \Gamma^k \overset{n}{\longrightarrow} \Gamma_2'$, we infer that there exists Γ_1' such that $\Gamma_1' \overset{seq}{\sim} \Gamma_2'$ and $1 \vdash \Gamma_1 \overset{n}{\Longrightarrow} \Gamma_1'$ holds according to Lemma 7.16.

□

Proof of Theorem 6.7. Let us assume that Γ_0 and Γ_0' are the initial configurations of P_1 and P_2. We can safely assume that $rv(node(\Gamma_0'), \epsilon) = vars(store(\Gamma_0'))$; otherwise any variables that are in $vars(store(\Gamma_0')) \setminus rv(node(\Gamma_0'), \epsilon)$ can simply be discarded from $store(\Gamma_0')$ since the definition of variables that are not relevant does not affect the slicing criterion. We assume that $store(\Gamma_0) =_{rv(node(\Gamma_0'), \epsilon)} store(\Gamma_0')$. This is always possible since we can obtain Γ_0' from Γ_0 by discarding some variable assignments from the store of Γ_0 for variables that are not part of the slice. Moreover, $obs(node(\Gamma_0), \epsilon) = obs(node(\Gamma_0'), \epsilon)$ as $node(\Gamma_0) = node(\Gamma_0')$ (the *start* node) and $\mathfrak{d}([node(\Gamma_0)..m], \epsilon) = \mathfrak{d}([node(\Gamma_0)..m], \epsilon)$ trivially holds for any $m \in obs(node(\Gamma_0), \epsilon)$. Thus, $\Gamma_0 \overset{seq}{\sim} \Gamma_0'$ holds (Def. 6.5). According to Theorem 6.6, $\overset{seq}{\sim}$ is either a *weak simulation* or a *weak bisimulation* relation.

When $\overset{seq}{\sim}$ is a *weak simulation* relation: For all transitions $1 \vdash \Gamma_0 \overset{n}{\Longrightarrow} \Gamma_1$ of P_1, there exists a transition $2 \vdash \Gamma_0' \overset{n}{\Longrightarrow} \Gamma_1'$ of P_2 such that $\Gamma_1 \overset{seq}{\sim} \Gamma_1'$ holds. $1 \vdash \Gamma_0 \overset{n}{\Longrightarrow} \Gamma_1$ implies there exists $\mathsf{\Gamma}_1$ such that

$$1 \vdash \Gamma_0 \overset{\tau}{\Longrightarrow} \mathsf{\Gamma}_1 \text{ and } 1 \vdash \mathsf{\Gamma}_1 \overset{n}{\longrightarrow} \Gamma_1 \text{ (Def. 6.1)} .$$

Similarly, $2 \vdash \Gamma_0' \overset{n}{\Longrightarrow} \Gamma_1'$ implies there exists $\mathsf{\Gamma}_1'$ such that

$$2 \vdash \Gamma_0' \overset{\tau}{\Longrightarrow} \mathsf{\Gamma}_1' \text{ and } 2 \vdash \mathsf{\Gamma}_1' \overset{n}{\longrightarrow} \Gamma_1'.$$

According to the definition of labeled transition, for the contexts δ and δ' of configurations $\mathsf{\Gamma}_1$ and $\mathsf{\Gamma}_1'$,

- $1 \vdash \mathsf{\Gamma}_1 \overset{n}{\longrightarrow} \Gamma_1$ implies $n \in S_C$, and $rv(n, \delta) \neq \emptyset$, and
- $2 \vdash \mathsf{\Gamma}_1' \overset{n}{\longrightarrow} \Gamma_1'$ implies $n \in S_C$, and $rv(n, \delta') \neq \emptyset$.

In both cases, $node(\mathsf{\Gamma}_1) = node(\mathsf{\Gamma}_1') = n$, and the transitions are S_C-observable moves. According to the definition of next observable behavior, we obtain $obs(n, \delta) = \{n\}$. Let $1 \vdash \Gamma_0 \overset{\tau}{\Longrightarrow} \mathsf{\Gamma}_1$ be the

sequence of silent transitions

$$1 \vdash \Gamma_{(1,0)} \xrightarrow{\tau} \Gamma_{(1,1)}, \ldots, 1 \vdash \Gamma_{(1,l-1)} \xrightarrow{\tau} \Gamma_{(1,l)}$$

such that $\Gamma_{(1,0)} = \Gamma_0$ and $\Gamma_{(1,l)} = \Gamma_1$. Let $\Gamma_{(1,j)}$ be the configuration at node m_j in context $\delta^j = \mathfrak{d}([node(\Gamma_0)..m_j], \epsilon)$ for all $0 \leq j \leq l$.

By applying Lemma 7.6 on the sequence of silent transitions $1 \vdash \Gamma_{(1,j)} \xrightarrow{\tau} \Gamma_{(1,j+1)}$ for all j in the sequence $l-1, \ldots, 0$, we infer

$$obs(m_j, \delta^j) = \{n\}$$

from $obs(n, \delta) = \{n\}$. Since $\Gamma_0 \overset{seq}{\sim} \Gamma_0'$ holds and $obs(m_j, \delta_j) \neq \emptyset$ for all m_j and δ^j as above, by repeatedly applying Lemma 7.9 on the sequence of above silent transitions for all j in the sequence $0, \ldots, l-1$, we infer that the relation $\Gamma_1 \overset{seq}{\sim} \Gamma_0'$ holds.

Similarly, let $2 \vdash \Gamma_0' \overset{\tau}{\Rightarrow} \Gamma_1{}'$ be the sequence of silent transitions

$$2 \vdash \Gamma_{(2,0)} \xrightarrow{\tau} \Gamma_{(2,1)}, \ldots, 2 \vdash \Gamma_{(2,t-1)} \xrightarrow{\tau} \Gamma_{(2,t)}$$

such that $\Gamma_{(2,0)} = \Gamma_0'$ and $\Gamma_{(2,t)} = \Gamma_1{}'$. Since $node(\Gamma_1{}') = n$, $obs(n, \delta) = \{n\}$, and $\Gamma_1 \overset{seq}{\sim} \Gamma_0'$ holds, by applying Lemma 7.10 on this sequence of transitions, we infer the relation $\Gamma_1 \overset{\mathbf{seq}}{\sim} \Gamma_1{}'$. Then, it yields

$$stores(\Gamma_1) =_{RV} stores(\Gamma_1{}') \text{ and } \delta = \mathfrak{d}([node(\Gamma_0)..n], \epsilon) = \mathfrak{d}([node(\Gamma_0')..n], \epsilon).$$

Thus, the contexts δ and δ' of configurations Γ_1 and $\Gamma_1{}'$ are same, and we conclude $store(\Gamma_1) =_{rv(n,\delta)} store(\Gamma_1{}')$ from the definition of equivalent stores upto RVs (Def. 6.3). As above, $\Gamma_1 \overset{seq}{\sim} \Gamma_1{}'$ leads to the existence of configurations Γ_2 and Γ_2' at node n_1 in context δ_1 such that $n_1 \in S_C$, $n_1 = node(\Gamma_2) = node(\Gamma_2')$, and $store(\Gamma_2) =_{rv(n_1,\delta_1)} store(\Gamma_2')$, and this process continues.

Thus, we get a (finite or infinite) sequence of configurations $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$ of P_1 and $\Gamma_0', \Gamma_1', \Gamma_2', \ldots$ of P_2 that visit the nodes in S_C in a pairwise fashion, and have the same values for RVs. As $nodes(C) \subseteq S_C$, if P_1 makes a C-observable move (i.e. visits a node in C), the sliced program P_2 will make the same C-observable move. The set of relevant variables includes the variables of interest stated in the slicing criterion (according to Def. 4.7). Thus, P_1 and P_2 visit C-observable nodes in a pairwise fashion finitely or infinitely many times observing the same values for the variables stated in C. Thus, $P_1 \gtrsim_C P_2$ holds.

When $\overset{seq}{\sim}$ is a *weak bisimulation* relation: the above result also holds here. Moreover, for all transitions $2 \vdash \Gamma_0' \overset{n}{\Rightarrow} \Gamma_1'$ of P_2, there exists a transition $1 \vdash \Gamma_0 \overset{n}{\Rightarrow} \Gamma_1$ of P_1 such that $\Gamma_1 \overset{seq}{\sim} \Gamma_1'$ holds, and this process continues. By following the symmetric reasoning as above, if P_2 makes a C-observable move finitely or infinitely many times, program P_1 will make the same C-observable move and values of relevant variables agree. Thus, $P_2 \gtrsim_C P_1$, and consequently, $P1 \simeq_C P2$. □

8 RELATED WORK

8.1 Semantics of slicing

The earliest work on the semantics of slicing by Weiser [50, 51] was based on finite trajectories, where each trajectory is a finite sequence of (statement label, state) pairs. Weiser then defines when a program is a static backward slice of an original program. Binkley et al. [7, 8] developed a formal framework for comparing different forms of slicing using the program projection theory. They used two relations over programs: a syntactic *pre-order* relation that defines the syntactic property that the particular form of slicing seeks to optimise, and an *equivalence* relation that captures the semantic property that the slice wants to preserve. These relations are used to formally compare different forms of slicing. For example, *static backward equivalence* is used to preserve the semantic relationship of backward static slicing originally defined by Weiser. However, the

slice semantics is provided only for terminating programs as the trajectories are finite. In case of slicing nonterminating programs, Ward and Zedan [48] illustrate that Weiser slicing allows us to delete nonterminating code and code which appears after a nonterminating loop. This semantics is not reflected by the static backward equivalence and thus only partially defines Weiser slicing. Ward and Zedan [48] presented an operational and an equivalent denotational semantics for slicing nonterminating and nondeterministic programs that capture Weiser slicing. They did not consider nontermination sensitive slicing in their semantics. The semantics provided by Reps and Yang includes a slicing theorem and a termination theorem [41]; the slicing theorem is another illustration of Weiser's semantics but demonstrated over the program dependence graph. Several non-standard semantics of program slicing are described in [10, 17, 32], and the shortcomings of these approaches were shown by Barraclough et al. [5]. The non-standard semantics in [10, 17] suffers from lack of substitutivity in which a segment of a program cannot be substituted by another, semantically equivalent part. The transfinite semantics approach [17, 32] has problems when the program flow after an infinite loop depends on values computed in the loop, and these non-standard semantics do not correspond to normal execution of programs on a von Neumann machine. Barraclough et al. [5] extended the semantics of Weiser for nonterminating programs but based on finite trajectories. However, as Ward and Zedan [48] uncovered, all these non-standard semantics [10, 17, 32] including the extended semantics of Barraclough et al. [5] have the serious problem of accepting a nonterminating program as a valid slice of a terminating program. In contrast, we provide an operational style semantics for programs, and we capture the semantic relation between a program and its slice through a *semi-equivalence* relation that considers infinite execution traces of possibly nonterminating programs. Moreover, as Theorem 3.10 demonstrates, a nontermination insensitive slice (and hence a nontermination sensitive slice which is also nontermination insensitive) preserves the nondeterministic behavior of the original program.

8.2 Correctness of slicing

Reps and Yang provided the first formal proof of correctness, showing that slices extracted from the program dependence graph have the desired semantic properties [41]. The proof is based on walking backward over the edges of the PDG, which is generated from programs with structured control flow. This work was extended by Ball and Horwitz who proved slicing correctness for arbitrary control flow [4]. The kind of slice considered in these proofs is general in the sense that the statements in the slice may have a different relative order than in the original program.

Our correctness proof of dependence-based slicing is based on a (bi)simulation that walks over the edges of the CFG, and we require that the relative order of statements in the sliced and in the original code should be the same. While the previous approach of correctness proofs considered slices extracted from PDG, the (bi)simulation approach is applicable to a more general framework.

In [2, 39], the correctness arguments are built by capturing various notions of control dependencies for intraprocedural programs, and demonstrate a simulation or bisimulation relation between the original and the sliced programs. In [19], a bisimulation-based correctness property is provided for multi-threaded programs.

Our results hold for interprocedural, possibly nonterminating programs, and our proofs considered that control dependence relation is captured by the well-formed weak or strong control closure relation. These relations are the generalisations of all previously defined control dependence relations found in the literature, and hence our proof framework is valid for a whole range of existing control dependence relations.

8.3 Dependence-based slicing

Since the pioneering work of slicing by Weiser [50, 51] most static slicing techniques are based on computing the PDG, which contains all data and control dependencies. In [6, 20–22, 34, 40], program dependencies are represented in a graph and the problem of computing the slice set is viewed as a graph reachability problem. In [28, 50], all the dependencies and slices are computed by solving flow equations on CFGs. All these techniques are different with regard to precision, complexity, applicability, and scalability. Very little work has been done on static backward slicing computing partial dependencies. In [26], a static demand-driven slicing technique is provided for interprocedural code. Khanfar et al. [24, 25] presented an on-demand interprocedural slicing algorithm that is very efficient for structured programs, and uses a program representation that is tailored to such programs.

There are other approaches to program slicing that do not track the dependencies explicitly and produce smaller slices than dependency tracking algorithms. For example, the semantic slicing approach presented in [30, 45, 47] is based on the theory of program transformation, and the abstract slicing approach in [31, 42] is based on the theory of abstract interpretation [12].

8.4 Interprocedural control dependency

Sinha et al. [44] defined interprocedural control dependency at the level of *interprocedural inlined flow graph* (IIFG) which is possibly an infinite CFG (for recursive programs) with a unique end node that results when all procedure calls are inlined at the call sites. Postdominator-based intraprocedural control dependency relation computed at the IIFG graph provides the interprocedural control dependence. The size of the resulting IIFG may limit obtaining an efficient and precise practical method to compute interprocedural dependency. However, the authors provided a less precise but efficient statement-based approach summarizing all calling contexts for a statement by using *augmented control dependency graph* (ACDG) for each procedure, connecting all ACDGs to construct an interprocedural control dependence graph (ICDG), and finally, traversing the ICDG to compute interprocedural control dependences. Loyall and Mathisen [27] combined intraprocedural CFGs by connecting all call nodes to the procedure entry and exit nodes to form the interprocedural CFG and used postdominator-based control dependence relation to define interprocedural control dependence. In this article, we have provided the well-formed weak and strong control closure relation for interprocedural program which are non-termination insensitive and non-termination sensitive. These relations are the extensions of the generalised form of control dependence by Danicic et al. [13] provided for intraprocedural programs.

9 CONCLUSION AND FUTURE WORK

We have developed a framework for proving the correctness of static backward slicing of interprocedural programs. Our framework is based on control-flow graphs, where program slices are defined through reachability in what is basically a program dependence graph formed by the data and control dependency relations. We define an operational semantics for interprocedural CFG's in terms of transitions between configurations representing the current state of the program. Based on this semantics, we define semi-equivalences which are relations between the original program and its slice provided that the slice has the desired properties. A slice is thus correct if a semi-equivalence relation holds between the original program and the slice.

The semi-equivalence relations actually come in two different varieties. The first one requires that the slice should always be able to "mimic" the original program as regards the "observable" behaviours, i.e., the order of visiting CFG nodes in the slicing criterion should be the same, as well as the values of the variables in the slicing criterion during each visit. The second in addition

demands that the original program should be able to mimic the slice. In the latter case we say that the slice is nontermination-sensitive.

We show that if a certain simulation (or bisimulation) relation holds between certain configurations of the original program and its slice, then they are semi-equivalent. The relation is based on the data and control dependency relations. The control dependency relation is captured by the well-formed weak and strong control closure relations that are nontermination insensitive and -sensitive, respectively. Danicic et al. [13] proved that all control dependence relations defined in previous works are special cases of control dependence captured by the weak and strong control closure. We define the well-formed weak and strong control closure, which are the interprocedural extensions of the generalised weak/strong control closure. We show that if the control dependency relation is nontermination-insensitive then the relation is a weak simulation, and if it is nontermination-sensitive then the relation is a weak bisimulation.

Our work is inspired by Amtoft et al. [2, 39] who used a similar technique with weak (bi)simulation for proving the correctness of slicing for intraprocedural programs. Our contributions are the extension to interprocedural programs using the the well-formed weak and strong control closure relations, which capture all previously defined control dependency relations, rather than proving the correctness for some fixed control dependence relation. The latter is interesting since a number of control dependency relations have been defined in the literature.

Future work includes investigating how to apply the framework to prove the correctness of concrete slicing algorithms. In particular demand-driven algorithms, where some of the dependencies are not explicitly computed [25, 26], provide an interesting challenge.

ACKNOWLEDGMENTS

This research has been supported by the Knowledge Foundation through the SPACES and HERO projects. We thank the anonymous reviewers whose constructive comments have greatly improved this article.

REFERENCES

[1] Matthew Allen and Susan Horwitz. 2003. Slicing Java Programs That Throw and Catch Exceptions. *SIGPLAN Not.* 38, 10 (June 2003), 44–54. https://doi.org/10.1145/966049.777394

[2] Torben Amtoft. 2007. *Correctness of practical slicing for modern program structures.* Technical Report. Department of Computing and Information Sciences, Kansas State University.

[3] Torben Amtoft. 2008. Slicing for modern program structures: a theory for eliminating irrelevant loops. *Inf. Process. Lett.* 106, 2 (2008), 45–51.

[4] Thomas Ball and Susan Horwitz. 1993. Slicing Programs with Arbitrary Control-flow. In *Proc. First International Workshop on Automated and Algorithmic Debugging (AADEBUG '93).* Springer-Verlag, London, UK, UK, 206–222. http://dl.acm.org/citation.cfm?id=646902.710193

[5] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Ákos Kiss, Mike Laurence, and Lahcen Ouarbya. 2010. A trajectory-based strict semantics for program slicing. *Theor. Comput. Sci.* 411, 11-13 (2010), 1372–1386.

[6] David Binkley. 1993. Precise Executable Interprocedural Slices. *ACM Lett. Program. Lang. Syst.* 2, 1-4 (March 1993), 31–45. https://doi.org/10.1145/176454.176473

[7] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. 2006. A Formalisation of the Relationship Between Forms of Program Slicing. *Sci. Comput. Program.* 62, 3 (Oct. 2006), 228–252. https://doi.org/10.1016/j.scico.2006.04.007

[8] Dave Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Ákos Kiss, and Bogdan Korel. 2006. Theoretical Foundations of Dynamic Program Slicing. *Theor. Comput. Sci.* 360, 1 (Aug. 2006), 23–41.

[9] David W. Binkley and Keith Brian Gallagher. 1996. Program Slicing. *Advances in Computers* 43 (1996), 1–50. https://doi.org/10.1016/S0065-2458(08)60641-5

[10] Robert Cartwright and Matthias Felleisen. 1989. The Semantics of Program Dependence. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, Richard L. Wexelblat (Ed.). ACM, 13–27.

[11] A. Chaturvedi and D. Binkley. 2018. Web Service Slicing: Intra and Inter-Operational Analysis to Test Changes. *IEEE Transactions on Services Computing* (2018), 1–1. https://doi.org/10.1109/TSC.2018.2821157

[12] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (proc. POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[13] S. Danicic, R. Barraclough, M. Harman, J. D. Howroyd, Á. Kiss, and M. Laurence. 2011. A unifying theory of control dependence and its application to arbitrary program structures. *Theoretical Computer Science* 412, 49 (2011), 6809–6842.

[14] Sebastian Danicic, Mark Harman, John Howroyd, and Lahcen Ouarbya. 2007. A non-standard semantics for program slicing and dependence analysis. *The Journal of Logic and Algebraic Programming* 72, 2 (2007), 191 – 206. https://doi.org/10.1016/j.jlap.2007.02.010 Programming Language Interference and Dependence.

[15] Jakob Engblom. 1999. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99), Atlanta, Georgia, USA, May 5, 1999*, Y. Annie Liu and Reinhard Wilhelm (Eds.). ACM, 96–103. https://doi.org/10.1145/314403.314470

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[17] Roberto Giacobazzi and Isabella Mastroeni. 2003. Non-Standard Semantics for Program Slicing. *Higher-Order and Symbolic Computation* 16, 4 (2003), 297–339.

[18] David Harel. 1981. On the total correctness of nondeterministic programs. *Theoretical Computer Science* 13, 2 (1981), 175 – 192.

[19] John Hatcliff, James C. Corbett, Matthew B. Dwyer, Stefan Sokolowski, and Hongjun Zheng. 1999. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proceedings of the 6th International Symposium on Static Analysis (SAS '99)*. Springer-Verlag, London, UK, UK, 1–18. http://dl.acm.org/citation.cfm?id=647168.718134

[20] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. ACM, New York, NY, USA, 35–46. https://doi.org/10.1145/53990.53994

[21] C. R. Chou J. C. Hawang, M. W. Du. 1988. Finding Program Slices for recursive procedures. In *Proc. 12th Annual International Computer Software and Applications Conference (COMPSAC '88)*. Chicago, 220–227.

[22] Daniel Jackson and Eugene J. Rollins. 1994. A New Model of Program Dependences for Reverse Engineering. In *Proc. 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*. ACM, New York, NY, USA, 2–10. https://doi.org/10.1145/193173.195281

[23] Andreas Johnsen, Kristina Lundqvist, Paul Pettersson, and Kaj Hänninen. 2014. Regression Verification of AADL Models through Slicing of System Dependence Graphs. In *Tenth International ACM Sigsoft Conference on the Quality of Software Architectures*. ACM, 103–112. http://www.es.mdh.se/publications/3535-

[24] Husni Khanfar and Björn Lisper. 2016. Enhanced PCB Based Slicing. In *Fifth International Valentin Turchin Workshop on Metacomputation*.

[25] Husni Khanfar, Björn Lisper, and Abu Naser Masud. 2015. Static Backward Program Slicing for Safety Critical Systems. In *The 20th International Conference on Reliable Software Technologies*. 50–65.

[26] Björn Lisper, Abu Naser Masud, and Husni Khanfar. 2015. Static Backward Demand-Driven Slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation (PEPM '15)*. ACM, New York, NY, USA, 115–126.

[27] J. P. Loyall and S. A. Mathisen. 1993. Using dependence analysis to support the software maintenance process. In *1993 Conference on Software Maintenance*. 282–291.

[28] James Robert Lyle. 1984. *Evaluating Variations on Program Slicing for Debugging (Data-flow, Ada)*. Ph.D. Dissertation. College Park, MD, USA. AAI8514553.

[29] C. Mao. 2009. Slicing Web service-based software. In *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. 1–8. https://doi.org/10.1109/SOCA.2009.5410460

[30] Ward Martin, Zedan Hussein, Ladkau Matthias, and Natelberg Stefan. 2008. Conditioned semantic slicing for abstraction; industrial experiment. *Software: Practice and Experience* 38, 12 (2008), 1273–1304. https://doi.org/10.1002/spe.869

[31] Isabella Mastroeni and Damiano Zanardini. 2017. Abstract Program Slicing: An Abstract Interpretation-Based Approach to Program Slicing. *ACM Trans. Comput. Logic* 18, 1, Article 7 (Feb. 2017), 58 pages. https://doi.org/10.1145/3029052

[32] Härmel Nestra. 2005. Transfinite semantics in program slicing. In *Proceedings of the Estonian Academy of Sciences (Engineering)*. 313–328.

[33] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[34] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. *SIGSOFT Softw. Eng. Notes* 9, 3 (April 1984), 177–184. https://doi.org/10.1145/390010.808263

[35] Keshav Pingali and Gianfranco Bilardi. 1997. Optimal Control Dependence Computation and the Roman Chariots Problem. *ACM Trans. Program. Lang. Syst.* 19, 3 (May 1997), 462–491.

[36] A. Podgurski and L. A. Clarke. 1990. A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance. *IEEE Trans. Softw. Eng.* 16, 9 (Sept. 1990), 965–979.

[37] Reese T. Prosser. 1959. Applications of Boolean Matrices to the Analysis of Flow Diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference (IRE-AIEE-ACM '59 (Eastern))*. ACM, New York, NY, USA, 133–138.

[38] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. 2005. A New Foundation for Control Dependence and Slicing for Modern Program Structures. In *European Symposium on Programming (LNCS)*, Vol. 3444. Springer-Verlag, 77–93.

[39] Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B. Dwyer. 2007. A New Foundation for Control Dependence and Slicing for Modern Program Structures. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 27 (Aug. 2007).

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Proc. POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[41] Thomas W. Reps and Wuu Yang. 1989. The Semantics of Program Slicing and Program Integration. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL) (Lecture Notes in Computer Science)*. Springer, 360–374.

[42] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. 2005. Abstract Slicing: A New Approach to Program Slicing Based on Abstract Interpretation and Model Checking. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '05)*. IEEE Computer Society, Washington, DC, USA, 25–34. https://doi.org/10.1109/SCAM.2005.2

[43] Josep Silva. 2012. A Vocabulary of Program Slicing-based Techniques. *ACM Comput. Surv.* 44, 3, Article 12 (June 2012), 41 pages. https://doi.org/10.1145/2187671.2187674

[44] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. 2001. Interprocedural Control Dependence. *ACM Trans. Softw. Eng. Methodol.* 10, 2 (April 2001), 209–254.

[45] Martin Ward and Hussein Zedan. 2007. Slicing As a Program Transformation. *ACM Trans. Program. Lang. Syst.* 29, 2, Article 7 (April 2007). https://doi.org/10.1145/1216374.1216375

[46] Martin P. Ward. 2001. The Formal Transformation Approach to Source Code Analysis and Manipulation. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*. IEEE Computer Society, 187–195. https://doi.org/10.1109/SCAM.2001.972680

[47] Martin P. Ward. 2003. Slicing the SCAM Mug: A Case Study in Semantic Slicing. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 88–97. https://doi.org/10.1109/SCAM.2003.1238035

[48] Martin P. Ward and Hussein Zedan. 2017. The formal semantics of program slicing for nonterminating computations. *Journal of Software: Evolution and Process* 29, 1 (2017).

[49] Mark Weiser. 1981. Program Slicing. In *Proc. 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[50] Mark Weiser. 1984. Program Slicing. *IEEE Trans. Software Eng.* 10, 4 (1984), 352–357.

[51] Mark David Weiser. 1979. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* Ph.D. Dissertation. Ann Arbor, MI, USA. AAI8007856.