Abstract Program Slicing on Dependence Condition Graphs

Raju Halder and Agostino Cortesi

Università Ca' Foscari Venezia, Italy
{halder, cortesi}@unive.it

Abstract

Many slicing techniques have been proposed based on the traditional Program Dependence Graph (PDG) representation. In traditional PDGs, the notion of dependency between statements is based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Mastroeni and Zanardini first introduced the notion of semantics-based data dependency, both at concrete and abstract domains, that helps in converting the traditional syntactic PDGs into more refined semantics-based (abstract) PDGs by disregarding some false dependences from them. As a result, the slicing techniques based on these semantics-based (abstract) PDGs result into more precise slices. In this paper, we strictly improve this approach by (*i*) introducing the notion of semantic relevancy of statements, and (*ii*) combining it with conditional dependency. This allows us to transform syntactic PDGs into semantics-based (abstract) Dependence Condition Graphs (DCGs) that enable to identify the conditions for dependences between program points.

Keywords: Dependence Graph, Program Slicing, Abstract Interpretation

1 Introduction

Program slicing is a well-known decomposition technique that extracts from programs the statements which are relevant to a given behavior. It is a fundamental operation for addressing many software-engineering problems, including program understanding, debugging, maintenance, testing, parallelization, integration, software measurement etc. See, for instance, [6, 14, 15, 18, 23, 25, 26, 28, 31, 33, 34]. The notion of program slice was originally introduced by Mark Weiser [42] who defines a static program slice as any executable subset of program statements that preserves the behavior of the original program at a particular program point for a subset of program variables for all program inputs. Therefore, the static slicing criteria is denoted by $\langle p, v\rangle$ where p is the program point and v is the variable of interest. This is not restrictive, as it can

easily be extended to slicing with respect to a set of variables V, formed from the union of the slices on each variable in V. In contrast, in dynamic slicing [24], programmers are more interested in a slice that preserves the program's behavior for a specific program input rather than for all program inputs: the dependences that occur in a specific execution of the program are taken into account. Therefore slicing criteria for dynamic slicing is denoted by $\langle p, v, i \rangle$, where i represents the input sequence of interest.

Program slicing can be defined in concrete as well as in abstract domain [29], where in the former case we consider exact values of the program variables of interest, while in the latter case we consider some properties instead of their exact values. These properties are represented as abstract domains of the variable domains in the context of Abstract Interpretation [10, 16]. The notion of Abstract Program Slicing was first introduced by Hong, Lee and Sokolsky [37] where the abstract interpretation is considered in the restricted area of predicate abstraction. Some recent works on abstract program slicing, although few, includes property-driven program slicing [3, 7], semantics-based abstract program slicing [29, 30] etc. Abstract slicing helps in finding all statements affecting some particular properties of the variables of interest. For instance, suppose some variables at some point of execution are not resulting the correct properties (positive value, say) as desired. In such case, abstract slicing is considered as an effective way to identify the statements that are responsible for this error.

The original static slicing algorithm by Mark Weiser [42] is expressed as a sequence of data-flow analysis problems and the influence of predicates on statement execution, while Korel and Lasky [24] extended it into the dynamic context and proposed an iterative dynamic slicing algorithm based on dynamic data flow and control influence. Both Weiser's and Korel-Laski's algorithms produce executable form [41] of slices that are easier to fit into a theoretical framework by Binkley [4] which is based on projection theory [19] and provides relationship between different well-known forms of slicing by exploiting the requirements of syntactic and semantic equivalence between the slice and the original program. Recently, Mastroeni and Nicolić [29] extended the theoretical framework of Binkley [4] to the abstract domain in order to define abstract slicing, and to represent and compare different forms of slicing in abstract domain. Slicing criteria in an abstract domain, thus, includes the observable properties of variables of interest as well. For instance, static and dynamic abstract slicing criteria are denoted by $\langle p, v, \rho \rangle$ and $\langle p, v, i, \rho \rangle$ respectively, where ρ is the observable property of v.

All the techniques mentioned so far make use (explicitly or implicitly) of the notion of Program Dependence Graph (PDG) [13, 27, 33]. However, different forms of PDG representations have been proposed, depending on the intended applications [5, 21]. Over the last 25 years, many PDG-based program slicing techniques have been proposed [1, 17, 22, 32, 34, 36, 38]. In general, a PDG makes explicit both the data and control dependences for each operation in a program. Data dependences have been used to represent only the relevant data flow relationship of a program, while control dependences are derived

from the actual control flow graph and represent only the essential control flow relationships of a program. However, PDG-based slicing is somewhat restricted: a slice must be taken *w.r.t.* variables that are defined or used at that program point.

In traditional PDGs, the notion of dependency between statements is based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependences. Mastroeni and Zanardini [30] introduce semantic data dependency which fills up the existing gap between syntax and semantics. The semantic data dependency, which is computed for each expression in the program over the states possibly reaching these program points, helps in obtaining a more precise semantics-based PDG by removing the false dependences from the traditional syntactic PDG. For instance, although the expression "$e = x + 4w \bmod 2$" syntactically depends on w, but semantically there is no dependency as the evaluation of "$4w \bmod 2$" is always zero. So, we can remove this dependency from the syntactic PDG, yielding a more precise semantics-based PDG. The semantic data dependency can also be lifted to an abstract domain where dependences are computed with respect to some specific properties of interest rather than concrete values. This is the basis to design abstract semantics-based slicing algorithms aimed at identifying the part of the programs which is relevant with respect to a property (not necessarily the exact values) of the variables at a given program point.

Sukumaran et al. [39] present a refinement of the traditional PDGs of programs, called Dependence Condition Graph (DCG), based on the notion of conditional dependency. The DCG is built from a PDG by annotating each edge e of the PDG with conditional information $e^b = \langle e^R, e^A \rangle$ under which a particular dependence actually arises in a program execution. The first part e^R is referred to as *Reach Sequences* which represents the conditions that should be true for an execution to ensure that the target $e.tgt$ of e must be executed once the source $e.src$ is executed for a control edge e, and the target $e.tgt$ is reached from the source $e.src$ for a data edge e. The component e^A is referred to as *Avoid Sequences* which is only relevant for data edges (it is $\emptyset$ for control edges) and captures the possible conditions under which the assignment at $e.src$ can get overwritten before it reaches $e.tgt$. So, the conditions in e^A must not hold for an execution to ensure that the variable defined at $e.src$ must used at $e.tgt$.

This report provides two main contributions in this area. The first one is the introduction of the notion of semantic relevancy of statements *w.r.t.* a property. It determines whether a statement is relevant *w.r.t.* a property of interest, and is computed over all concrete (or abstract) states possibly reaching the statement. For instance, consider the following code fragment: {(1) x = *input*; (2) $x = x + 2$; (3) *print* x;}. If we consider an abstract domain of parity represented by $PAR = \{\top, ODD, EVEN, \bot\}$, we see that the variable x at program point 1 may have any parity from the set $\{ODD, EVEN\}$, and the execution of the statement at program point 2 does not change the parity of x at all. Therefore, the statement at 2 is semantically irrelevant *w.r.t.* PAR. By disregarding all the nodes that correspond to irrelevant statements *w.r.t.* concrete (or abstract) property from

a syntactic PDG, we obtain a more precise semantics-based (abstract) PDG. Observe that the combined effort of semantic relevancy of statements with the expression-level semantic data dependency introduced by Mastroeni and Zanardini [30] guarantees a more precise semantics-based (abstract) PDG.

The second contribution of the report is the refinement of the semantics-based PDGs obtained so far by applying the notion of conditional dependency proposed by Sukumaran et al. [39]. This allows us to transform PDGs into Dependence Condition Graphs (DCGs) that enable to identify the conditions for dependences between program points. We lift the semantics of DCGs from concrete domain to an abstract domain of interest. The satisfiability of the conditions in DCGs by (abstract) execution traces helps in removing semantically unrealizable dependences from them, yielding to refined semantics-based (abstract) DCGs.

These two contributions in combination with semantic data dependency [30] lead to an abstract program slicing algorithm that strictly improves with respect to the literature. The algorithm constructs a semantics-based abstract DCG from a given program by combining these three notions: (*i*) semantic relevancy of statements, (*ii*) semantic data dependency at expression level [30], and (*iii*) conditional dependency based on DCG annotations [39].

The rest of the report [1] is organized as follows: Section 2 recalls some basic background. Section 3 provides an example that motivates us to propose a further refinement of the existing semantics-based abstract PDGs into semantics-based abstract DCGs by combining the notion of statement relevancy, semantic data dependency, and conditional dependency. Section 4 introduces how to compute semantic relevancy of statements and blocks *w.r.t.* a concrete/abstract property, and discusses the way to treat the relevancy of control statements too. The atomicity property of the abstract states, as discussed in Section 5, is a crucial requirement while computing the statements relevancy. In section 6, we formalize an algorithm to construct semantics-based abstract PDG from a given program. The optimization of the relevancy and data dependency computation is discussed in section 7. In section 8, we lift the semantics of DCGs from concrete to an abstract domain of interest, and we propose a refinement of syntactic DCCs into semantics-based abstract DCGs. The proposed abstract slicing algorithm is formalized in section 9. The idempotancy of the proposal is discussed in section 10. In section 11, we prove the soundness and provide an overall complexity analysis of the proposal. Section 12 concludes the paper.

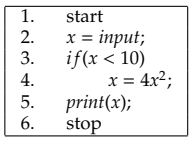
2 Preliminaries

In this Section, first we recall some basic ideas about PDG representation of programs in Static Single Assignment (SSA) form, PDG-based slicing techniques, and Dependence Condition Graph (DCG) representation; then we recall basic

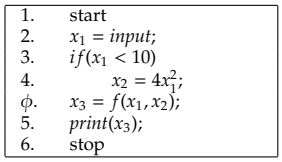
[1] The report is a revised and extended version of [8]

ideas about the Abstract Interpretation theory covering abstract semantics of expressions, statements and the induced partitioning of abstract domains.

Static Single Assignment (SSA). The SSA form [11] of a program is a semantically equivalent version of the program where each variable has a unique (syntactic) assignment. The SSA form introduces special ϕ-assignments at join nodes of the program where assignments to the same variable along multiple program paths may converge. Each assignment to a variable is given a unique name, and all of the uses reached by that assignment are renamed to match the assignment's new name. Figure 1(a) and 1(b) depict a program P and its SSA form P_{ssa} respectively. Observe that the operands to the ϕ-function indicate which assignments to x (x_1 or x_2) reach the join point. The subsequent uses of x become uses of x_3. Thus, there is only one assignment to x_i in the entire program.



(a) *Program P*



(b) P_{ssa}: *SSA form of P*

Figure 1: Program P and its SSA form P_{ssa}

Program Dependence Graph. Program Dependence Graph (PDG) [13, 27, 33, 40] for a program is a directed graph with vertices denoting program components (*start*, *stop*, *skip*, *assignment*, *conditional* or *repetitive* statements) and edges denoting dependences between components. An edge represents either *control dependence* or *data dependence*. The sub-graph of the PDG induced by the control dependence edges is called control dependence graph (CDG) and the sub-graph induced by the data dependence edges is called data dependence graph (DDG). The source node of a CDG edge corresponds to either *start* or *conditional* or *repetitive* statement. Any CDG edge e whose source node $e.src$ corresponds to *start* statement is denoted by an unlabeled edge $e = e.src \rightarrow e.tgt$, whereas any CDG edge e whose source node $e.src$ corresponds to *conditional* or *repetitive* statement is denoted by a labeled edges $e = e.src \xrightarrow{lab} e.tgt$, where $lab \in \{true, false\}$. In the former case, the condition represented by $e.src$ is implicitly *true* which means that during an execution once $e.src$ executed, then its target $e.tgt$ will eventually be executed, while in latter case, $e.tgt$ is lab-control dependent on $e.src$ which means that whenever the condition represented by $e.src$ is evaluated and its value matches the label lab, then its target node represented by $e.tgt$ will be executed, if the program terminates. A DDG edge is denoted by $e = e.src \xrightarrow{x} e.tgt$, representing that the target node $e.tgt$ is data

dependent on the source node *e.src* for a variable x. The PDG representation of the program P_{ssa} (Figure 1(b)) is depicted in Figure 2.

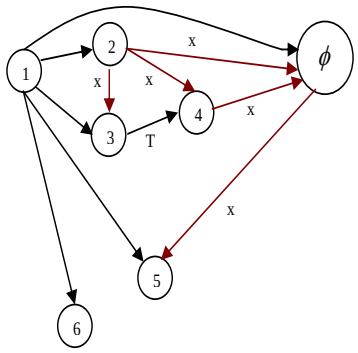


Figure 2: PDG of P_{ssa}

PDG-based Slicing. The results of the program dependence graph discussed so far have an impact on different forms of static slicing: backward slicing [42], forward slicing [2], and chopping [23, 35]. The backward slice with respect to variable v at program point p consists of those program points that affect v directly or indirectly. Forward slicing is the dual of backward slicing. The forward slice with respect to variable v at program point p consists of those program points that are affected by v. Chopping is a combination of both backward and forward slicing. A slicing criterion for chopping is represented by a pair $\langle s, t\rangle$ where s and t denote the source and the sink respectively. In particular, chopping of a program *w.r.t.* $\langle s, t\rangle$ identifies a subset of its statements that account for all influences of source s on sink t.

The slicing based on program dependence graph is slightly restrictive in the sense that the dependency graph permits slicing of a program with respect to program point p and a variable v that is defined or used at p, rather than *w.r.t.* arbitrary variable at p. PDG-based backward program slicing is performed by walking the graph backwards from the node of interest in linear time [33]. The walk terminates at either entry node or already visited node. Thus, the slice *w.r.t.* the variable v at PDG node n is a graph containing all vertices of the PDG that can reach directly or indirectly to n and affect the value of v via data-flow or control edges.

In case of PDG-based forward slicing technique, similarly, we traverse the graph in forward direction rather than backward from the node of interest. We can use the standard notion of *chop* of a program with respect to two nodes s and t in slicing technique [23, 35]: $chop(s, t)$ is defined as the set of inter-procedurally valid PDG paths from s to t where s, t are real program nodes, in contrast to ϕ nodes in SSA form of the program. We define it as follows [39]: $AC(s, t)$ is defined to be true if there exists at least one execution ψ that

satisfies a valid PDG path η between s and t *i.e.* $AC(s,t) \triangleq \exists \psi : AC(s,t,\psi)$ and $AC(s,t,\psi) \triangleq \exists \eta \in chop(s,t) : \psi \vdash \eta$. The $\neg AC(s,t)$ implies that $\forall \psi$ and $\forall \eta \in chop(s,t) : \psi \nvdash \eta$, that is, $chop(s,t)$ is empty.

In PDG-based dynamic slicing [1, 17, 32] *w.r.t.* a variable for a given execution history, a projection of the PDG *w.r.t.* the nodes that occur in the execution history is obtained, and then static slicing algorithm on the projected Dependence Graph is applied to find the desired dynamic slice. Agrawal and Horgan [1] also introduce a variant of it based on the graph-reachability framework, called Dynamic Dependence Graph and Reduced Dynamic Dependence Graph, to obtain more precise dynamic slice.

Dependence Condition Graph (DCG). Dependence Condition Graph (DCG) [39] is built from the PDG by annotating each edge $e = e.src \rightarrow e.tgt$ in the PDG with information $e^b = \langle e^R, e^A \rangle$ that captures the conditions under which the dependence represented by that edge is manifest. e^R refers to *Reach Sequences*, whereas e^A refers to *Avoid Sequences*. The informal interpretation of e^R is that the conditions represented by it should be true for an execution to ensure that $e.tgt$ is reached from $e.src$. The *Avoid Sequences* e^A captures the possible conditions under which the assignment at $e.src$ can get over-written before it reaches $e.tgt$. The interpretation of e^A is that the conditions represented by it must not hold in an execution to ensure that the variable being assigned at $e.src$ is used at $e.tgt$. For instance, consider the DDG edge $2 \xrightarrow{x} 4$ in Figure 2, the *Reach Sequence* of it is $(2 \xrightarrow{x} 4)^R = 1 \xrightarrow{true} 3 \xrightarrow{true} 4$ and the *Avoid Sequence* of it is $(2 \xrightarrow{x} 4)^A = \emptyset$, meaning that the condition at node 3 must be *true* to ensure that the definition of x at node 2 can reach node 4. If we consider the DDG edge $2 \xrightarrow{x} \phi$, we see that the *Reach Sequence* of it is $(2 \xrightarrow{x} \phi)^R = \emptyset$ and the *Avoid Sequence* of it is $(2 \xrightarrow{x} \phi)^A = 1 \xrightarrow{true} 3 \xrightarrow{true} 4$, meaning that the condition at node 3 should be $false$ to ensure that the definition of x at node 2 does not get over-written by the definition at node 4, so that x can reach from node 2 to ϕ. It is worthwhile to note that e^A is relevant only for DDG edges and it is $\emptyset$ for CDG edges. Table 1 depicts the DCG annotations for all DDG edges in the PDG of Figure 2. The interested readers may look at [39] to see the detail description of the algorithm to compute DCG from a PDG.

d	d^R	d^A
$2 \xrightarrow{x} 3$	$\emptyset$	$\emptyset$
$2 \xrightarrow{x} 4$	$1 \xrightarrow{true} 3 \xrightarrow{true} 4$	$\emptyset$
$4 \xrightarrow{x} \phi$	$\emptyset$	$\emptyset$
$2 \xrightarrow{x} \phi$	$\emptyset$	$1 \xrightarrow{true} 3 \xrightarrow{true} 4$
$\phi \xrightarrow{x} 5$	$\emptyset$	$\emptyset$

Table 1: DCG annotation for P_{ssa}

2.1 Abstract Interpretation

Abstract Interpretation, originally introduced by Cousot and Cousot is a well known semantics-based static analysis technique [10, 16]. Its main idea is to relate concrete and abstract semantics where the later are focussing only on some properties of interest. Abstract semantics is obtained from the concrete one by substituting concrete domain of computation and their basic concrete semantic operations with the abstract domain and corresponding abstract semantic operations. This can be expressed by means of closure operators.

An (*upper*) *closure* operator on a set C, or simply a closure, is an operator $\rho : C \to C$ which is monotone, idempotent, and extensive. The upper closure operator is the function that maps the concrete values with their abstract properties, namely with the best possible approximation of the concrete value in the abstract domain. For example, the operator $PAR : \wp(Z) \to \wp(Z)$ associates each subset of integers with its parity, $PAR(\bot) = \bot$, $PAR(S) = EVEN = \{n \in Z \mid n \text{ is even}\}$ if $\forall n \in S$. n is even, $PAR(S) = ODD = \{n \in Z \mid n \text{ is odd}\}$ if $\forall n \in S$. n is odd, and $PAR(S) = I\ don't\ know = Z$ otherwise.

Abstract Semantics: Expressions and Statements

As in [30], we consider the IMP language [43]. The statements of a program P act on a set of constants $\mathbb{C} = const(P)$ and a set of variables $VAR = var(P)$. A program variable $x \in VAR$ takes its values from the semantic domain $\mathbb{V} = Z_{\mho}$ where, $\mho$ represents an undefined or uninitialized value and Z is the set of integers. The arithmetic expressions $e \in Aexp$ and boolean expressions $b \in Bexp$ are defined by standard operators on constants and variables. The set of states Σ consists of functions $\sigma : VAR \to \mathbb{V}$ which map the variables to their values. For the program with k variables $x_1, \ldots, x_k$, the state is denoted by k-tuples: $\sigma = \langle v_1, \ldots, v_k \rangle$, where $v_i \in \mathbb{V}, i = 1, ..., k$ and hence, the set of states $\Sigma = (\mathbb{V})^k$. Given a state $\sigma \in \Sigma$, $v \in \mathbb{V}$, and $x \in VAR$: $\sigma[x \leftarrow v]$ denotes a state obtained from σ by replacing its contents in x by v, *i.e.* define

$$\sigma[x \leftarrow v](y) = \begin{cases} v & if\ x = y \\ \sigma(y) & if\ x \neq y \end{cases}$$

The semantics of arithmetic expression $e \in Aexp$ over the state σ is denoted by $E[\![e]\!](\sigma)$ where, the function E is of the type $Aexp \to (\sigma \to \mathbb{V})$. Similarly, $B[\![b]\!](\sigma)$ denotes the semantics of boolean expression $b \in Bexp$ over the state σ of type $Bexp \to (\sigma \to T)$ where $T \in \{true, false\}$.

The Semantics of statement s is defined as a partial function on states and is denoted by $S[\![s]\!](\sigma)$ which defines the effect of executing s in σ.

Consider an abstract domain ρ on values. The set of abstract states is denoted by $\Sigma^\rho \triangleq \rho(\wp(\mathbb{V}))^k$. The abstract semantics $E[\![e]\!]^\rho(\epsilon)$ of expression e is defined as the best correct approximation of $E[\![e]\!]$: let $\sigma = \langle v_1, \ldots, v_k \rangle \in \Sigma$ and $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho : E[\![e]\!]^\rho(\epsilon) = \rho(\{E[\![e]\!](\langle u_1, \ldots, u_k \rangle) \mid \forall i.\ u_i \in \rho(v_i)\})$.

Given an abstract state ϵ, a covering $\{\epsilon_1, \ldots, \epsilon_l\}$ is a set of states such that ϵ describes the same set of concrete states as all the ϵ_i: $\epsilon = \cup_i \epsilon_i$.

Partitions, Atoms

Given $\rho \in uco(\wp(Z_{\mho}))$, the induced partition $\Pi(\rho)$ of ρ is the set $\{V_1, \ldots, , V_j\}$, partition of $\mathbb{V}$, characterizing classes of values undistinguishable by ρ: $\forall i.\ \forall x, y \in V_i.\ \rho(x) = \rho(y)$. A domain ρ is partitioning if it is the most concrete among those inducing the same partition: for a partition P, $\rho = \sqcap\{\mu \mid \Pi(\mu) = P\}$. If ρ is partitioning, $\Pi(\rho)$ is the set of atoms of ρ, viewed as a complete lattice, *i.e.*, atoms of partitioning domain are the abstractions of singletons [30].

3 A Motivating Example

In [30], the traditional syntactic PDG is refined by introducing the notion of semantic data dependency between an expression e and the set of variables $var(e)$ appearing in e at program point p. The expression e does not semantically depend on a variable $x \in var(e)$ if the evaluation of e over any two different states σ_1 and σ_2 at p where $\forall y \in var(e) \wedge y \neq x : \sigma_1(y) = \sigma_2(y)$, results the same values for e. Although the presence of variable x in expression e shows the syntactic data dependency of e on x, semantically may be there is no such dependency. For instance, the expression $e = x+x-2x+4$ does not depend semantically on x. The concrete semantic data dependency can easily be lifted to an abstract domain where the dependency is computed with respect to properties of variables rather than their concrete values. This (abstract) semantic data dependency is used to eliminate irrelevant dependences from traditional PDGs, resulting into more precise (abstract) slices. However, if we observe carefully, we see that the semantic data dependency derived at expression level [30] does not always result in a more precise PDG for the program.

In the rest of the paper, we use static single assignment (SSA) form of programs due to its compact representation and easy to compute DCG annotations as well as to define its semantics. The SSA form also helps in improving the flow-sensitive analysis of any program [20, 12].

Example 1 *Consider the program P and the corresponding traditional Program Dependence Graph (G_{pdg}) for its SSA correspondent code, as depicted in Figure 3. Observe that the condition represented by node 1 is implicitly true which means that during an execution once node 1 is executed, then its target 2, 3, 4, 5, 6, 8, ϕ_1 and ϕ_2 will eventually be executed, whereas a control dependence edge labeled by 'lab' (where 'lab' $\in \{true, false\}$) represents lab-control dependence, for instance, the edge $8 \xrightarrow{false} 11$ represents that 11 is false-control dependent on 8. The data dependency between two nodes n_1 and n_2 for a variable x is represented by the edge $n_1 \xrightarrow{x} n_2$.*

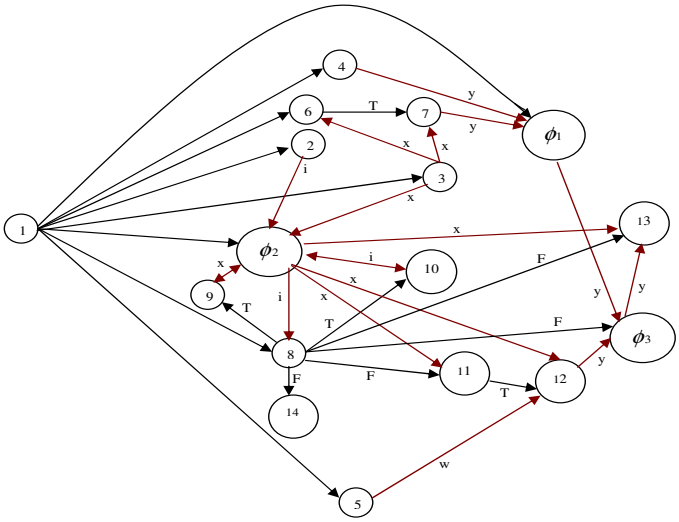
Suppose we are interested only in the sign of the program variables, and we consider the abstract domain SIGN $= \{\bot, +, 0, -, 0^+, 0^-, \top\}$, where 0^+ represents $\{x \in Z :\ x \geq 0\}$, 0^- represents $\{x \in Z :\ x \leq 0\}$, and Z is the set of integers. After computing the

```
1.   start
2.   i = −2;
3.   x = input;
4.   y = input;
5.   w = input;
6.   if(x ≥ 0)
7.         y = 4x^3;
8.   while(i ≤ 0){
9.         x = x × 2;
10.        i = i + 1;}
11.  if(x ≤ 0)
12.        y = x^2 + 4w mod 2;
13.  print(x, y);
14.  stop
```

(a) *Program P*

```
1.   start
2.   i1 = −2;
3.   x1 = input;
4.   y1 = input;
5.   w = input;
6.   if(x1 ≥ 0)
7.         y2 = 4 × (x1)^3;
φ1   y3 = f(y1, y2);
     while(
φ2   (x2, i2) = f((x1, i1), (x3, i3));
8.   i2 ≤ 0
     ){
9.         x3 = x2 × 2;
10.        i3 = i2 + 1;}
11.  if(x2 ≤ 0)
12.        y4 = (x2)^2 + 4w mod 2;
φ3   y5 = f(y3, y4);
13.  print(x2, y5);
14.  stop
```

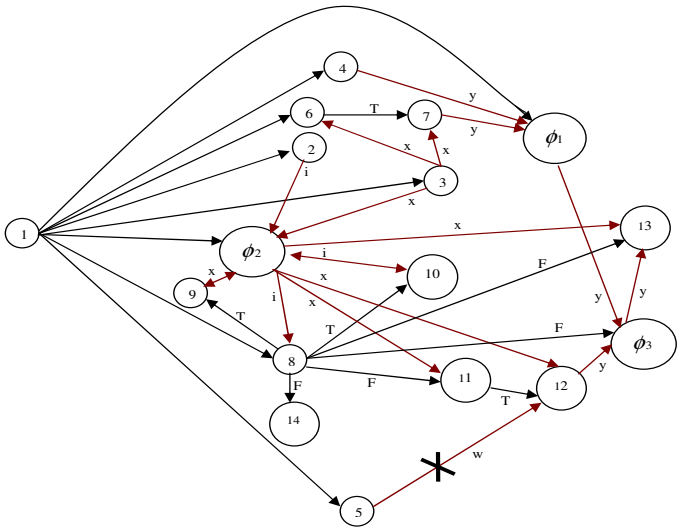
(b) P_{ssa}: *SSA form of program P*



(c) G_{pdg}: *PDG of* P_{ssa}

Figure 3: The traditional Program Dependence Graph (PDG)

semantic data dependency [30] w.r.t. SIGN, we get the semantics-based abstract PDG shown in Figure 4(a). Observe that the semantic data dependency w.r.t. SIGN removes the data dependency between y_4 *and* w *at statement 12, as "*$4w \bmod 2$*" always yields to* 0*. Therefore, the corresponding data dependence edge* $5 \xrightarrow{w} 12$ *is disregarded from the traditional PDG. The slicing algorithm based on this semantics-based abstract PDG with respect to criteria* $\langle 13, y, SIGN\rangle$ *returns the program depicted in Figure 4(b). At program point* 9*, the variable* x_2 *may have any abstract value in* $\{+, 0, -\}$*. Since the evaluation of the expression* $x_2 \times 2$ *over all these possible abstract values yields to the dependency of the expression* $x_2 \times 2$ *on* x_2 *w.r.t. SIGN, the dependency is included in the semantics-based abstract PDG by the edge linking node* 9 *to node* ϕ_2*. However, we observe that the execution of statement* 9 *does not affect at all the sign of* x*. This "false positive" is due to the fact that the semantic data dependency in [30] is defined*



(a) G^d_{pdg} : *PDG of* P_{ssa} *after computing Semantic Data Dependency w.r.t. SIGN* [30]

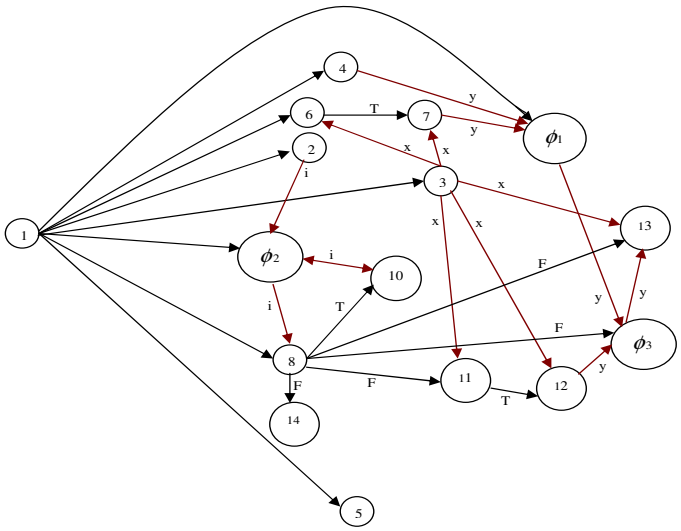
```
1.    start
2.    i = −2;
3.    x = input;
4.    y = input;
6.    if(x ≥ 0)
7.          y = 4x^3;
8.    while(i ≤ 0){
9.          x = x × 2;
10.         i = i + 1;}
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

(b) *Slice w.r.t. criteria* $\langle 13, y, SIGN\rangle$ *computed from* G^d_{pdg}

Figure 4: Semantics-based abstract PDG and corresponding slice after computing Semantic Data Dependency w.r.t. SIGN

at expression level. The abstract semantics of the program should say that the entire statement 9 is not relevant w.r.t. SIGN for the slicing criteria $\langle 13, y, SIGN\rangle$*. Thus, slicing with respect to criteria* $\langle 13, y, SIGN\rangle$ *should have the correct and more precise slice shown in Figure 5(b), as the sign of x at line 11 and 12 in the original program is the same as that in the input value at line 3.*

The point we raise is that the semantics-based PDG obtained from the semantic data dependency [30] can be improved w.r.t. accuracy if, before deriving the data dependency at expression level, we compute the semantic relevancy of statements in the program w.r.t. the property of interest. This way, we can refine the PDG, by eliminating the nodes corresponding to the semantically irrelevant statements from the PDG. In our example, Figure 5(a) depicts a more precise semantics-based abstract PDG $G^{r,d}_{pdg}$ *obtained by computing first the semantic relevancy of the statements w.r.t. SIGN that removes the node corresponding to the irrelevant statement at program point 9, and then by computing the semantic data dependency w.r.t. SIGN [30] that removes the*



(a) $G^{r,d}_{pdg}$: *PDG of* P_{ssa} *by computing Statement Relevancy first, and then Semantic Data Dependency w.r.t. SIGN* − *Observe that node* 9 *does not appear anymore.*

```
1.    start
2.    i = −2;
3.    x = input;
4.    y = input;
6.    if(x ≥ 0)
7.          y = 4x^3;
8.    while(i ≤ 0){
10.         i = i + 1;}
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

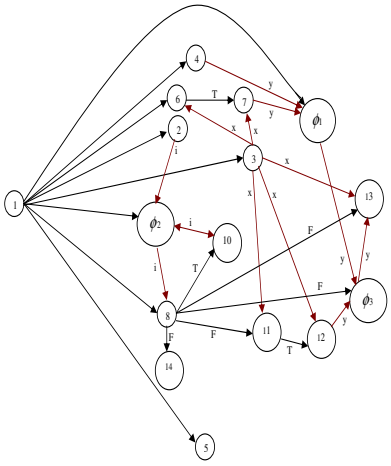
(b) *Slice w.r.t. criteria* $\langle 13, y, SIGN\rangle$ *obtained from* $G^{r,d}_{pdg}$

Figure 5: Semantics-based abstract PDG and corresponding slice by computing Statement Relevancy first, and then Semantic Data Dependency w.r.t. SIGN

data dependency $5 \xrightarrow{w} 12$.

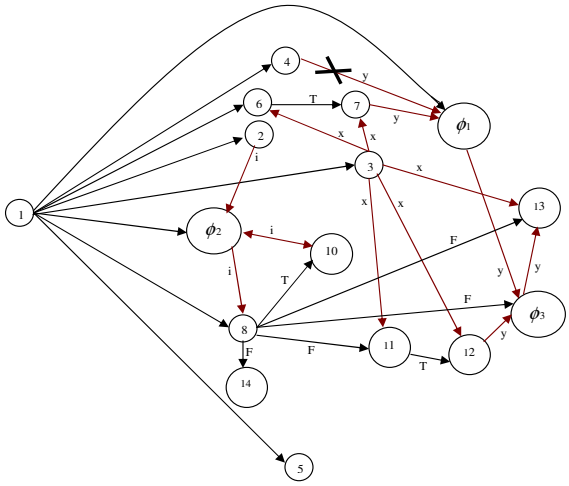
We can even be more accurate in the slicing procedure. By following [39], we can extend every PDG into the Dependence Condition Graph (DCG) by introducing the annotation $e^b \triangleq \langle e^R, e^A\rangle$ *over all the data/control dependence edges* $e = e.src \rightarrow e.tgt$*:* e^R *is referred to as Reach Sequences and represents the conditions required to reach the data from e.src to e.tgt, and* e^A *is referred to as Avoid Sequences and used to avoid the re-definition of the data coming from e.src (in case of control dependence edge,* e^A *is* $\emptyset$*). An execution trace* ψ *is said to satisfy* e^b *for a data dependence edge e if it satisfies all the conditions in* e^R*, and at the same time, it avoids the conditions in* e^A*. This means that the execution trace* ψ *ensures that the value computed at e.src successfully reaches e.tgt. We can easily extend this to any arbitrary dependence path* $\eta = e_1 e_2 \ldots e_n$ *in DCG.*

Let us illustrate the computation of DCG annotations over the edges as reported



e	e^R	e^A
$4 \xrightarrow{y} \phi_1$	$\emptyset$	$1 \xrightarrow{true} 6 \xrightarrow{true} 7$
$7 \xrightarrow{y} \phi_1$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 6$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 7$	$1 \xrightarrow{true} 6 \xrightarrow{true} 7$	$\emptyset$
$3 \xrightarrow{x} 11$	$1 \xrightarrow{true} 8 \xrightarrow{false} 11$	$\emptyset$
$3 \xrightarrow{x} 12$	$1 \xrightarrow{true} 8 \xrightarrow{false} 11 \xrightarrow{true} 12$	$\emptyset$
$3 \xrightarrow{x} 13$	$1 \xrightarrow{true} 8 \xrightarrow{false} 13$	$\emptyset$
$12 \xrightarrow{y} \phi_3$	$\emptyset$	$\emptyset$
$\phi_1 \xrightarrow{y} \phi_3$	$1 \xrightarrow{true} 8 \xrightarrow{false} \phi_3$	$1 \xrightarrow{true} 8 \xrightarrow{false} 11 \xrightarrow{true} 12$
$\phi_3 \xrightarrow{y} 13$	$\emptyset$	$\emptyset$
$2 \xrightarrow{i} \phi_2$	$\emptyset$	$\emptyset$
$\phi_2 \xrightarrow{i} 8$	$\emptyset$	$\emptyset$
$10 \xrightarrow{i} \phi_2$	$\emptyset$	$\emptyset$
$\phi_2 \xrightarrow{i} 10$	$1 \xrightarrow{true} 8 \xrightarrow{true} 10$	$\emptyset$

(a) G_{dcg} : *DCG after computing annotations on* $G^{r,d}_{pdg}$



(b) G^{s}_{dcg} : *semantics-based abstract DCG after removing* $e = 4 \xrightarrow{y} \phi_1$ *from* G_{dcg}

Figure 6: Semantics-based abstract DCG

in [39] on our example: Given the semantics-based abstract PDG $G^{r,d}_{pdg}$ of Figure 5(a), consider the DDG edge $e = \phi_1 \xrightarrow{y} \phi_3$. For this DDG edge, we see that node ϕ_3 does not post-dominate the node ϕ_1. Thus, the reach sequence for the edge is $e^R = (\phi_1 \xrightarrow{y} \phi_3)^R = \{1 \xrightarrow{true} 8 \xrightarrow{false} \phi_3\}$. This means that the condition at 8 must be false in the execution trace to ensure that once ϕ_1 is executed ϕ_3 is also executed, and the data y assigned at ϕ_1 can reach ϕ_3. To compute the avoid sequences for the edge $\phi_1 \xrightarrow{y} \phi_3$, we consider two data dependence edges $e_1 = \phi_1 \xrightarrow{y} \phi_3$ and $e_2 = 12 \xrightarrow{y} \phi_3$ with ϕ_3 as target. By following the algorithm of [39], we get the avoid sequences of e_1 as $e_1^A = (\phi_1 \xrightarrow{y} \phi_3)^A = \{1 \xrightarrow{true} 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$. This reflects the fact that the "if" condition at 11 must be false in order to guarantee that the definition of y at ϕ_1 is not re-defined at 12 and can reach ϕ_3. The table in the Figure 6(a) depicts the DCG annotations over the data dependence edges of $G^{r,d}_{pdg}$, yielding to a DCG G_{dcg}.

Let us consider the dependence path $\eta_\phi = 4 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_3 \xrightarrow{y} 13$, which is a ϕ-sequence [39] representing the flow of definition at 4 to 13 in the DCG G_{dcg}. Observe that, since the abstract values of x may have any value from the set $\{+, 0, -\}$, there is no such execution trace ψ over the abstract domain SIGN that can avoid both $(4 \xrightarrow{y} \phi_1)^A = \{1 \xrightarrow{true} 6 \xrightarrow{true} 7\}$ and $(\phi_1 \xrightarrow{y} \phi_3)^A = \{1 \xrightarrow{true} 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$ simultaneously, i.e., $\forall \psi : \psi \not\kappa^{SIGN} \eta_\phi$. For each execution trace over the abstract domain of sign, at least one of the conditions among $6 \xrightarrow{true} 7$ and $11 \xrightarrow{true} 12$ must be satisfied. This means that the definition at 4 is over-written either by 7 or by 12 or by both, and can never reach 13. Since there exists no semantically realizable ϕ-sequence from the node 4 to any target node t such that y defined at 4 can reach t, we can remove the edge $4 \xrightarrow{y} \phi_1$ from G_{dcg}, resulting into a more precise semantics-based abstract DCG G^s_{dcg} as depicted in Figure 6(b). Thus, if we apply the backward slicing technique [33] on G^s_{dcg} w.r.t. the slicing criteria $\langle 13, y, SIGN \rangle$, we get a sub-DCG G_{sdcg} and a more precise slice as shown in Figure 7.

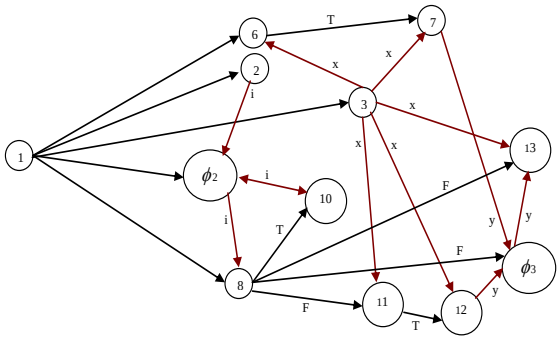
4 Semantic Relevancy

In this section, we define the semantic relevancy of statements *w.r.t.* a property of interest in both the concrete and the abstract domains. We also discuss how to treat the relevancy of the control statements based on the relevancy at block level.

4.1 Semantic Relevancy of Statements

The semantic relevancy of a statement *w.r.t.* a concrete/abstract property determines whether the execution of the statement affects the conceret/abstract property of the variables of interest.

Definition 1 *(Concrete Semantic Relevancy)*
Given a program P and a concrete property ω on states, the statement s at program



(a) G_{sdcg}: sub-DCG after performing backward slicing on G^{s}_{dcg} *w.r.t.* $\langle 13, y\rangle$

```
1.    start
2.    i = −2;
3.    x = input;
6.    if(x ≥ 0)
7.          y = 4x^3;
8.    while(i ≤ 0){
10.         i = i + 1;}
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

(b) *Slice w.r.t.*$\langle 13, y\rangle$ *computed from* G_{sdcg}

Figure 7: Slicing *w.r.t.* $\langle 13, y, SIGN\rangle$

point p in P is semantically relevant w.r.t. ω if:

$$\exists \sigma \in \Sigma_p : \; S[\![s]\!](\sigma) = \sigma' \; \wedge \; \omega(\sigma) \neq \omega(\sigma')$$

where Σ_p are the set of states that can possibly reach the program point p.

In other words, the statement s at program point p is semantically irrelevant *w.r.t.* a concrete property ω if the execution of s over any state $\sigma \in \Sigma_p$ yields to a state that is equivalent to σ *w.r.t.* ω.

In particular, whenever ω distributes over program variables, we may use the Definition 2.

Definition 2 *(Concrete Semantic Relevancy for a Set of Variables)*
Given a program P and a concrete property ω that distributes over program variables, the statement s at program point p in P is semantically relevant w.r.t. ω for a subset of variables U if

$$\exists \sigma = \langle \sigma(v_1), \sigma(v_2), \ldots, \sigma(v_k)\rangle \in \Sigma_p \text{ and } \exists v_i \in U \text{ such that } \omega(\pi_i(S[\![s]\!]\sigma)) \neq \omega(\pi_i(\sigma))$$

where $\pi_i(\langle \sigma(v_1), \sigma(v_2), \ldots, \sigma(v_k)\rangle) = \sigma(v_i)$.

Example 2 *Consider the concrete property $\omega_{(x\neq 4)} : \Sigma \to \{true, false\}$, which is true in state σ iff $\sigma(x) \neq 4$. The statement $x = y + 1$ is semantically relevant w.r.t. $\omega_{(x\neq 4)}$ only if $\exists \sigma \in \Sigma_p$ such that $\sigma(y) = 3$ and $\sigma(x) \neq 4$.*

Example 3 *Consider the program P_{ssa} in Figure 8. The statement at program point 7 is semantically irrelevant w.r.t. all concrete properties, as the execution of this statement over any state possibly reaching program point 7 does not change the state. That is, $\forall\sigma \in \Sigma_7$ where $\sigma(x) = \sigma(y)$ and $\sigma(z) = 4$, the execution of the statement over σ does not modify the value of z.*

```
1.    start
2.    x1 = input;
3.    y1 = input;
4.    w = input;
5.    z1 = 4;
6.    if(x1 == y1){
7.          z2 = 2 × (x1 + y1) − 4 × (x1) + 4;
8.          x2 = x1 + 1;}
9.    else { z3 = x1 + w;
10.         x3 = x1 + 2;}
φ     (x4, z4) = f((x2, z2), (x3, z3));
11.   print(x4, z4);
12.   stop
```

Figure 8: Program P_{ssa}

Example 4 *Consider the program P_{ssa} in Figure 8 and the property defined by $\omega \triangleq \#\{x \in VAR : [\![x]\!]\sigma \in EVEN\} = \#\{x \in VAR : [\![x]\!]\sigma \in ODD\}$ where VAR is the set of program variables, $\sigma \in \Sigma$, EVEN represents $\{y \in Z : y \text{ is even}\}$, ODD represents $\{y \in Z : y \text{ is odd}\}$, Z is the set of integers, and # denotes the cardinality of set. The statement $s \triangleq x = x + 1$ at program point 8 is relevant w.r.t. ω, since the execution of s over any state σ at 8 with equal number of values belonging to ODD and EVEN sets, yields to a state σ' where the value of x move from one set to another. Observe that the statement at program point 10, on the other hand, is irrelevant w.r.t. ω.*

In order to compute semantic relevancy of a statement at program point p *w.r.t.* a concrete property, we need to compare the concrete property of each state possibly reaching p with that of the corresponding state resulting after the execution of the statement. If the execution changes the property for at least one state, we say that the statement is relevant *w.r.t.* that concrete property.

Now we discuss how to obtain all possible reaching states at each program points of a program using its collecting semantics.

Collecting Semantics. Let Σ and *Label* be the set of states and the set of program points. The context vector is defined by $\mathit{Context\text{-}Vector} : \mathit{Label} \rightarrow \mathit{Context}$, where $\mathit{Context} = \wp(\Sigma)$.

Let P be a program of size n. The context at program point i is denoted by Cx_i. The context vector associated with program P is, thus, denoted by $Cv_P = \langle Cx_1, Cx_2, \ldots, Cx_n \rangle$, where $Cv_P(i) = Cx_i$.

Let

$$F_i : \mathit{Context\text{-}Vector} \rightarrow \mathit{Context}$$

be a collection of monotone functions, where

$$\begin{aligned} Cx_1 &= F_1(Cx_1, \ldots, Cx_n) \\ Cx_2 &= F_2(Cx_1, \ldots, Cx_n) \\ &\ldots\ldots \\ Cx_n &= F_n(Cx_1, \ldots, Cx_n) \end{aligned}$$

Combining the above functions, we get

$$F: \ \textit{Context-Vector} \rightarrow \textit{Context-Vector}$$

That is,

$$F(Cx_1, \ldots, Cx_n) = (F_1(Cx_1, \ldots, Cx_n), \ldots, F_n(Cx_1, \ldots, Cx_n))$$

The function F_i includes the transition function defined as follows:

$$Cx_i = \bigcup_{s_j \in pred(s_i)} \cup_{\sigma_j \in Cx_j} \ S[\![s_j]\!](\sigma_j)$$

where, $pred(s_i)$ is the set of predecessors of the statement s_i. Now consider the following:

- s_j is assignment statement ($s_j \triangleq x = e$):

$$S[\![x = e]\!]\sigma_j = \sigma_j[x = v] \text{ if } E[\![e]\!]\sigma_j = v$$

- s_j is conditional/while statement containing *cond* as the condition in it, and s_i is *true*-successor:

$$S[\![s_j]\!]\sigma_j = \sigma_j \text{ if } B[\![cond]\!]\sigma_j = true$$

- s_j is conditional/while statement containing *cond* as the condition in it, and s_i is $false$-successor:

$$S[\![s_j]\!]\sigma_j = \sigma_j \text{ if } B[\![cond]\!]\sigma_j = false$$

We can compute the least fix-point of F in order to obtain the collecting semantics for P. We start with the initial *Context-Vector*=$\langle \bot, \ldots, \bot \rangle$ which is the bottom element of the lattice L^n, where $L = (\wp(\Sigma), \subseteq, \cap, \cup, \top, \bot)$. With this collecting semantics, we can easily obtain the states possibly reaching each program points in a program that help in computing semantic relevancy of all statements *w.r.t.* any concrete property of interest. The following example shows how to compute collecting semantics of a program using fix-point computation.

```
1.  start
2.  n = input;
3.  x = 1;
4.  while(n > 0){
5.          x = x × n;
6.          n = n − 1; }
7.  stop
```

Figure 9: Program P

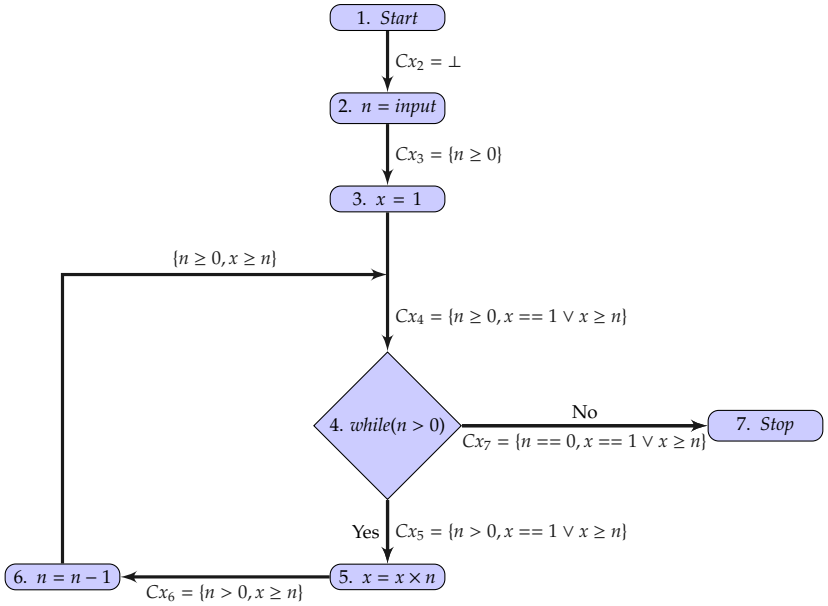


Figure 10: Control Flow Graph of P

Example 5 *Let us consider the following program P as depicted in Figure 9. The control-flow graph of the program P is shown in Figure 10. Observe that initially the contexts associated with each program point is* $\perp$*, i.e.,* $\forall i \in [1..7]$*:* $Cx_i = \perp$*. Thus, the initial context vector is represented by* $Cv_P = \langle Cx_1, \ldots, Cx_7 \rangle = \langle \perp, \ldots, \perp \rangle$*.*

We assume that the initial value of n is always greater than or equal to 0. The monotone function $F = (F_1, F_2, \ldots, F_7)$ *for the program P is defined as follows:*

$$
\begin{aligned}
Cx_1 &= F_1(Cx_1 \ldots, Cx_7) = \perp \\
Cx_2 &= F_2(Cx_1 \ldots, Cx_7) = \perp \\
Cx_3 &= F_3(Cx_1 \ldots, Cx_7) = \{\sigma \mid \sigma(n) \geq 0\} \\
Cx_4 &= F_4(Cx_1 \ldots, Cx_7) = \{\sigma[x = 1] \mid \sigma \in C_3\} \cup \{\sigma[n = \sigma(n) - 1] \mid \sigma \in C_6\} \\
Cx_5 &= F_5(Cx_1 \ldots, Cx_7) = C_4 \cap \{\sigma \mid \sigma(n) > 0\} \\
Cx_6 &= F_6(Cx_1 \ldots, Cx_7) = \{\sigma[x = \sigma(x) \times \sigma(n)] \mid \sigma \in C_5\} \\
Cx_7 &= F_7(Cx_1 \ldots, Cx_7) = C_4 \cap \{\sigma \mid \sigma(n) == 0\}
\end{aligned}
$$

Starting from the initial context vector $Cv_P = \langle Cx_1, \ldots, Cx_7 \rangle = \langle \bot, \bot, \ldots, \bot \rangle$*, if we apply F repeatedly, we get the following least solution, as depicted in Figure 10:*

$$\begin{aligned}
Cx_1 &= \bot \\
Cx_2 &= \bot \\
Cx_3 &= \{n \geq 0\} \\
Cx_4 &= \{n \geq 0, x == 1 \vee x \geq n\} \\
Cx_5 &= \{n > 0, x == 1 \vee x \geq n\} \\
Cx_6 &= \{n > 0, x \geq n\} \\
Cx_7 &= \{n == 0, x == 1 \vee x \geq n\}
\end{aligned}$$

We now lift to an abstract setting, and we define the relevancy of statements *w.r.t.* an abstract property of interest.

Definition 3 *(Abstract Semantic Relevancy)*
Given a program P and $\rho \in UCO(\wp(\mathbb{V}))$*, the statement s at program point p in P is semantically relevant w.r.t. abstract property* ρ *if*

$$\exists \epsilon \in \Sigma_p^\rho : \; S[\![s]\!]^\rho(\epsilon) \neq \epsilon$$

where Σ_p^ρ *are the set of abstract states that can possibly reach the program point p, and all the abstract states* $\epsilon \in \Sigma_p^\rho$ *take atomic values from the induced partition* $\Pi(\rho)$*.*

In other words, the statement s at program point p is semantically irrelevant *w.r.t.* the abstract property ρ if no changes take place in the abstract states ϵ possibly reaching program point p, when s is executed over ϵ.

Observe that Definition 3 is not parametric on variables. Below we provide a parametric definition for the abstract statements relevancy:

Definition 4 *(Abstract Semantic Relevancy for a Set of Variables)*
Given a program P and $\rho \in UCO(\wp(\mathbb{V}))$*, the statement s at program point p in P is semantically relevant w.r.t. abstract property* ρ *for a subset of variables U if*

$$\exists \epsilon = \langle \rho(v_1), \rho(v_2), \ldots, \rho(v_k) \rangle \in \Sigma_p^\rho \text{ and } \exists v_i \in U \text{ such that } \pi_i(S[\![s]\!]^\rho(\epsilon)) \neq \pi_i(\epsilon)$$

where $\pi_i(\langle \rho(v_1), \rho(v_2), \ldots, \rho(v_k) \rangle) = \rho(v_i)$*.*

This definition may be useful, combined with independence analysis, to further refine the slicing when focussing just on a proper subset of program variables in the slicing criteria.

Intuitively, the semantic relevancy of statements just says that an assertion remains true over the states possibly reaching the corresponding program points. Observe that if we consider the predicate ω as an abstraction on the concrete states, Definition 3 is just a rephrasing of Definition 1.

Example 6 *Consider the statement* $s \triangleq x = x + 2$ *at program point* 10 *of the program* P_{ssa} *depicted in Figure 8. The statement s is semantically relevant w.r.t.* $\rho = SIGN$*,*

because $\exists \epsilon = \langle \rho(x), \rho(y), \rho(w), \rho(z) \rangle = \langle -, +, +, + \rangle \in \Sigma_{10}^{SIGN}$*:* $S[\![s]\!]^{\rho}(\langle -, +, +, + \rangle) = \langle \top, +, +, + \rangle$*. On the other hand, if we consider the abstract domain* $\rho = PAR$*, we see that s is semantically irrelevant w.r.t. PAR because* $\forall \epsilon \in \Sigma_{10}^{PAR}$*:* $S[\![s]\!]^{\rho}(\epsilon)$ *does not change the parity of x.*

It is worthwhile to mention that only the "*assignment*" statements are able to change the property of the variables which are defined by those statements. So, relevancy checking for the statements except "*assignment*" are meaningless, as they are not able to change the property for any variable.

We can easily compute abstract collecting semantics of the program by computing the least fix point of abstract monotone function $F^{\sharp}$ (corresponds to the concrete monotone function F) in the abstract domain ρ, where we consider abstract contexts denoted by $Context^{\sharp}=\wp(\Sigma^{\rho})$ instead of the concrete one. This way, we can obtain all possible abstract states that can possibly reach each program points in the program. It is worthwhile to mention that during the computation of abstract contexts $Cx_i^{\sharp}$ for statement s_i using the abstract monotone function $F_i^{\sharp}$, we consider ϵ_j if it is atomic, otherwise we consider the covering $\{\epsilon_1, \ldots, \epsilon_k\}$ of ϵ_j where each ϵ_i is atomic, to compute $S[\![s_j]\!]^{\rho}(\epsilon_j)$ for all $\epsilon_j \in Cx_j^{\sharp}$ for all predecessors s_j.

The notion of statements relevancy have many interesting application areas. For instance, if we are analyzing a speed control engine and we are just interested on the portion of program that may lead to a totally unexpected negative value of a speed variable (yielding to a crash-prone situation), then every statement that does not affect neither directly nor indirectly its sign can immediately be disregarded.

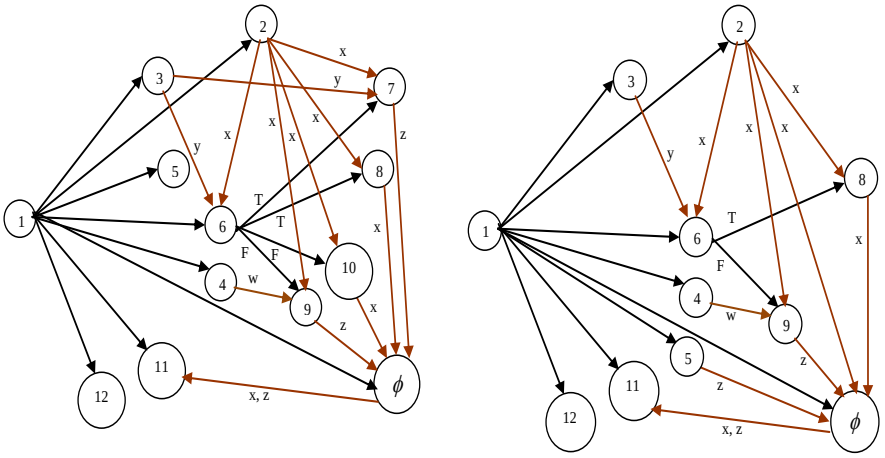
Given a program and its syntactic PDG, we can obtain a more precise semantics-based (abstract) PDG for the program by disregarding from the syntactic PDG all the nodes that corresponds to the irrelevant statements *w.r.t.* the concrete/abstract property. Figure 11(a) depicts the syntactic PDG G_{pdg} of the program P_{ssa} shown in Figure 8, whereas Figure 11(b) depicts the semantics-based abstract PDG G^r_{pdg} which is obtained by disregarding from G_{pdg} two nodes corresponding to the statements 7 and 10 which are irrelevant *w.r.t.* PAR.

4.2 Semantic Relevancy of Blocks

In the previous section, we defined the semantic relevancy of statements *w.r.t.* concrete/abstract property. Now we define semantic relevancy of blocks *w.r.t.* concrete/abstract property, where by block we mean a set of statements $S = \{s_1, \ldots, s_n\}$. Let us denote a block of a set of statements S by blk_S.

Definition 5 *blk_S is semantically irrelevant w.r.t. a concrete property ω (or abstract property ρ) if*

$$\forall s_i \in blk_S,\ i \in [1..n] :\ s_i \textit{ is semantically irrelevant w.r.t. } \omega \textit{ (or } \rho)$$



(a) G_{pdg} : *Syntactic – based PDG of* P_{ssa} (b) G'_{pdg} : *Semantics – based abstract PDG of* P_{ssa} *w.r.t. PAR*

Figure 11: Syntactic and Semantics-based abstract PDG of P_{ssa} of Figure 8

Definition 6 *blk_S is partially relevant w.r.t. a concrete property ω (or abstract property ρ) if*

$$\exists s_i, s_j \in blk_S \ \wedge i,\ j \in [1..n] \wedge i \neq j : s_i \textit{ is semantically relevant w.r.t. } \omega \textit{ (or } \rho\textit{) and } s_j \textit{ is semantically irrelevant w.r.t. } \omega \textit{ (or } \rho\textit{)}$$

Definition 7 *blk_S is completely relevant w.r.t. a concrete property ω (or abstract property ρ) if*

$$\forall s_i \in blk_S,\ i \in [1..n] :\ s_i \textit{ is semantically relevant w.r.t. } \omega \textit{ (or } \rho\textit{)}$$

Observe that we can convert any partially relevant block *w.r.t.* ω (or ρ) into a corresponding completely relevant block by removing all the irrelevant statements *w.r.t.* ω (or ρ) present in that block.

4.3 Treating Relevancy of Control Statements

In this section, we consider the conditional statements "if", "$if-else$" and the repetitive statement "*while*", and their relevancy *w.r.t.* a concrete property ω (or abstract property ρ).

The "if" statement can be expressed as

$$if(cond)\ then\ blk_{if}$$

The "$if - else$" statement can be expressed as

$$if(cond)\ then\ blk_{if}\ else\ blk_{else}$$

Similarly, the repetitive statement "*while*" can be expressed as

$$while(cond)\ blk_{while}$$

The semantic relevancy of "if", "$if - else$" and "*while*" statements solely depend on the relevancy of their corresponding blocks blk_{if}, blk_{else} and blk_{while} respectively.

For a "if" statement "$if(cond)\ then\ blk_{if}$", if the corresponding block blk_{if} is irrelevant *w.r.t.* a concrete property ω (or abstract property ρ), we say that the "if" statement is irrelevant *w.r.t.* ω (or ρ).

For a "*while*" statement "$while(cond)\ blk_{while}$", if the corresponding block blk_{while} is irrelevant *w.r.t.* ω (or ρ), we say that the relevancy of the "*while*" statement is equivalent to the relevancy of the statement "$while(cond)\ then\ skip$" *w.r.t.* ω (or ρ).

The relevancy of "$if-else$" statement, *i.e.*, "$if(cond)\ then\ blk_{if}\ else\ blk_{else}$" *w.r.t.* a concrete property ω (or abstract property ρ) is defined as follows:

1. An "$if - else$" statement "$if(cond)\ then\ blk_{if}\ else\ blk_{else}$" is semantically irrelevant *w.r.t.* ω (or ρ) if both blk_{if} and blk_{else} are semantically irrelevant *w.r.t.* ω (or ρ).

2. If any or both of the blocks blk_{if} and blk_{else} in "$if - else$" statement are partially relevant *w.r.t.* ω (or ρ), we say that the "$if - else$" statement is partially relevant *w.r.t.* ω (or ρ). The partial relevancy of the "$if - else$" statement can be converted into complete relevancy by converting the corresponding partial relevant blocks into completely relevant blocks.

3. If blk_{else} in "$if-else$" statement is semantically irrelevant *w.r.t.* ω (or ρ), then the relevancy of the "$if - else$" statement is equivalent to the relevancy of the statement "$if(cond)\ then\ blk_{if}$" *w.r.t.* ω (or ρ).

4. If blk_{if} in "$if-else$" statement is semantically irrelevant *w.r.t.* ω (or ρ), then the relevancy of the "$if - else$" statement is equivalent to the relevancy of the statement "$if(cond)\ then\ skip\ else\ blk_{else}$" *w.r.t.* ω (or ρ).

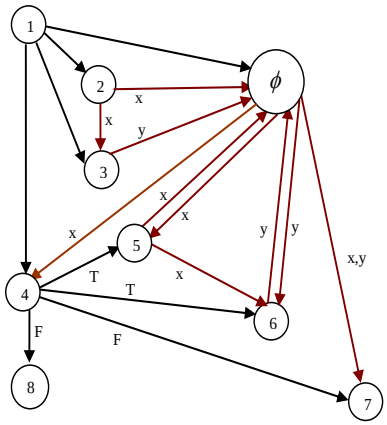
The semantic irrelevancy of a repetitive statement "*while*" *w.r.t. PAR* and a "$if - else$" statement *w.r.t. SIGN* are illustrated in Example 7 and Example 8 respectively.

Example 7 *Consider the program of Figure 12(a) and the property PAR. The traditional syntactic PDG of it, is shown in Figure 12(b). Consider the repetitive statement "while" and the corresponding block blk_{while} that contains two statements at program points 5 and 6. At program point 2 and 3, the parity of x and y are odd and even respectively. These properties of x and y remain unchanged during the execution of the while loop. Statements 5 and 6, therefore, are not semantically relevant w.r.t. PAR. In*

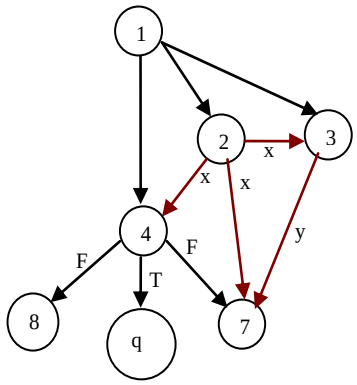
fact, at 5 the parity of x_2 is not affected as its value is not changed by the assignment, and the parity of x_3 is preserved by adding an even value like the constant 2. Similar in case of statement at 6. Since, the block blk_{while} is semantically irrelevant w.r.t. PAR, we can replace it by a "skip" statement (denoted by node "q" in the graph), yielding a more precise PDG shown in Figure 12(c).

1.	start
2.	$x_1 = 1;$
3.	$y_1 = 2 \times x_1;$
	while(
ϕ	$(x_2, y_2) = f((x_1, y_1), (x_3, y_3))$
4.	$x_2 < 20$){
5.	$x_3 = x_2 + 2;$
6.	$y_3 = y_2 + 2 \times x_3^2;$}
7.	*print*(x_2, y_2);
8.	stop

(a) Program P_{ssa}



(b) *Traditional PDG of the Program* P_{ssa}



(c) *Semantics – based abstract PDG of* P_{ssa} *after computing Semantic Relevancy w.r.t. PAR*

Figure 12: Treating "*while*" block

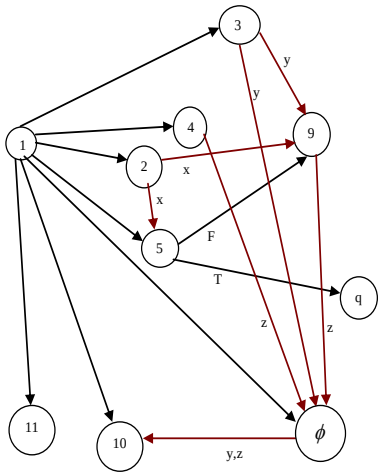
Example 8 *Consider the program in Figure 13(a). Observe that the statements 6 and 7 in the "if" block* blk_{if} *and the statement 8 in the "else" block* blk_{else} *are semantically irrelevant w.r.t. the sign property of the variables. The execution of the statements 6, 7, and 8 over all possible abstract states reaching these program points does not change the sign of y and z. Thus, the "if" block* blk_{if} *is completely irrelevant, whereas the "else" block* blk_{else} *is partially relevant as it contains one relevant statement 9 w.r.t. the sign property. According to the rules discussed above, we can't remove* blk_{if}*. Hence, we replace* blk_{if} *with the statement "skip". The corresponding semantics-based form of the program and semantics-based abstract PDG w.r.t. the sign property are shown in Figure 13(b) and 13(c) respectively.*

1. start
2. $x_1 = input$;
3. $y_1 = 5$;
4. $z_1 = 2$;
5. $if(x_1 \geq 0)\{$
6. $\quad y_2 = y_1 + 2$;
7. $\quad z_2 = y_2 + x_1;\}$
   $else\{$
8. $\quad y_3 = y_1 + 5$;
9. $\quad z_3 = y_3 + x_1\}$;
ϕ. $(y_4, z_4) = f((y_2, z_2), (y_3, z_3))$;
10. $print(y_4, z_4)$;
11. stop

(a) *Program* P_{ssa}

1. start
2. $x_1 = input$;
3. $y_1 = 5$;
4. $z_1 = 2$;
5. $if(x_1 \geq 0)$
q. $\quad skip$;
   $else\{$
9. $\quad z_3 = y_1 + x_1\}$;
ϕ. $(y_4, z_4) = f((y_1, z_1), (y_1, z_3))$;
10. $print(y_4, z_4)$;
11. stop

(b) *Program* P_{ssa} *after computing semantic relevancy w.r.t.SIGN*



(c) *Semantics−based abstract PDG of* P_{ssa} *after computing Semantic Relevancy w.r.t. SIGN*

Figure 13: Treatment of "if-else" block

5 Atomicity of the Abstract States in computing Statements Relevancy

While computing the semantic relevancy of a statement over an abstract state ϵ *w.r.t.* the property ρ, the atomicity of the abstract value for each variable in ϵ is one of the crucial requirements. These atomic abstract values are obtained from induced partitioning [30]. Example 9 shows how to compute the semantic relevancy of statements by using covering techniques satisfying the atomicity requirement.

Example 9 *Consider the abstract domain of parity PAR and an abstract state* $\epsilon = \langle even, odd, \top \rangle$ *for the variables* $x, y, z \in dom(\epsilon)$*. The induced partitions for the domain PAR is* $\Pi(PAR) = \{even, odd\}$*. Since* $\top$ *is not an atomic state for the domain PAR, we can instead consider a covering* $\{\epsilon_1, \epsilon_2\}$ *for the state* ϵ*, where* $\epsilon_1 = \langle even, odd, even \rangle$ *and* $\epsilon_2 = \langle even, odd, odd \rangle$*. Hence, the relevancy of the statement s at program point p (if* ϵ *occurs at p) is computed over* ϵ_1 *and* ϵ_2*. Observe that the elements in the covering*

contains atomic abstract values (atoms, in other word) for the variables.

Consider the example shown in Figure 3. One abstract state possibly reaching program points 9 and 10 w.r.t. SIGN is $\epsilon = \langle -, \top, \top, \top \rangle$ *where* $dom(\epsilon) = \langle i, x, y, w \rangle$*. Since the values for* x, y, w *are provided by the user, it can be any value from the set* $\{+, 0, -\}$ *and is denoted by the top element* $\top$ *of the lattice for SIGN. When we compute the semantic relevancy of statements 9 and 10, the execution over the abstract state* ϵ *can not reveal the fact of semantic relevancy because of the overapproximated state and lack of precision. Therefore, we compute the semantic relevancy of 9 and 10 over the covering of* ϵ *i.e.* $\{\langle -,-,-,- \rangle, \langle -,0,-,- \rangle, \langle -,+,-,- \rangle, \ldots, \langle -,+,+,+ \rangle\}$*. This way, we can easily conclude about the semantic irrelevancy of 9 and the semantic relevancy of 10 w.r.t. SIGN.*

6 Algorithm for Semantics-based Abstract PDG

We are now in position to formalize a new algorithm to construct semantics-based abstract PDG $G^{r,d}_{pdg}$ of a program P *w.r.t.* an abstract property ρ as depicted in Figure 14. In this proposed algorithm, we combine (*i*) the notion of semantic relevancy of statements, and (*ii*) the notion of semantic data dependency of expressions.

We use the notation ϵ_{ij} to denote the j^{th} abstract state *w.r.t.* ρ possibly reaching program point p_i. The input of the algorithm is a program P and output is the semantics-based abstract PDG $G^{r,d}_{pdg}$ of P *w.r.t.* ρ.

Step 3 computes the semantic relevancy of all "*assignment*" statements in the program P *w.r.t.* ρ, and thus at step 6, P_{rel} contains only the relevant non-control statements along with all the control statements from P. Steps 7 computes the relevancy of control statements in P_{rel} *w.r.t.* ρ. Step 8 deals with the repetitive statement "*while*(*cond*) *then* blk_{while}", step 9 deals with the conditional statement "if(*cond*) *then* blk_{if}" and step 10 deals with the conditional statement "if(*cond*) *then* blk_{if} *else* blk_{else}", where we denote by blk_S a block of a set of statements S. Observe that steps 9 and 11 disregard the irrelevant control statements from P_{rel}, whereas steps 8, 12 and 13 replace the control statements by another form with equivalent relevancy *w.r.t.* ρ. In step 15, we compute abstract semantic data dependency for all expressions in P_{rel} by following the algorithm of Mastroeni and Zanardini [30]. Finally, in step 16, we construct PDG from P_{rel} that contains only the relevant statements and the relevant data dependences *w.r.t.* ρ.

The idea to obtain a semantics-based abstract PDG is to unfold the program into an equivalent program where only statements that have an impact *w.r.t.* the abstract domain are combined with the semantic data flow *w.r.t.* the same domain.

Algorithm 1: REFINE-PDG

Input: Program P and an abstract domain ρ
Output: Semantics-based Abstract PDG $G^{r,d}_{pdg}$ of P *w.r.t.* ρ

1. `FOR` each *assignment*-statement s at program point p_i in P `DO`
2. `FOR` all $\epsilon_{ij} \in \Sigma^{\rho}_{p_i}$ `DO`
3. Execute s on ϵ_{ij} and determine its relevancy;
4. `END FOR`
5. `END FOR`
6. Disregard all the irrelevant *assignment*-statements from P and generate its relevant version P_{rel};
7. `FOR` each *control*-statement in P_{rel} `DO`
8. Case 1: Repetitive statement "$while(cond)$ *then* blk_{while}":
   If the block blk_{while} is semantically irrelevant *w.r.t.* ρ, replace "$while(cond)$ *then* blk_{while}" in P_{rel} by the statement "$while(cond)$ *then skip*";
9. Case 2: Conditional statement "$if(cond)$ *then* blk_{if}":
   If the block blk_{if} is semantically irrelevant *w.r.t.* ρ, disregard "$if(cond)$ *then* blk_{if}" from P_{rel};
10. Case 3: Conditional statement "$if(cond)$ *then* blk_{if} *else* blk_{else}":
11. Case 3a: Both blk_{if} and blk_{else} are semantically irrelevant *w.r.t.* ρ:
    Disregard the statement "$if(cond)$ *then* blk_{if} *else* blk_{else}" from P_{rel};
12. Case 3b: Only blk_{else} is semantically irrelevant *w.r.t.* ρ:
    Replace the statement "$if(cond)$ *then* blk_{if} *else* blk_{else}" in P_{rel} by the statement "$if(cond)$ *then* blk_{if}";
13. Case 3c: Only blk_{if} is semantically irrelevant *w.r.t.* ρ:
    Replace the statement "$if(cond)$ *then* blk_{if} *else* blk_{else}" in P_{rel} by the statement "$if(cond)$ *then skip else* blk_{else}";
14. `END FOR`
15. Compute abstract semantic data dependency for all expressions in P_{rel} *w.r.t.* ρ by following the algorithm of Mastroeni and Zanardini;
16. Construct PDG from P_{rel} by using only the relevant statements and relevant data dependences *w.r.t.* ρ, as obtained in previous steps;

Figure 14: Algorithm to generate Semantics-based Abstract PDG

7 Reversing the order of Slicing and Relevancy Computation

In program slicing, we are interested only on a subset of program variables rather than all.

Till now, we have defined the relevancy of statements *w.r.t.* the abstract property of all program variables. We propose to construct a semantics-based abstract PDG from a syntactic PDG by computing statements relevancy and semantic data dependency.

However, if we apply slicing on the syntactic PDG first, and then, compute statements relevancy and semantic data dependency on this sliced program, we see that the computation complexity can be reduced.

When we perform slicing on a program, the resulting slice behaves like a reduced program with a subset of statements and subset of program variables with the same behavior as the original program. Thus, the computation of statements relevancy and semantic data dependency over these subset of statement *w.r.t.* the abstract property of subset of variables of interest reduce the overall time complexity.

However, this can not be applicable in the situations where repeated application of slicing with respect to different criteria is performed. In such case, we construct semantics-based abstract PDG once, and then, we perform the slicing repeatedly on it as many times as necessary with respect to different slicing criteria.

8 Dependence Condition Graph (DCG)

In this section, we extend the semantics-based abstract PDGs obtained so far into semantics-based abstract Dependence Condition Graphs (DCGs).

The notion of Dependence Condition Graphs (DCGs) is introduced by Sukumaran et al. in [39]. A DCG is built from a PDG by annotating each edge e of the PDG with the information e^b whose semantic interpretation encodes the condition for which the dependency represented by that edge actually arises in a program execution. The annotation e^b on any edge $e = e.src \rightarrow e.tgt$ is a pair $\langle e^R, e^A \rangle$. The first component e^R is referred to as Reach Sequences, and represents the conditions that should be true for an execution to ensure that the target $e.tgt$ of e is executed once the source $e.src$ of e is executed if e is a control dependence edge, or that the target $e.tgt$ is reached from the source $e.src$ if e is a data dependence edge. The component e^A is referred to as Avoid Sequences which is only relevant for data dependence edges (for control dependence edges it is $\emptyset$), and captures the possible condition under which the assignment at $e.src$ can be overwritten before it reaches $e.tgt$. Sukumaran et al. also described the semantics of DCG annotations in terms of execution semantics of the program over concrete domain.

Below, we first define the abstract semantics of DCG annotations in an abstract domain of interest. Then, we propose a refinement of the DCGs by removing semantically unrealizable dependences from them under their abstract semantics.

Abstract Semantics of $e^b \triangleq \langle e^R, e^A \rangle$

The program executions are recorded in finite or infinite sequences of states over a given set of commands, called traces. An execution trace ψ of a program P over an abstract domain ρ is a (possibly infinite) sequence $\langle (p_i, \epsilon_{p_i}) \rangle_{i \geq 0}$ where ϵ_{p_i} represents the abstract data state at the entry of the statement at program point p_i in P. We use the notion "$\iota : \ (p_i, \epsilon_{p_i})$" to denote that (p_i, ϵ_{p_i}) is the ι-th element in the sequence ψ. The trace ψ holds the following conditions:

1. The first abstract state in the sequence is (p_0, ϵ_{p_0}) where p_0 = "start" and ϵ_{p_0} is the initial abstract data state.
2. Each state (p_i, ϵ_{p_i}), $i = 1, 2, 3, \ldots$ is the successor of the previous state $(p_{i-1}, \epsilon_{p_{i-1}})$.
3. The last abstract state in the sequence ψ of length $\#\psi = m$, if it exists, is (p_m, ϵ_{p_m}) where p_m="stop".

Note that the DCG nodes corresponding to the statements at program points p_i are labeled by p_i. We denote the control dependence edge (CDG edge) in DCG by $e = p_i \xrightarrow{lab} p_j$, where the node p_i corresponds to the conditional or repetitive statement containing the condition $p_i.cond$ and the label $e.lab$ associated with e represents the truth value (either *true* or $false$). We denote the data dependence edge (DDG edge) in DCG by $e = p_i \xrightarrow{x} p_j$, where x is the data defined by the statement corresponding to the node p_i.

We now define the semantics of the annotations $e^b \triangleq (e^R, e^A)$ on dependence edges e in DCG in terms of the execution traces ψ over an abstract domain ρ.

Definition 8 *(Execution satisfying e^b for a CDG edge e at index ι)*
An execution trace ψ over an abstract domain ρ is said to satisfy e^b at index ι for a CDG edge $e = p_i \xrightarrow{lab} p_j$ (written as $\psi \vdash^{\rho}_{\iota} e$) if the following conditions hold:

- $e^b \triangleq \langle e^R, e^A \rangle = \langle \{e\}, \emptyset \rangle$, *and*
- *ψ contains $\iota : (p_i, \epsilon_{p_i})$ such that $[\![p_i.cond = e.lab]\!](\epsilon_{p_i})$ yields either to "true" or to the logic value "unknown" (meaning possibly true or possibly false).*

Definition 9 *(Execution satisfying e^b for a CDG edge e)*
An execution trace ψ over an abstract domain ρ satisfying e^b for a CDG edge $e = p_i \xrightarrow{lab} p_j$ (written as $\psi \vdash^{\rho} e$) is defined as:

$$\psi \vdash^{\rho} e \triangleq \exists \iota \geq 0 : \psi \vdash^{\rho}_{\iota} e$$

Definition 10 *(Execution satisfying e^R for a DDG edge e at ι)*
An execution trace ψ over an abstract domain ρ is said to satisfy e^R at index ι for a DDG edge $e = p_i \xrightarrow{x} p_j$ (written as $\psi \vdash^{R}_{\iota} e$) if the following conditions hold:

- *The trace ψ contains $\iota : (p_i, \epsilon_{p_i})$, and*
- *Either $e^R = \emptyset$ or for each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \ldots p_{s_n} \xrightarrow{lab_{s_n}} p_j) \in e^R$ where $p_{s_1}, p_{s_2}, \ldots, p_{s_n}$ correspond to conditional statements, the trace ψ contains $\iota_k : (p_{s_k}, \epsilon_{p_{s_k}})$ for $k = 1, \ldots, n$ and $\iota_1 < \iota < \iota_2 < \cdots < \iota_n$ and $\bigwedge_{1 \leq k \leq n} [\![p_{s_k}.cond = lab_{s_k}]\!](\epsilon_{p_{s_k}})$ yields to "true" or "unknown".*

Definition 11 *(Execution satisfying e^A for a DDG edge e at ι)*
An execution trace ψ over an abstract domain ρ is said to satisfy e^A at index ι for a DDG edge $e = p_i \xrightarrow{x} p_j$ (written as $\psi \vdash_\iota^A e$) if ψ contains $\iota : (p_i, \epsilon_{p_i})$, and for each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \ldots p_{s_n} \xrightarrow{lab_{s_n}} p') \in e^A$ the subtrace ψ' of ψ from the index ι to the next occurrence of (p_j, ϵ_{p_j}) (or, if (p_j, ϵ_{p_j}) does not occur then ψ' is the suffix of ψ starting from ι), satisfies exactly one of the following conditions:

- *ψ' does not contain $(p_{s_k}, \epsilon_{p_{s_k}})$ for $1 \leq k \leq n$, or*
- *$\exists\, k :\ 1 \leq k \leq n$: ψ' contains $(p_{s_k}, \epsilon_{p_{s_k}})$ such that $[\![p_{s_k}.cond = lab_{s_k}]\!](\epsilon_{p_{s_k}})$ yields to false.*

Definition 12 *(Execution satisfying e^b for a DDG edge e at ι)*
An execution trace ψ over an abstract domain ρ satisfying e^b at index ι for a DDG edge $e = p_i \xrightarrow{x} p_j$ (written as $\psi \vdash_\iota^\rho e$) is defined as

$$\psi \vdash_\iota^\rho e \triangleq (\psi \vdash_\iota^R e) \wedge (\psi \vdash_\iota^A e)$$

Definition 13 *(Execution satisfying e^b for a DDG edge e)*
An execution trace ψ over an abstract domain ρ satisfying e^b for a DDG edge $e = p_i \xrightarrow{x} p_j$ (written as $\psi \vdash^\rho e$) is defined as

$$\psi \vdash^\rho e \triangleq \exists \iota \geq 0 : \psi \vdash_\iota^\rho e$$

Theorem 8.1 *Given a DDG edge $e = p_i \xrightarrow{x} p_j$ and an execution trace ψ over an abstract domain ρ, the trace ψ satisfies e^b for e (denoted $\psi \vdash^\rho e$) iff the abstract value of x computed at p_i reaches the next occurrence of p_j in ψ.*

Proof *Since the execution trace ψ over an abstract domain ρ satisfies e^b for $e = p_i \xrightarrow{x} p_j$, we have*

$$\begin{aligned} \psi \vdash^\rho e &\triangleq \exists \iota \geq 0 :\ \psi \vdash_\iota^\rho e \\ &\triangleq \exists \iota \geq 0 :\ (\psi \vdash_\iota^R e) \wedge (\psi \vdash_\iota^A e) \end{aligned}$$

This means that ψ satisfies both the Reach Sequences e^R and the Avoid Sequences e^A for e at some index ι in ψ.

The trace ψ satisfies e^R at some index ι meaning that ψ contains $\iota : (p_i, \epsilon_{p_i})$, and ψ satisfies all the conditions in e^R which ensures that p_j is reached from p_i i.e. the abstract value of x computed at p_i can reach p_j. At the same time, the trace ψ satisfies e^A at ι meaning that ψ avoids the execution of all the other possibilities (if exists) so that the abstract value of x computed at p_i can not be overwritten by any other intermediate statements that also define x. Thus, $\psi \vdash^\rho e$ implies that the abstract value of x computed at p_i reaches the next occurrence of p_j in ψ.

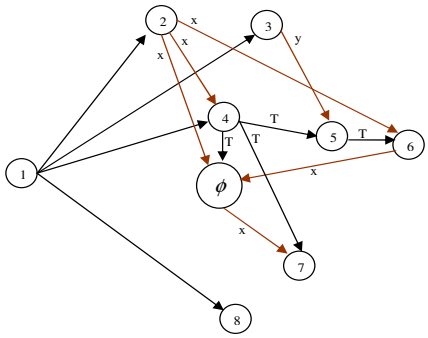
On the other side, we should also prove that if the abstract value of x computed at p_i reaches the next occurrence of p_j in ψ, then $\psi \vdash^\rho e$ where $e = p_i \xrightarrow{x} p_j$.

Since the abstract value of x computed at p_i reaches to the next occurrence of p_j, we can say that there is entries $\iota : (p_i, \epsilon_{p_i})$ and $\iota' : (p_j, \epsilon_{p_j})$ in ψ where ι' is the smallest index greater than ι and the statements corresponding to both p_i and p_j are executed in ψ (i.e. ψ satisfies Reach Sequences e^R), and at the same time it avoids the execution of all other intermediate statements which can overwrite the abstract value of x coming from p_i (i.e. ψ satisfies Avoid Conditions e^A). Thus, $\exists \iota \geq 0 : \ (\psi \vdash_\iota^R e) \wedge (\psi \vdash_\iota^A e)$ i.e. $\psi \vdash^\rho e$.

Example 10 *Consider the program P and its PDG depicted in Figure 15(a) and 15(b) respectively. The set of program variables in P is $VAR = \{x, y\}$. Consider the DDG edge $e = 2 \xrightarrow{x} \phi$. By following the algorithm in [39], we get $e^R = \{1 \xrightarrow{true} 4 \xrightarrow{true} \phi\}$ and $e^A = \{1 \xrightarrow{true} 4 \xrightarrow{true} 5 \xrightarrow{true} 6\}$. The DCG annotations over the DDG edges are shown in Figure 15(c). Consider the abstract domain SIGN. The initial state of P is defined*

```
1.    start
2.    x1 = input;
3.    y = input;
4.    if(x1 > 0){
5.          if(y == 5)
6.                x2 = x1 × 2;
φ.          x3 = f(x1, x2);
7.          print(x3);}
8.    stop
```

(a) P_{ssa} : *Program P in SSA form*



(b) G_{pdg} : *PDG of program* P_{ssa}

e	e^R	e^A
$2 \xrightarrow{x} 6$	$1 \xrightarrow{true} 4 \xrightarrow{true} 5 \xrightarrow{true} 6$	$\emptyset$
$2 \xrightarrow{x} 4$	$\emptyset$	$\emptyset$
$2 \xrightarrow{x} \phi$	$1 \xrightarrow{true} 4 \xrightarrow{true} \phi$	$1 \xrightarrow{true} 4 \xrightarrow{true} 5 \xrightarrow{true} 6$
$3 \xrightarrow{y} 5$	$1 \xrightarrow{true} 4 \xrightarrow{true} 5$	$\emptyset$
$6 \xrightarrow{x} \phi$	$\emptyset$	$\emptyset$
$\phi \xrightarrow{x} 7$	$\emptyset$	$\emptyset$

(c) *DCG annotations $\langle e^R, e^A \rangle$ for DDG edges e of* G_{pdg}

Figure 15: The program P_{ssa} and its DCG

by $\langle start, \epsilon_{start} \rangle = \langle start, (\bot, \bot) \rangle$ where $\epsilon_{start} = (\bot, \bot)$ are the initial abstract values for $x, y \in VAR$ respectively. Consider the execution trace

$\psi = \iota_1 : \langle 1, (\bot, \bot) \rangle \ \iota_2 : \langle 2, (\bot, \bot) \rangle \ \iota_3 : \langle 3, (+, \bot) \rangle \ \iota_4 : \langle 4, (+, -) \rangle \ \iota_5 : \langle 5, (+, -) \rangle$
$\iota_\phi : \langle \phi, (+, -) \rangle \ \iota_7 : \langle 7, (+, -) \rangle \ \iota_8 : \langle 8, (+, -) \rangle$

where in each state of ψ the first component represents program point of the corresponding statement and the other component represents abstract values of x and y respectively. Note that the condition from the statement at program point 1 to 4 is implicitly "true" irrespective of the states at 1. We have $\psi \vdash^R_{\iota_2} (2 \xrightarrow{x} \phi)$ because

- *ψ contains the entry $\iota_2 : \langle 2, (\bot, \bot)\rangle$ corresponding to the statement 2 at index ι_2, and*
- *ψ is of the form $\psi = \iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle \ldots \iota_4 : \langle 4, (+, -)\rangle \ldots \iota_\phi : \langle \phi, (+, -)\rangle \ldots$ for $1 \xrightarrow{true} 4 \xrightarrow{true} \phi \in (2 \xrightarrow{x} \phi)^R$ such that $[\![1.cond = true]\!](\bot, \bot)$ and $[\![4.cond = true]\!](+, -)$ are evaluated to "true".*

Similarly, $\psi \vdash^A_{\iota_2} (2 \xrightarrow{x} \phi)$, because for $1 \xrightarrow{true} 4 \xrightarrow{true} 5 \xrightarrow{true} 6 \in (2 \xrightarrow{x} \phi)^A$ the sub-trace of ψ contains the entry $\iota_5 : \langle 5, (+, -)\rangle$ such that $[\![5.cond = true]\!](+, -)$ is "false".

As $\psi \vdash^R_{\iota_2} (2 \xrightarrow{x} \phi)$ and $\psi \vdash^A_{\iota_2} (2 \xrightarrow{x} \phi)$, we can say $\psi \vdash^{SIGN}_{\iota_2} (2 \xrightarrow{x} \phi)$ meaning that in ψ the sign of x defined at program point 2 reaches program point ϕ, and it is not changed or overwritten by the intermediate statement 6.

Abstract Semantics of Dependence Paths in DCGs

The final step, in order to combine Dependence Condition Graphs with abstract semantics-based Program Dependence Graphs, is to define the abstract semantics of the dependence paths in a Dependence Condition Graph.

Given a program P and its DCG, we consider dependence paths in this graph. First we define the ϕ-sequence and then the semantics of a dependence path over an abstract domain ρ.

Definition 14 *(PhiSeqs)*
A ϕ-sequence η_ϕ is a DDG path of the form: $n_1 \to \phi_1 \to \phi_2 \to \cdots \to \phi_k \to n_2$, where n_1 and n_2 are nodes of the program and all the ϕ_i ($1 \le i \le k$) are ϕ-nodes (that correspond to assignments to the same variable along different paths). Observe that all edges on a ϕ-sequence will be labeled with the same variable.

Consider an arbitrary dependence path $\eta = e_1 e_2 \ldots e_n$ in DCG representing a chain of dependences. To satisfy η by an execution trace ψ over an abstract domain ρ, we need to satisfy the annotations e^b of each edge e_i, $i \in [1..n]$, at some ι_i (*i.e.*, $\psi \vdash^\rho_{\iota_i} e_i$) such that the execution sub-traces of ψ corresponding to the e_i are contiguous.

Definition 15 *(Evidence)*
For an execution trace ψ over an abstract domain ρ and a dependence edge e, s.t. $\psi \vdash^\rho_\iota e$, $evidence(\psi, e, \iota) = \iota'$ where ι' is the index of the first occurrence of $(e.tgt, -)$ in ψ from index ι.

Definition 16 *(Execution satisfying a dependence path)*
A series of program dependences represented by a dependence path $\eta = e_1 e_2 \ldots e_n$ is said to be satisfied by an execution ψ over an abstract domain ρ (written as $\psi \vdash^{\rho} \eta$) if

$$\bigwedge_{1 \le i \le n} \psi \vdash^{\rho}_{\iota_i} e_i \quad \wedge \; (\; \forall 1 \le i \le n : evidence(\psi, e_i, \iota_i) = \iota_{i+1} \;)$$

Theorem 8.2 *Given a ϕ-sequence $\eta_\phi = e_1 e_2 \ldots e_n$ and the execution trace ψ over an abstract domain ρ, the trace ψ satisfies η_ϕ (denoted $\psi \vdash^{\rho} \eta_\phi$) iff the abstract value computed at $e_1.src$ reaches $e_n.tgt$ in ψ along the execution path that satisfies η_ϕ.*

Proof *Since $\psi \vdash^{\rho} \eta_\phi$, we have*

$$\bigwedge_{1 \le i \le n} \psi \vdash^{\rho}_{\iota_i} e_i \quad \wedge \; (\; \forall 1 \le i \le n : evidence(\psi, e_i, \iota_i) = \iota_{i+1} \;)$$

This means that the sub-traces ψ_i of ψ satisfying the annotations of e_i of η_ϕ, where $i = 1, \ldots, n$, are contiguous. Observe that all the edges e_i of η_ϕ are labeled with the same variable x. Consider any two consecutive edges $e_i = p_r \xrightarrow{x} p_s$ and $e_{i+1} = p_s \xrightarrow{x} p_t$ in η_ϕ where $1 \le i < i+1 \le n$ and the corresponding contiguous sub-traces ψ_i and ψ_{i+1} that satisfy e_i and e_{i+1} respectively. From Theorem 8.1, we can say that the abstract value of x can reach from p_r to p_s and from p_s to p_t in ψ_i and ψ_{i+1} respectively. If the intermediate node p_s is a ϕ node (which does not recompute the value but only pass through), then this transitivity implies that the abstract value of x can reach from p_r to p_t in ψ_q where ψ_q is the concatenation of ψ_i and ψ_{i+1}. Since in a ϕ-sequence all the intermediate nodes are ϕ nodes, we can extend this transitivity for all i from 1 to n, and we can say that the abstract value of x can reach from $e_1.src$ to $e_n.tgt$ in ψ where ψ is the concatenation of all subtraces ψ_i, $i = 1, \ldots, n$. Observe that in a ϕ-sequence the starting and end nodes are not ϕ nodes, and the datum is computed at starting node and is used by the end node. Thus, the abstract value computed at $e_1.src$ reaches $e_n.tgt$ in ψ.

Let us now prove the "only if" part of the theorem: given a ϕ-sequence $\eta_\phi = e_1 e_2 \ldots e_n$ and an execution trace ψ over an abstract domain ρ, if the abstract value at $e_1.src$ reaches $e_n.tgt$ in ψ, then $\psi \vdash^{\rho} \eta_\phi$.

Since in ϕ-sequence $\eta_\phi = e_1 e_2 \ldots e_n$ all the intermediate nodes are ϕ nodes except the starting and end ones, and the datum computed at $e_1.src$ reaches $e_n.tgt$ in ψ, from Theorem 8.1 we can say that $\forall i$, $1 \le i \le n$: $\psi \vdash^{\rho}_{\iota_i} e_i$ where ι_i is the index of $(e_i.src, -)$ in ψ. Now we show that $evidence(\psi, e_i, \iota_i) = \iota_{i+1}$ for all i, $1 \le i \le n$. Consider two consecutive edges $e_i = p_r \xrightarrow{x} p_s$ and $e_{i+1} = p_s \xrightarrow{x} p_t$ in η_ϕ where $1 \le i < i+1 \le n$. Since $\psi \vdash^{\rho}_{\iota_i} e_i$ and $\psi \vdash^{\rho}_{\iota_{i+1}} e_{i+1}$, the trace ψ contains $\iota_i : (p_r, \epsilon_{p_r})$ and $\iota_{i+1} : (p_s, \epsilon_{p_s})$. Thus, we have $evidence(\psi, e_i, \iota_i) = \iota_{i+1}$ because ι_{i+1} is the index of the first occurrence of $(e_i.tgt, -)$ i.e. (p_s, ϵ_{p_s}) in ψ from the index ι_i. Therefore, we have

$$\bigwedge_{1 \le i \le n} \psi \vdash^{\rho}_{\iota_i} e_i \quad \wedge \; (\; \forall 1 \le i \le n : evidence(\psi, e_i, \iota_i) = \iota_{i+1} \;)$$

That is,

$$\psi \vdash^{\rho} \eta_\phi.$$

Example 11 *Consider the dependence path* $\eta = 2 \xrightarrow{x} 6 \xrightarrow{x} \phi \xrightarrow{x} 7$ *in the graph of Figure 15, and the following execution trace over the abstract domain SIGN:*

$\psi = \iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle\ \iota_3 : \langle 3, (+, \bot)\rangle\ \iota_4 : \langle 4, (+, +)\rangle\ \iota_5 : \langle 5, (+, +)\rangle$
$\iota_6 : \langle 6, (+, +)\rangle\ \iota_7 : \langle \phi, (+, +)\rangle\ \iota_8 : \langle 7, (+, +)\rangle\ \iota_9 : \langle 8, (+, +)\rangle$

The trace ψ *satisfies* e^b *for all the edges* $2 \xrightarrow{x} 6$, $6 \xrightarrow{x} \phi$ *and* $\phi \xrightarrow{x} 7$ *of* η*, and the sub-traces of* ψ *that satisfy these edges are contiguous, that is,*

- $\psi \vdash^{SIGN}_{\iota_2} (2 \xrightarrow{x} 6)$ *and evidence*$(\psi, 2 \xrightarrow{x} 6, \iota_2) = \iota_6$,

  where $1 \xrightarrow{true} 4 \xrightarrow{true} 5 \xrightarrow{true} 6 \in (2 \xrightarrow{x} 6)^R$ *and* $(2 \xrightarrow{x} 6)^A = \emptyset$ *and* ψ *is of the form* $\iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle \ldots \iota_4 : \langle 4, (+, +)\rangle\ \iota_5 : \langle 5, (+, +)\rangle\ \iota_6 : \langle 6, (+, +)\rangle \ldots$ *such that* $[\![1.cond = true]\!](\bot, \bot)$, $[\![4.cond = true]\!](+, +)$ *are evaluated to "true" and* $[\![5.cond = true]\!](+, +)$ *is evaluated to "unknown".*

- $\psi \vdash^{SIGN}_{\iota_6} (6 \xrightarrow{x} \phi)$ *and evidence*$(\psi, 6 \xrightarrow{x} \phi, \iota_6) = \iota_7$,

  where $(6 \xrightarrow{x} \phi)^R = \emptyset$ *and* $(6 \xrightarrow{x} \phi)^A = \emptyset$ *and* ψ *is of the form* $\ldots \iota_6 : \langle 6, (+, +)\rangle$ $\iota_7 : \langle \phi, (+, +)\rangle \ldots$.

- $\psi \vdash^{SIGN}_{\iota_7} (\phi \xrightarrow{x} 7)$ *and evidence*$(\psi, \phi \xrightarrow{x} 7, \iota_7) = \iota_8$,

  where $(\phi \xrightarrow{x} 7)^R = \emptyset$ *and* $(\phi \xrightarrow{x} 7)^A = \emptyset$ *and* ψ *is of the form* $\ldots \iota_7 : \langle \phi, (+, +)\rangle$ $\iota_8 : \langle 7, (+, +)\rangle \ldots$.

Thus, we can say that the dependence path η *is satisfied by* ψ *over the abstract domain SIGN i.e.* $\psi \vdash^{SIGN} \eta$.

Satisfiability of Dependence Paths with Semantic Relevancy

Let η be a dependence path in a DCG. Suppose an execution trace ψ over an abstract domain ρ satisfies η (denoted $\psi \vdash^{\rho} \eta$). If we compute semantic relevancy *w.r.t.* ρ and we disregard the irrelevant entries from both η and ψ, we see that the satisfiability of the refined path is also preserved, as depicted in Theorem 8.3.

Theorem 8.3 *Given a program P and its DCG* G_{dcg}*. Let* ψ *be an execution trace of P over an abstract domain* ρ*, and* $\eta = e_1 e_2 \ldots e_l e_{l+1} \ldots e_h$ *be a dependence path in* G_{dcg} *where* $e_l : p_i \xrightarrow{x} p_j$ *and* $e_{l+1} : p_j \xrightarrow{x} p_k$ *(*e_l *and* e_{l+1} *are contiguous). Suppose removal of the element corresponding to irrelevant statement at* p_j *w.r.t.* ρ *from* η *and* ψ *yield to a dependence path* $\eta' = e_1 e_2 \ldots e_q \ldots e_h$*, where* $e_q : p_i \xrightarrow{x} p_k$*, and a trace* ψ' *respectively. Then,*

$$if\ \psi \vdash^{\rho} \eta,\ then\ \psi' \vdash^{\rho} \eta'$$

.

Proof *Since $\psi \vdash^{\rho} \eta$, we can say that for the edges $e_l : p_i \xrightarrow{x} p_j$ and $e_{l+1} : p_j \xrightarrow{x} p_k$ in η:*

- $\psi \vdash^{R}_{\iota_i} e_l \wedge \psi \vdash^{A}_{\iota_i} e_l \wedge evidence(\psi, e_l, \iota_i) = \iota_j$, *and*
- $\psi \vdash^{R}_{\iota_j} e_{l+1} \wedge \psi \vdash^{A}_{\iota_j} e_{l+1} \wedge evidence(\psi, e_{l+1}, \iota_j) = \iota_k$

where, ι_i, ι_j and ι_k are the indexes where (p_i, ϵ_{p_i}), (p_j, ϵ_{p_j}) and (p_k, ϵ_{p_k}) occur respectively in ψ.

We already know that the dependence edges e in DCG are annotated by the Reach Sequences e^R and the Avoid Sequences e^A. The Reach Sequences e^R for the edge $e : e.src \xrightarrow{x} e.tgt$ represents the conditions that need to be satisfied by the execution trace ψ to reach the data x from e.src to e.tgt. If $e^R = \emptyset$, it means that e.tgt post-dominates e.src and thus, the execution trace should contain e.src, and once e.src is executed e.tgt will also be executed which yield x to reach from e.src to e.tgt. When $e^R \neq \emptyset$, e.tgt does not post-dominate e.src and thus, the conditions in e^R need to be satisfied by the trace ψ so that e.tgt executes, and since ψ also contains e.src the data x must reach from e.src to e.tgt. Therefore, we have ψ as follows under the four different cases of Reach Sequences:

r_1: $e^R_l = e^R_{l+1} = \emptyset$: *$p_i$ is post-dominated by p_j and p_j is post-dominated by p_k, and ψ contains $\iota_i : (p_i, \epsilon_{p_i})$, $\iota_j : (p_j, \epsilon_{p_j})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_j < \iota_k$.*

r_2: $e^R_l \neq \emptyset$ *and* $e^R_{l+1} = \emptyset$: *ψ contains $\iota_i : (p_i, \epsilon_{p_i})$, $\iota_j : (p_j, \epsilon_{p_j})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_j < \iota_k$. For each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_n} \xrightarrow{lab_{s_n}} p_j) \in e^R_l$, ψ contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, n$ and $\iota_1 < \iota_i < \iota_2 < \dots < \iota_n < \iota_j < \iota_k$ and $\bigwedge_{1 \leq m \leq n} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown", and p_j is post-dominated by p_k.*

r_3: $e^R_l = \emptyset$ *and* $e^R_{l+1} \neq \emptyset$: *p_i is post-dominated by p_j and ψ contains $\iota_i : (p_i, \epsilon_{p_i})$, $\iota_j : (p_j, \epsilon_{p_j})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_j < \iota_k$. For each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_n} \xrightarrow{lab_{s_n}} p_k) \in e^R_{l+1}$, ψ contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, n$ and $\iota_1 < \iota_i < \iota_j < \iota_2 < \dots < \iota_n < \iota_k$ and $\bigwedge_{1 \leq m \leq n} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown".*

r_4: $e^R_2 \neq \emptyset$ *and* $e^R_3 \neq \emptyset$: *ψ contains $\iota_i : (p_i, \epsilon_{p_i})$, $\iota_j : (p_j, \epsilon_{p_j})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_j < \iota_k$. For each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_n} \xrightarrow{lab_{s_n}} p_j) \in e^R_l$ and $(p_{s_n} \xrightarrow{lab_{s_n}} p_{s_{n+1}} \xrightarrow{lab_{s_{n+1}}} \dots p_{s_r} \xrightarrow{lab_{s_r}} p_k) \in e^R_{l+1}$, ψ contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, r$ and $\iota_1 < \iota_i < \iota_2 < \dots < \iota_n < \iota_j < \iota_{n+1} < \dots < \iota_r < \iota_k$ and $\bigwedge_{1 \leq m \leq r} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown".*

We know that after computing semantic relevancy w.r.t. ρ and after removing the irrelevant element corresponding to p_j from η and ψ, we get $\eta' = e_1 e_2 \dots e_q \dots e_h$ where $e_q : p_i \xrightarrow{x} p_k$ and the execution trace ψ'. Now we have to show that $\psi' \vdash^{\rho} \eta'$. That is $\psi' \vdash^{R}_{\iota_i} e_q \wedge \psi' \vdash^{A}_{\iota_i} e_q$ and $evidence(\psi, e_q, \iota_i) = \iota_k$.

Corresponding to the above four cases r_1, r_2, r_3 and r_4, we have the following four cases:

r'_1: *$[e^R_l = \emptyset, e^R_{l+1} = \emptyset]$ $e^R_q = \emptyset$: Since p_i is post-dominated by p_j and p_j is post-dominated by p_k, after removing the irrelevant entry corresponding to p_j, we have that p_i is post-dominated by p_k. Since the trace ψ' contains $\iota_i : (p_i, \epsilon_{p_i})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_k$, we get $\psi' \vdash^R_{\iota_i} e_q$.*

r'_2: *$[e^R_l \neq \emptyset, e^R_{l+1} = \emptyset]$:*

- *$e^R_q = \emptyset$: Since p_i is not dominated by p_j and p_j is post-dominated by p_k, after removing the irrelevant element corresponding to p_j, it may happen that p_i is post-dominated by p_k. Since ψ' contains $\iota_i : (p_i, \epsilon_{p_i})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_k$, we get $\psi' \vdash^R_{\iota_i} e_q$.*
- *$e^R_q \neq \emptyset$: After removing the irrelevant element corresponding to p_j, we have that p_i is not post-dominated by p_k. In such case, the trace ψ' contains $\iota_i : (p_i, \epsilon_{p_i})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_k$ and for each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_t} \xrightarrow{lab_{s_t}} p_k) \in e^R_q$ where $t \in [2...n]$, the trace ψ' contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, t$ and $\iota_1 < \iota_i < \iota_2 < \dots < \iota_t < \iota_k$ and $\bigwedge_{1 \leq m \leq t} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown". Thus, $\psi' \vdash^R_{\iota_i} e_q$.*

r'_3: *$[e^R_l = \emptyset, e^R_{l+1} \neq \emptyset]$ $e^R_q \neq \emptyset$: Here removal of irrelevant element corresponding to p_j, we have that p_i is not post-dominated by p_k. In such case ψ' contains $\iota_i : (p_i, \epsilon_{p_i})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_k$ and for each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_n} \xrightarrow{lab_{s_n}} p_k) \in e^R_{l+1}$, ψ' contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, n$ and $\iota_1 < \iota_i < \iota_2 < \dots < \iota_n < \iota_k$ and $\bigwedge_{1 \leq m \leq n} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown". Thus, $\psi' \vdash^R_{\iota_i} e_q$.*

r'_4: *$[e^R_l \neq \emptyset, e^R_{l+1} \neq \emptyset]$ $e^R_q \neq \emptyset$: Since p_i is not post-dominated by p_j and p_j is not post-dominated by p_k, the removal of element corresponding to p_j results that p_i is never post-dominated by p_k. In such case the trace ψ' contains $\iota_i : (p_i, \epsilon_{p_i})$ and $\iota_k : (p_k, \epsilon_{p_k})$ where $\iota_i < \iota_k$ and for each $(p_{s_1} \xrightarrow{lab_{s_1}} p_{s_2} \xrightarrow{lab_{s_2}} \dots p_{s_t} \xrightarrow{lab_{s_t}} p_{s_{t+1}} \xrightarrow{lab_{s_{t+1}}} \dots p_{s_r} \xrightarrow{lab_{s_r}} p_k) \in e^R_q$ where $t \in [1...n]$, ψ' contains $\iota_m : (p_{s_m}, \epsilon_{p_{s_m}})$ for $m = 1, \dots, r$ and $\iota_1 < \iota_i < \iota_2 < \dots < \iota_t < \iota_{t+1} < \dots < \iota_r < \iota_k$ and $\bigwedge_{1 \leq m \leq r} [\![p_{s_m}.cond = lab_{s_m}]\!](\epsilon_{p_{s_m}})$ yields to "true" or "unknown".*

Similarly, we can prove that $\psi' \vdash^A_{\iota_1} e_q$.

Thus, for any dependence path $\eta = e_1 e_2 \dots e_l e_{l+1} \dots e_h$ where $e_l : p_i \xrightarrow{x} p_j$ and $e_{l+1} : p_j \xrightarrow{x} p_k$, and an execution trace ψ over an abstract domain ρ: if $\psi \vdash^\rho \eta$ then $\psi' \vdash^\rho \eta'$, where $\eta' = e_1 e_2 \dots e_q \dots e_h$ (where $e_q = p_i \xrightarrow{x} p_k$) and ψ' are the dependence path and the execution trace corresponding to η and ψ respectively obtained after computing semantic relevancy w.r.t. ρ.

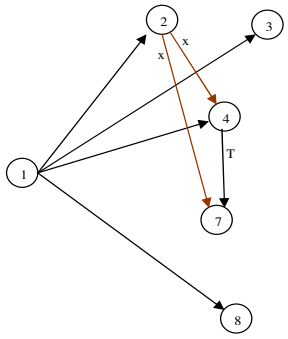
Example 12 *Look at Figure 15 and consider the dependence path $\eta = 2 \xrightarrow{x} 6 \xrightarrow{x} \phi \xrightarrow{x} 7$ and the following execution trace over the abstract domain SIGN:*

$\psi = \iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle\ \iota_3 : \langle 3, (+, \bot)\rangle\ \iota_4 : \langle 4, (+, +)\rangle\ \iota_5 : \langle 5, (+, +)\rangle$
$\iota_6 : \langle 6, (+, +)\rangle\ \iota_7 : \langle \phi, (+, +)\rangle\ \iota_8 : \langle 7, (+, +)\rangle\ \iota_9 : \langle 8, (+, +)\rangle$

Note that $\psi \vdash^{SIGN} \eta$*, as already shown in Example 11.*

1. start
2. $x = input;$
3. $y = input;$
4. $if(x > 0)\{$
7. $\quad print(x);\}$
8. stop

(a) P'_{ssa} : *after computing Semantic Relevancy of* P_{ssa} *w.r.t. SIGN*



(b) G'_{pdg} : *PDG of* P'_{ssa}

e	e^R	e^A
$2 \xrightarrow{x} 7$	$1 \xrightarrow{true} 4 \xrightarrow{true} 7$	$\emptyset$
$2 \xrightarrow{x} 4$	$\emptyset$	$\emptyset$

(c) *DCG annotations* $\langle e^R, e^A\rangle$ *for the DDG edges e of* G'_{pdg}

Figure 16: Program P'_{ssa} and its DCG after relevancy computation

Figure 16(a) and 16(b) depict the program P'_{ssa} *and its PDG* G'_{pdg} *which are obtained after computing semantic relevancy w.r.t. SIGN. Observe that in* P_{ssa} *the statement at program point 6 is irrelevant w.r.t. SIGN. Therefore, we can remove the conditional statement at 5 and the* ϕ *statement, because after removing statement 6 the corresponding "*if*" block becomes semantically irrelevant and the SSA function* f *is not necessary anymore, as x has just a single definition. The DCG annotations over the DDG edges of* G'_{pdg} *are shown in Figure 16(c).*

After removing the irrelevant entries from η *and* ψ*, we get the dependence path* $\eta' = 2 \xrightarrow{x} 7$*, and the execution trace* ψ' *as follows:*

$\psi' = \iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle\ \iota_3 : \langle 3, (+, \bot)\rangle\ \iota_4 : \langle 4, (+, +)\rangle\ \iota_8 : \langle 7, (+, +)\rangle$
$\iota_9 : \langle 8, (+, +)\rangle$

Now, let us show that $\psi' \vdash^{SIGN} \eta'$*.*

Algorithm 2: REFINE-DCG

Input: Syntactic DCG G_{dcg} and an abstract domain ρ
Output: Semantics-based abstract DCG G^{s}_{dcg} *w.r.t.* ρ

1. FOR each nodes $q \in G_{dcg}$ DO
2. If $\forall\psi$: $\psi \not\vdash^{\rho} (p \xrightarrow{lab} q)$ where $lab \in \{true, false\}$ THEN
3. Remove from G_{dcg} the node q and all its associated dependences. If q is a control node, the removal of q also removes all the nodes transitively control-dependent on it. Data dependences have to be re-adjusted accordingly;
4. END IF
5. FOR each data dependence edge $e = (q \xrightarrow{x} p_i)$ DO
6. IF $\forall\psi$: $\psi \not\vdash^{\rho} e$ THEN
7. Remove e from G_{dcg} and re-adjust the data dependence of p_i for the data x;
8. END IF
9. END FOR
10. FLAG:=true;
11. FOR each ϕ-sequences $\eta_{\phi} = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \ldots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ starting from q DO
12. IF $\exists\psi$: $\psi \vdash^{\rho} \eta_{\phi}$ THEN
13. FLAG:=false;
14. BREAK;
15. END IF
16. END FOR
17. IF FLAG==true THEN
18. Remove the edge $q \xrightarrow{x} \phi_1$;
19. END IF
20. END FOR

Figure 17: Algorithm to generate Semantics-based Abstract DCG

In η', for the edge $2 \xrightarrow{x} 7$, the statement at 7 does not post-dominate the statement at 2. The Reach Sequences and the Avoid Sequences for the edge $e = 2 \xrightarrow{x} 7$ are $(2 \xrightarrow{x} 7)^R = \{1 \xrightarrow{true} 4 \xrightarrow{true} 7\}$ and $(2 \xrightarrow{x} 7)^A = \emptyset$ respectively. For $1 \xrightarrow{true} 4 \xrightarrow{true} 7 \in (2 \xrightarrow{x} 7)^R$: $[\![1.cond = true]\!](\perp, \perp)$ and $[\![4.cond = true]\!](+, +)$ yields to true. Thus, $\psi' \vdash^{R}_{\iota_1} (2 \xrightarrow{x} 7)$.

The Avoid Sequence behaves similarly, yielding to $\psi' \vdash^{SIGN} \eta'$.

8.1 Refinement into Semantics-based Abstract DCG

Given a DCG, we can refine it into more precise semantics-based abstract DCG by removing from it all the semantically unrealizable dependences where conditions for a control dependence never be satisfiable or data defined at a source node can never be reachable to a target node in all possible abstract execution

traces. The notion of semantically unrealizable dependence path is defined in Definition 17.

Definition 17 *(Semantically Unrealizable Dependence Path)*
Given a DCG G_{dcg} and an abstract domain ρ. A dependence path $\eta \in G_{dcg}$ is called semantically unrealizable in the abstract domain ρ if $\forall\psi$: $\psi \not\vdash^{\rho} \eta$, where ψ is an abstract execution trace.

The refinement algorithm of a DCG from syntactic to semantic one is depicted in Figure 17. Step 2 says that if the condition on which a node q is control-dependent, never be satisfied by any of the execution traces, then the node and all its associated dependences are removed. In that case, if q is a control node, we remove all the nodes transitively control-dependent on q. If any DDG edge with q as source is semantically unrealizable under its abstract semantics, the corresponding DDG edge is removed in step 5. If all the ϕ-sequences emerging from q are semantically unrealizable under its abstract semantics, we remove the dependence of the ϕ-sequences on q in step 11.

Observe that in case of static slicing the satisfiability of the dependence paths are checked against all possible traces of the program, whereas in case of dynamic slicing or other forms of slicing the checking is performed against the traces generated only for the inputs of interest.

9 Slicing Algorithm

We are now in position to formalize our proposed slicing algorithm, depicted in Figure 18, that takes a program P and an abstract slicing criteria $\langle p, v, \rho\rangle$ as inputs, and produces an abstract slice *w.r.t.* $\langle p, v, \rho\rangle$ as output. The proposed slicing algorithm make the use of semantics-based abstract DCG of the program that is obtained in two steps: first by generating semantics based abstract PDG by following the algorithm `REFINE-PDG` depicted in section 6, and then by converting it into semantics-based abstract DCG by following the algorithm `REFINE-DCG` depicted in section 8.

Observe that the sub-DCG G_{sdcg} which is obtained in step 4 by applying slicing criteria on the semantics-based abstract DCG G^{s}_{dcg}, is further refined in step 5 by removing unrealizable data dependences, if present, from it. Let us illustrate the reason behind it with an example. Consider the graph in Figure 19(a) showing a portion of DCG with three ϕ-sequences ϕ_1, ϕ_2 and ϕ_3 that describe the data dependences of the nodes 3, 5 and 7 respectively on the node 1 for a data y.

- $\eta_1 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} 3$
- $\eta_2 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_2 \xrightarrow{y} 5$
- $\eta_3 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_2 \xrightarrow{y} \phi_3 \xrightarrow{y} 7$

Algorithm 3: GEN-SLICE

Input: Program P and an abstract slicing criteria $\langle p, v, \rho \rangle$
Output: Abstract Slice *w.r.t.* $\langle p, v, \rho \rangle$

1. Generate a semantics-based abstract PDG $G^{r,d}_{pdg}$ from the program P by following the algorithm `REFINE-PDG`.
2. Convert $G^{r,d}_{pdg}$ into the corresponding DCG G_{dcg} by computing annotations over all the data/control edges of it.
3. Generate a semantics-based abstract DCG G^{s}_{dcg} from G_{dcg} by following the algorithm `REFINE-DCG`.
4. Apply the criteria $\langle p, v \rangle$ on G^{s}_{dcg} by following PDG-based slicing techniques [33] and generate a sub-DCG G_{sdcg} that includes the node corresponding to the program point p as well.
5. Refine G_{sdcg} into more precise one G'_{sdcg} by performing the following operation for all nodes $q \in G_{sdcg}$:

   $\forall \eta_\phi = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \ldots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ and $\forall \psi$: if $\psi \not\vdash^{\rho} \eta_\phi$, then remove the edge $q \xrightarrow{x} \phi_1$ from G_{sdcg}.

6. Apply again the criteria $\langle p, v \rangle$ on G'_{sdcg} that results into the desired slice.

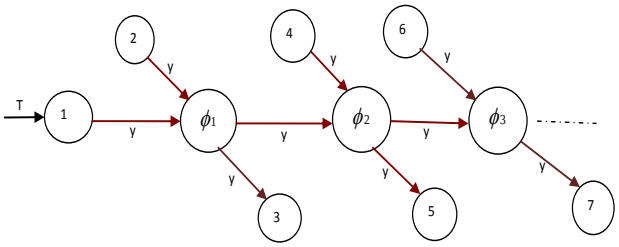
Figure 18: Slicing Algorithm

Suppose, $\exists \psi$: $\psi \vdash^{\rho} \eta_1 \bigwedge \forall \psi$: $\psi \not\vdash^{\rho} (\eta_2 \bigwedge \eta_3)$. During refinement of a DCG in the algorithm `REFINE-DCG`, we can not remove the dependence edge $1 \xrightarrow{y} \phi_1$ because there is one semantically realizable ϕ-sequence ϕ_1 from node 1.

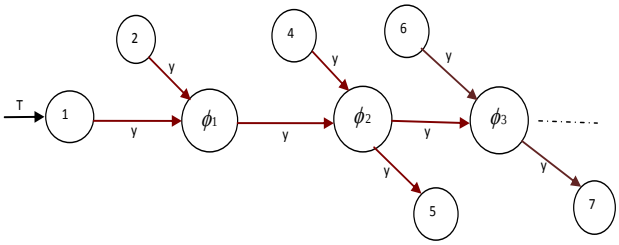
Given a slicing criteria C. In algorithm `GEN-SLICE`, suppose the sub-DCG generated after applying C does not include the node 3, as depicted in Figure 19(b). Now if we apply step 5 on the sub-DCG, we see that all the ϕ-sequences emerging from node 1 (ϕ_2 and ϕ_3) are not semantically realizable. Therefore, we can remove the edge $1 \xrightarrow{y} \phi_1$ from it as depicted in Figure 19(c). The further application of the slicing criteria C (in step 6) on this refined sub-DCG generate a slice that does not include the statement corresponding to the node 1 any more.

10 Idempotency

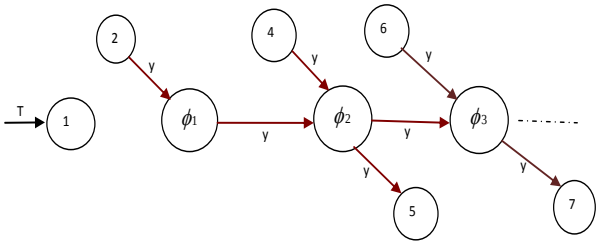
In this Section, we show that our slicing technique is idempotent. That means, if we apply our slicing procedure more than once, it yields the same result if we apply it only once.



(a) A part of a DCG containing three ϕ-sequences



(b) Sub-DCG after applying Slicing Criteria C



(c) Refiement of Sub-DCG

Figure 19: Refinement of sub-DCG during slicing

Given a set of programs $\mathbb{P}$. Let f_ρ be a mapping function from $\mathbb{P}$ to $\mathbb{P}^\rho_{sem}$, where $\mathbb{P}^\rho_{sem}$ is the set of programs that are obtained after computing semantics-based statement relevancy of the programs in $\mathbb{P}$ *w.r.t.* the property ρ and by disregarding the irrelevant statements from them. That is,

$$f_\rho : \mathbb{P} \rightarrow \mathbb{P}^\rho_{sem}$$

Let S_c be a function that generates slice *w.r.t.* the slicing criteria c from the programs in $\mathbb{P}^\rho_{sem}$. Thus,

$$S_c : \mathbb{P}^\rho_{sem} \rightarrow Slice^\rho_c$$

Thus, we have,

$$g^\rho_c = S_c \circ f_\rho : \mathbb{P} \rightarrow Slice^\rho_c$$

The next theorem shows that g^ρ_c is idempotent, *i.e.*, there is no benefit by applying the slicing procedure more than once.

Theorem 10.1 *Given a set of programs* $\mathbb{P}$ *and two mapping functions* $f_\rho : \mathbb{P} \to \mathbb{P}^\rho_{sem}$ *and* $S_c : \mathbb{P}^\rho_{sem} \to Slice^\rho_c$*, where* $\mathbb{P}^\rho_{sem}$ *is the set of programs obtained after removing irrelevant statements from* $P \in \mathbb{P}$ *w.r.t. an abstract property* ρ*, and* $Slice^\rho_c$ *is the set of sliced programs w.r.t. the criteria c. The application of* $g^\rho_c = S_c \circ f_\rho$ *on* $P \in \mathbb{P}$ *is idempotent, i.e.,*

$$\forall P \in \mathbb{P},\ g^\rho_c(g^\rho_c(P)) = g^\rho_c(P)$$

Proof Since $g^\rho_c = S_c \circ f_\rho$, we have $g^\rho_c(P) = S_c \circ f_\rho(P)$ where $P \in \mathbb{P}$. The first application of f_ρ removes from P all the irrelevant statements *w.r.t.* ρ and yield to $P_{rel} \in \mathbb{P}^\rho_{sem}$.

Now we prove that the removal of any irrelevant statement $s \in P$ by the application of f_ρ does not make any other relevant statement $s' \in P$ irrelevant, and thus, all statements in P_{rel} are relevant *w.r.t.* ρ. We prove it by contradiction.

Suppose the removal of irrelevant statement s at program point p in P makes another relevant statement s' at program point p' in P irrelevant, and s' appears in P_{rel}.

Since s' at p' is relevant, we can say $\exists \epsilon \in \Sigma^\rho_{p'} : S[\![s']\!]^\rho(\epsilon) \neq \epsilon$. The removal of irrelevant statement s makes s' irrelevant, that is, all ϵ at p' for which s' is relevant change into different states ϵ' such that $S[\![s']\!]^\rho(\epsilon') = \epsilon'$.

Since s is irrelevant *w.r.t.* ρ and does not change any state at p, the states resulting from the execution of the predecessors of s are equal to the states reaching its successor at $p + 1$. That is, s does not introduce any changes to the states which can flow forward and can not change the states reaching program point p'. It means that the removal of s does not change the states at p', and hence, the relevancy of s'.

So after applying f_ρ, the resulting semantics-based program P_{rel} contains all relevant statements *w.r.t.* ρ. The application of S_c on P_{rel} results into a sliced program P_c which, in turn, also contains only the relevant statements *w.r.t.* ρ.

11 Soundness and Complexity Analysis

In this section, we prove that the abstract semantic relevancy computation is sound, and we perform the complexity analysis of the proposed slicing technique.

11.1 Semantic Relevancy: Soundness

When we lift the semantics-based program slicing from the concrete domain to an abstract domain, we are loosing some information about the states occurring at different program points in P. Thus, some relevant statements at the concrete level may be treated as irrelevant in an abstract domain as they do not have any impact on the property observed through the abstract domains.

In order to prove the soundness of the abstract semantic relevancy of statements, we need to show that if any statement s at program point p in the program P is irrelevant *w.r.t.* an abstract property ρ, then the execution of s

over all the concrete states possibly reaching p does not change the property ρ of the variables in those concrete states.

Theorem 11.1 (Soundness) *If a statement s at program point p in the program P is semantically irrelevant w.r.t. an abstract property ρ, then s is semantically irrelevant with respect to the concrete property ω defined by: $\omega \triangleq \forall \sigma \in \Sigma_p,\ \forall x_i \in VAR\ :\ \rho(\sigma[x_i]) = \rho((S[\![s]\!]\sigma)[x_i])$.*

Proof *Given an abstract domain ρ on values, the set of abstract states is denoted by Σ^ρ whose elements are tuples $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle$ where $v_i = \sigma(x_i)$ for $x_i \in VAR$ being the set of program variables.*

Let $\sigma = \langle v_1, \ldots, v_k \rangle \in \Sigma$ and $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho$. Observe that since $\forall x_i \in VAR : \sigma(x_i)$ is a singleton and ρ is partitioning, each variable x_i will have the atomic property obtained from the induced partition $\Pi(\rho)$ [30]. The concretization of the abstract state ϵ is represented by $\gamma(\epsilon) = \{\langle u_1, \ldots, u_k \rangle \mid \forall i.\ u_i \in \rho(v_i)\}$. We denote the j^{th} concrete state in $\gamma(\epsilon)$ by the notation $\langle u_1, \ldots, u_k \rangle^j$ and we denote by u_i^j the elements of that tuple.

As $S[\![s]\!]^\rho(\epsilon)$ is defined as the best correct approximation of $S[\![s]\!]$ on the concrete states in $\gamma(\epsilon)$, we get:

$$
\begin{aligned}
S[\![s]\!]^\rho(\epsilon) &= \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ S[\![s]\!] \langle u_1, \ldots, u_k \rangle^j \} \Big) \\
&= \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ \langle u_1', \ldots, u_k' \rangle^j \mid S[\![s]\!] \langle u_1, \ldots, u_n \rangle^j = \langle u_1', \ldots, u_k' \rangle^j \} \Big) \\
&= \langle\ \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_1'^j \} \Big), \ldots, \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_k'^j \} \Big)\ \rangle
\end{aligned}
$$

where $u_i'^j$ denotes the concrete value of the i^{th} variable $x_i \in VAR$ in the state obtained after the execution of the statement s over the j^{th} concrete state in $\gamma(\epsilon)$. Observe that the later equality relies on the distributivity of ρ, that comes from the assumption of the atomicity of abstract domain obtained from induced partitioning.

From the definition of abstract irrelevancy of a statement s at program point p w.r.t. abstract property ρ, we get

$$\forall \epsilon \in \Sigma_p^\rho :\ \ S[\![s]\!]^\rho(\epsilon) = \epsilon$$

Therefore,

$$S[\![s]\!]^\rho(\epsilon) = \langle\ \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_1'^j \} \Big), \ldots, \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_k'^j \} \Big)\ \rangle = \epsilon$$

Then, by def. of ϵ,

$$\langle\ \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_1'^j \} \Big), \ldots, \rho\Big(\bigcup_{j \in [1..|\gamma(\epsilon)|]} \{ u_k'^j \} \Big)\ \rangle = \langle\ \rho(v_1), \ldots, \rho(v_k)\ \rangle \qquad (1)$$

And so, by def. of $\gamma(\epsilon)$ we get:

$$\langle\, \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_1'^j\}), \ldots, \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_k'^j\})\,\rangle = \langle\, \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_1^j\}), \ldots, \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_k^j\})\,\rangle \tag{2}$$

We already mentioned that given an abstract property ρ, since $\forall x_i \in VAR : \sigma(x_i)$ is a singleton and ρ is a partitioning, each variable x_i will have the property obtained from the induced partition $\Pi(\rho)$ [30]. Thus, $\forall x_i \in VAR :\ \rho(\sigma(x_i)) = \rho(v_i)$ is atomic.

Therefore, from the Equations 1 and 2, we get

$$\forall x_i \in VAR,\ \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_i'^j\}) = \rho(v_i) = \rho(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u_i^j\})\ \textit{is atomic.}$$

This allows us to conclude that for each i^{th} program variables $x_i \in VAR$ (where $i \in [1...k]$) in all the j^{th} concrete states (where $j \in [1..|\gamma(\epsilon)|]$), the concrete value $u_i'^j$ which is obtained after the execution of s over those concrete states have the same property as the concrete value u_i^j before the execution of s. This means that for any irrelevant statement s at program point p in P w.r.t. abstract property ρ, the execution of s over the concrete states possibly reaching p does not lead to any change of the property ρ of the concrete values of the program variables $x_i \in VAR$ in those concrete states.

Thus, s is semantically irrelevant w.r.t. the concrete property $w \triangleq \forall \sigma \in \Sigma_p,\ \forall x_i \in VAR\ :\ \rho(\sigma[x_i]) = \rho((S[\![s]\!]\sigma)[x_i])$.

11.2 Complexity Analysis

Given an abstract domain ρ, our proposal has the following four subsequence steps to obtain abstract slice *w.r.t.* a slicing criteria C:

1. Compute semantic relevancy of program statements *w.r.t.* ρ.
2. Obtain semantic data dependency of each expression on the variables appearing in it *w.r.t.* ρ.
3. Generation of semantics-based abstract DCG by removing all the unrealizable dependences *w.r.t.* ρ.
4. Finally, slice the semantics-based abstract DCG *w.r.t.* C.

11.2.1 Complexity in Computing Statements Semantic Relevancy

Given an abstract domain ρ. To compute semantic relevancy of a statement s at program point p *w.r.t.* ρ, we compare each abstract state $\epsilon \in \Sigma_p^\rho$ possibly reaching p with the state $\epsilon' = S[\![s]\!]^\rho(\epsilon)$. For all state $\epsilon \in \Sigma_p^\rho$, if $\epsilon = \epsilon'$, we say s is irrelevant *w.r.t.* ρ.

To obtain all possible abstract states reaching each program point in a program, we compute its abstract collecting semantics by using the following abstract monotone function

$$F^\sharp(Cx_1^\sharp, \ldots, Cx_n^\sharp) = (F_1^\sharp(Cx_1^\sharp, \ldots, Cx_n^\sharp), \ldots, F_n^\sharp(Cx_1^\sharp, \ldots, Cx_n^\sharp))$$

The least fix point of $F^\sharp$ *i.e.* $FIX^\sharp(F^\sharp)$ gives the abstract collecting semantics for the program that helps in obtaining all possible abstract states reaching each program point.

Complexity to compute abstract collecting semantics. The abstract function $F^\sharp$ involves n monotone functions $F_i^\sharp$, $i = 1, \ldots, n$. The time complexity of each $F_i^\sharp$ depends on the no. of predecessors of s_i and the execution time of $S[\![.]\!]^\rho$, assuming the no. of possible abstract states appearing at each program point as constant. For "skip" statement, $S[\![skip]\!]^\rho$ is constant, whereas for assignment/conditional/repetitive statements, it depends on the execution time for arithmetic and boolean expressions occurred in those statements. Theoretically, there is no limit of the length of expressions *i.e.* the no. of variables/constants/operations present in the expressions. However, practically, we assume that β is the maximum no. of operations (arithmetic or boolean) that can be present in any expression. Assuming the time needed to perform each operation as constant, we get the time complexity of $S[\![.]\!]^\rho$ as $O(\beta)$. Since in a control flow graph the no. of predecessors of each s_i is constant, and $F^\sharp$ involves n monotone functions, the time complexity of $F^\sharp$ is $O(n\beta)$, where n is the no. of statements in the program.

The least solution for $F^\sharp$ depends on the number of iteration performed to reach the fix-point. In case of finite height lattice, let h be the height of the context-lattice $L = (\wp(\Sigma^\rho), \sqsubseteq, \sqcap, \sqcup, \top, \bot)$. The height of L^n is, thus, nh which bounds the number of iteration we perform to reach the fix-point. So the time complexity for $Fix(F^\sharp)$ is $O(n^2\beta h)$.

However, for the lattice with infinite height, a widening operation is mandatory [9] and the overall complexity of $Fix(F^\sharp)$ depends on it.

Complexity to compute statements semantic relevancy. Once we obtain the collecting semantics for a program P in an abstract domain ρ, the time complexity to compute semantic relevancy of each statement depends only on the comparison between the abstract states in the contexts associated with it and in the corresponding contexts of its successors. Any change in the abstract states determines its relevancy *w.r.t.* ρ. For a program with n statements, the time complexity to compute semantic relevancy is, thus, $O(n)$.

11.2.2 Complexity in computing Semantic Data Dependency

Mastroeni and Zanardini [30] introduced an algorithm to compute semantic data dependency of an expression on the variables appearing in it. Before

discussing the complexity, we briefly mention the algorithm.

Given an expression e and an abstract state ϵ, the atomicity condition $A_e^U(\epsilon)$ holds iff execution of e over ϵ *i.e.* $E[\![e]\!]^\rho(\epsilon)$ results an atomic abstract value U, or there exists a covering $\{\epsilon_1, \ldots, \epsilon_k\}$ of ϵ such that $A_e^U(\epsilon_i)$ holds for every i.

In order to compute semantic data dependency of an expression e on the variables $var(e)$ appearing in it, the algorithm calls a recursive function with $X = var(e)$ as parameter. The recursive function uses an assertion $A'_e(\epsilon, X)$, where ϵ is an abstract state possibly reaching the statement containing the expression e. The assertion $A'_e(\epsilon, X)$ holds iff $\exists U : A_e^U(\epsilon)$, or there exists an X-covering $\{\epsilon_1, \ldots, \epsilon_k\}$ of ϵ such that $\forall i : A'_e(\epsilon_i, X)$. Intuitively, X-covering is a set of restriction on a state, which do not involve X. If $A'_e(\epsilon, X)$ holds, it implies the non-relevance of X in the computation of e, otherwise for each $x \in X$ the same is repeated with $X \backslash x$ as parameter.

Thus, the time complexity to compute semantic data dependency at expression level for the whole program depends on the following factors:

- The time complexity of $E[\![e]\!]^\rho$: Theoretically there is no limit of the length of expression e *i.e.* the no. of variables/constants/operations present in e. However, practically, we assume that β is the maximum no. of operations (arithmetic or boolean) that can be present in e. Assuming the time needed to perform each operation as constant, we get the time complexity of $E[\![e]\!]^\rho$ as $O(\beta)$.

- The time complexity of the atomicity condition $A_e^U(\epsilon)$: In worst case, the time complexity of $A_e^U(\epsilon)$ depends on the time complexity of $E[\![e]\!]^\rho$ and the no. of elements in the covering of ϵ. Let m be the no. of atomic values in the abstract domain ρ. Since the no. of elements in a covering depends on the no. of atomic values in the abstract domain, the time complexity of $A_e^U(\epsilon)$ is $O(m\beta)$.

- The time complexity of the assertion $A'_e(\epsilon, X)$: In worst case, the time complexity of $A'_e(\epsilon, X)$ depends on the time complexity of atomicity condition $A_e^U(\epsilon)$ and the no. of elements in the X-covering of ϵ. Thus, the time complexity of $A'_e(\epsilon, X)$ is $O(m^2\beta)$.

In worst case, the recursive function that uses $A'_e(\epsilon, X)$ executes for all subset of variables appearing in e *i.e.* $\forall X \in \wp(var(e))$. So, it depends on the set of program variables VAR. Therefore, the time-complexity of the recursive function is $O(m^2\beta\text{VAR})$, where VAR is the set of program variables. For a program P of size n, the no. of expressions that can occur in worst case is n. Thus, finally we get the time complexity to compute semantic data dependency for a program P of size n is $O(m^2\beta n\text{VAR})$.

11.2.3 Complexity to generate Semantics-based Abstract DCG and slicing based on it

Given a program P (in IMP language) and its PDG, the time complexity to construct DCG from a PDG is $O(n)$ [39], where n is the no. of nodes in the PDG.

However, to obtain semantics-based abstract DCG G^s_{dcg} from a syntactic DCG G_{dcg}, our algorithm removes all the unrealizable dependences present in G_{dcg}.

To do this, the algorithm checks the satisfiability of the annotations of all the outgoing DDG edges, incoming CDG edge and outgoing ϕ-sequences associated with each node in the DCG against all the abstract execution traces.

For a DCG with n nodes, the maximum no. of edges need to check is $O(n^2)$. Thus, in case of lattice of finite height h, the worst case time complexity to verify all dependences for their satisfiability against abstract execution traces is $O(n^3h)$.

As we know that the slicing which is performed by walking a DCG backwards or forwards from the node of interest takes $O(n)$ [33], the worst case time complexity to obtain DCG-based slicing is, therefore, $O(n^3h)$.

11.2.4 Overall Complexity of the Proposal

Let us consider that the maximum number of operations (β) presented in any expression and the number of atomic values (m) present in the abstract domain are constants, and $O(\text{VAR}) = O(n)$.

As an overall complexity evaluation of the techniques presented so far, we can say that it has worst case time complexity, in case of finite height abstract lattices, is $O(n^3h))$, where n is the no. of statements in the program and h is the height of the lattice of context.

12 Discussions and Conclusions

We acknowledge that there are other possible improvements that deserve to be considered as a future works. As for instance, the semantic relevancy at statement-level does not take into account the semantic interaction between statements. For example, if consider a block consisting of two statements $\{y = y + 3;\ y = y - 1;\}$, we observe that each of the two statements is not semantically irrelevant *w.r.t. PAR*, while the block as a whole is irrelevant *w.r.t. PAR*.

Therefore, to be more precise, we should start to compute the relevancy of a program from block-level to statement-level. If any block is irrelevant, we disregard all statements in that block; otherwise, we compute relevancy for all sub-blocks of that block. In this way, we compute the relevancy by moving from block-level towards the statement-level. Instead, we can also use the partial evaluation technique, although costlier, to resolve this issue. For instance, the above two statements can be replaced by a single statement y=y+2 which is irrelevant *w.r.t. PAR*.

In [30], the problem related to the control dependency is not addressed. For example, consider the following example:

```
4.    ....
5.    if ((y + 2x mod 2) == 0) then
6.          w=5;
7.    else w=5;
8.    ....
```

Here the abstract semantic data dependency says that the condition in "if" statement is only dependent on y. But it does not say anything about the dependency between w and y.

Observe that although w is invariant *w.r.t.* the evaluation of the guard, this is not captured by [30].

The block-level semantic relevancy, rather than statement-level, can resolve this issue of independency. Let us denote the complete "if-*else*" block by s. The semantics of s says that $\forall\sigma_1, \sigma_2 \in \Sigma_5$, $S[\![s]\!](\sigma_1) = \sigma_1'$ and $S[\![s]\!](\sigma_2) = \sigma_2'$ implies $\sigma_1' = \sigma_2'$, where $\sigma_1'(w) = \sigma_2'(w) = 5$. It means that there is no control of the "if-*else*" over the resultant state which is invariant. So we can replace the whole conditional block s by the single statement $w = 5$. Notice that this is also true if we replace the statement at line 6 by w=y+5, as the line 6 is executed when y==0.

The combined result of semantic relevancy, semantic data dependency and conditional dependency in the refinement of the PDGs can be applied to all forms of slicing: static, dynamic, conditional, amorphous etc. Since the allowed initial states are different for different forms of slicing, we compute statements relevancy and semantic data dependency of expressions over all the possible states reaching the program points in the program by starting only from the allowed initial states, according to the criteria. Similarly, the satisfiability of the depenedence paths in the DCG are checked against the traces generated from the allowed initial states only. For instance, in case of conditional slicing, a condition is specified in the slicing criteria to disregard some initial states that do not satisfy it. In case of dynamic slicing, inputs of interest are specified in the slicing criteria. Therefore, the collecting semantics and execution traces are generated based on the allowed initial states specified or satisfying the conditions in the criteria, and are used to compute statements relevancy, semantic data dependency and satisfiability of the dependence paths.

The combination of results on the refinement of dependence graphs with static analysis techniques discussed in this report may give rise to further interesting applications to enhance the accuracy of the static analysis and for accelerating the convergence of the fixed-point computation. This is the topic of our ongoing research.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)*, pages 246–256, White Plains, New York, June 1990. ACM Press.

[2] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[3] S. Bhattacharya. *Property Driven Program Slicing and Watermarking in the Abstract Interpretation Framework*. PhD thesis, Università Ca' Foscari Venezia, 2011.

[4] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel. A formalisation of the relationship between forms of program slicing. *Science of Computer Programming*, 62(3):228–252, 2006.

[5] Robert Cartwright and Mattias Felleisen. The semantics of program dependence. *ACM SIGPLAN Notices*, 24(7):13–27, 1989.

[6] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven programslicing: a case study. In *Proceedings of the 11th International Conference on Software Maintenance (ICSM '95)*, pages 124–133, Opio (Nice), France, October 1995. IEEE Computer Society.

[7] A. Cortesi and S. Bhattacharya. A framework for property-driven program slicing. In *Proceedings of the 1st International Conference on Computer, Communication, Control and Information Technology*, pages 118–122, Kolkata, India, March 2009. Macmillan Publishers India Ltd.

[8] Agostino Cortesi and Raju Halder. Dependence condition graph for semantics-based abstract program slicing. In *Proceedings of the 10th International Workshop on Language Descriptions Tools and Applications (LDTA '10)*, Paphos, Cyprus, March 2010. ACM Press.

[9] Agostino Cortesi and Matteo Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

[10] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, 1977. ACM Press.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[12] Ran Ettinger. *Refactoring via Program Slicing and Sliding*. PhD thesis, Oxford University Computing Laboratory, 2006.

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

[14] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[15] R. Gerber and S. Hong. Slicing real-time programs for enhanced schedulability. *ACM Trans on Programming Languages and Systems*, 19(3):525–555, May 1997.

[16] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[17] D. Goswami and Rajib Mall. An efficient method for computing dynamic program slices. *Information Processing Letters*, 81(2):111–117, 2002.

[18] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of International Conference on Software Maintenance (ICSM '92)*, pages 299–308, Orlando, FL, USA, November 1992. IEEE Computer Society.

[19] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, 2003.

[20] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. *ACM SIGPLAN Notices*, 33:97–105, 1998.

[21] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*, pages 146–157, San Diego, California, United States, January 1988. ACM Press.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[23] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. *ACM SIGSOFT Software Engineering Notes*, 19(5):2–10, 1994.

[24] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[25] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. In *Proceedings of the 5th International Workshop on Program Comprehension (WPC '97)*, pages 80–89, Dearborn, MI, USA, May 1997. IEEE Computer Society.

[26] Jens Krinke. Static slicing of threaded programs. *ACM SIGPLAN Notices*, 33(7):35–42, 1998.

[27] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '81)*, pages 207–218, Williamsburg, Virginia, January 1981. ACM Press.

[28] Filippo Lanubile and Giuseppe Visaggio. Extracting reusable functions by flow graph-based program slicing. *IEEE Transactions on Software Engineering*, 23(4):246–259, 1997.

[29] Isabella Mastroeni and Durica Nikolic. Abstract program slicing: From theory towards an implementation. In *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM '10)*, pages 452–467, Shanghai, China, November 2010. Springer LNCS, Volume 6447.

[30] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM '08)*, pages 125–134, San Francisco, California, USA, January 2008. ACM Press.

[31] Markus Muller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *Proceedings of the 33rd annual ACM symposium on Theory of computing (STOC '01)*, pages 647 – 656, Hersonissos, Greece, July 2001. ACM press.

[32] G. B. Mund and Rajib Mall. An efficient interprocedural dynamic slicing method. *The Journal of Systems and Software*, 79(6):791–806, 2006.

[33] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5):177–184, 1984.

[34] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[35] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '95)*, pages 41–52, Washington, DC, USA, October 1995. ACM Press.

[36] Vivek Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(5–6):779–804, 1991.

[37] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '05)*, pages 25–34, Budapest, Hungary, 30 Sept.–1 Oct. 2005. IEEE Computer Society.

[38] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 432–441, Los Angeles, CA, USA, May 1999. ACM Press.

[39] S. Sukumarana, A. Sreenivasb, and R. Metta. The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems & Structures*, 36:96–121, 2010.

[40] Reps Thomas and Yang Wuu. The semantics of program slicing. Technical report, University of Wisconsin, 1988.

[41] G. A. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107–119, 1991.

[42] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[43] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.