Science of Computer Programming 78 (2013) 1240–1263

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Abstract program slicing on dependence condition graphs

Raju Halder, Agostino Cortesi [*]

Università Ca' Foscari Venezia, Italy

ARTICLE INFO

Article history:
Received 19 February 2011
Received in revised form 9 May 2012
Accepted 16 May 2012
Available online 12 June 2012

Keywords:
Program slicing
Dependence graph
Abstract interpretation

ABSTRACT

Context: Mastroeni and Zanardini introduced the notion of semantics-based data dependences, both at concrete and abstract domains, that helps in converting the traditional syntactic Program Dependence Graphs (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 in more precise slices. *Aim*: The aim of this paper is to further refine the slicing algorithms when focusing on a given property. *Method*: The improvement is obtained by (i) applying the notions of semantic relevancy of statements and semantic data dependences, and (ii) combining them with conditional dependences. *Result*: We provide an abstract slicing algorithm based on semantics-based abstract Dependence Condition Graphs (DCGs) that enable us to identify the conditions for dependences between program points.

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. The notion of a program slice was originally introduced by Mark Weiser [40] who defines a static program slice as an executable subset of program statements that preserves the original program's behavior at a particular program point for a subset of program variables for all program inputs. Therefore, the static slicing criterion 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 [23], 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, the slicing criterion 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 the abstract domain, 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 [10,14]. The notion of Abstract Program Slicing was first introduced by Hong, Lee and Sokolsky [35]. Some recent works include semantics-based abstract program slicing [27,28,42] and property-driven program slicing [4]. Abstract slicing helps in finding all statements affecting some particular properties of the variables of interest. For instance, suppose a program variable at some point of execution is not resulting in the correct properties (a positive value, say) as desired. In such a case, abstract slicing can effectively identify the statements responsible for this error. Recently, Mastroeni and Nicolić [27] extended the theoretical framework of slicing proposed by Binkley [5] to an abstract domain in order to define abstract

[*] Corresponding author. Tel.: +39 0412348450; fax: +39 0412348419.
E-mail addresses: halder@unive.it (R. Halder), cortesi@unive.it (A. Cortesi).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.05.007

program slicing, and to represent and compare different forms of program slicing in the abstract domain. Slicing criteria in an abstract domain, thus, include observable properties of the 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 an observable property of v.

All the techniques mentioned so far make use (explicitly or implicitly) of the notion of a Program Dependence Graph (PDG) [13,24,31]. However, different forms of PDG representation have been proposed, depending on the intended applications [1,21,22]. Over the last 25 years, many program slicing techniques have been proposed based on the PDG representation or based on its variants like System Dependence Graph (SDG) or Dynamic Dependence Graph (DDG) representations [1,22,34,36]. 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.

This paper provides two main contributions in this area. The first one is the use 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 dependences introduced by Mastroeni and Zanardini [28] guarantees a more precise semantics-based (abstract) PDG.

The second contribution of the paper is the refinement of the semantics-based PDGs obtained so far by applying the notion of conditional dependences proposed by Sukumaran et al. [37]. This allows us to transform PDGs into Dependence Condition Graphs (DCGs) that enable us to identify the conditions for dependences between program points. We lift the semantics of DCGs from concrete domain to an abstract domain. 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 dependences [28] 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 dependences at expression level [28], and (iii) conditional dependences based on DCG annotations [37].

The rest of the paper[1] is organized as follows: Section 2 recalls some basic background. Section 3 introduces the notion of semantic relevancy of statements *w.r.t.* concrete/abstract property. In Section 4, we formalize an algorithm to construct a semantics-based abstract PDG from a given program. In Section 5, we lift the semantics of DCGs from a concrete to an abstract domain, and we propose a refinement of syntactic DCCs into semantics-based abstract DCGs. The proposed abstract slicing algorithm is formalized in Section 6. Section 7 illustrates the proposed slicing technique with an example. In Section 8, we prove the soundness and provide an overall complexity analysis of the proposal. Section 9 presents an overall architecture of a tool implementing our abstract program slicing algorithm. In Section 10, we discuss related works in the literature. Finally, Section 11 concludes the paper.

2. Preliminaries

In this section we recall some basic backgrounds.

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. Fig. 1(a) and (b) depict a program P and its SSA form P_{ssa} respectively. In the rest of the paper, we use the 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].

Program dependence graph. Program Dependence Graph (PDG) [13,24,31] 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 a control dependence graph (CDG) and the sub-graph induced by the data dependence edges is called a data dependence graph (DDG). The source node of a CDG edge corresponds to either *start* or

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

```
1.    start
2.    $x = input;$
3.    $y = input;$
4.    $w = input;$
5.    $z = 4;$
6.    $if\,(x == y)\{$
7.        $z = 2 \times (x + y) - 4 \times (x) + 4;$
8.        $x = x + 1;\ \}$
9.    $else\ \{\ z = x + w;$
10.       $x = x + 2;\ \}$
11.   $print\,(x, z);$
12.   stop
```

(a) *Program P*.

```
1.    start
2.    $x_1 = input;$
3.    $y_1 = input;$
4.    $w = input;$
5.    $z_1 = 4;$
6.    $if\,(x_1 == y_1)\{$
7.        $z_2 = 2 \times (x_1 + y_1) - 4 \times (x_1) + 4;$
8.        $x_2 = x_1 + 1;\ \}$
9.    $else\ \{\ z_3 = x_1 + w;$
10.       $x_3 = x_1 + 2;\ \}$
$\phi$    $(x_4, z_4) = f((x_2, z_2), (x_3, z_3));$
11.   $print\,(x_4, z_4);$
12.   stop
```

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

Fig. 1. Program P and its SSA form P_{ssa}.

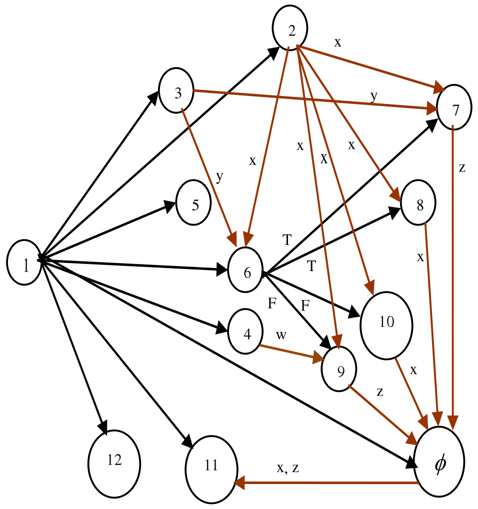


Fig. 2. G_{pdg}: PDG of P_{ssa}.

conditional or *repetitive* statement. A CDG edge e whose source node $e.src$ corresponds to the *start* statement is denoted by an unlabeled edge $e = e.src \longrightarrow e.tgt$, meaning that the condition represented by $e.src$ is implicitly *true*, i.e., during an execution once $e.src$ executes its target $e.tgt$ will eventually be executed. If the source node $e.src$ of any CDG edge e corresponds to a *conditional* or *repetitive* statement it is denoted by a labeled edge $e = e.src \xrightarrow{lab} e.tgt$ where $lab \in \{true, false\}$, meaning that $e.tgt$ is *lab*-control dependent on $e.src$, i.e., 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} (Fig. 1(b)) is depicted in Fig. 2.

PDG-based slicing. The results of the program dependence graph discussed so far have an impact on different forms of static slicing: backward slicing [40], forward slicing [3], and chopping [32]. 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 the program dependence graph is slightly restrictive in the sense that the dependence 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.* an arbitrary variable at p. PDG-based backward program slicing is performed by walking the graph backwards from the node of interest in linear time [31]. The walk terminates at either entry node or already visited node. In case of the PDG-based forward slicing technique, similarly, we traverse the graph in the forward direction 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 the slicing technique [32]: $chop(s, t)$ 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 [37]: $AC(s, t)$ 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 [1,29] *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 the static slicing algorithm on the projected dependence graph is applied to find the desired dynamic slice.

In general, there exist many different slices for a given program *w.r.t.* a slicing criterion where one can be more precise than the other, as defined in Definition 1.

Definition 1 (*Precision of Slicing*). Given two programs P' and P'' such that both P' and P'' are slices of P *w.r.t.* a criterion C. P' is more precise than P'' if P' $(\neq P'')$ is a slice of P'' *w.r.t.* C.

Abstract interpretation and program slicing. Abstract Interpretation, originally introduced by Cousot and Cousot is a well known semantics-based static analysis technique [10,14]. Its main idea is to relate concrete and abstract semantics where the latter are focusing 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.

A first attempt to combine Abstract Interpretation with program slicing is done by Mastroeni and Zanardini in [28]. In traditional PDGs, the notion of dependences between statements is based on syntactic presence of a variable in the definition of another variable or in a conditional expression. Therefore, the definition of slices at the semantic level creates a gap between slicing and dependences. Mastroeni and Zanardini [28] introduced the notion of semantic data dependences which fills up the existing gap between syntax and semantics. For instance, although the expression "$e = x^2 + 4w \bmod 2 + z$" syntactically depends on w, but semantically there is no dependence as the evaluation of "$4w \bmod 2$" is always zero. This can also be lifted to an abstract setting where dependences are computed with respect to some specific properties of interest rather than concrete values. For instance, if we consider the abstract domain $SIGN = \{\top, pos, neg, \bot\}$, the expression e does not semantically depend on x *w.r.t.* *SIGN*, as the abstract evaluation of x^2 always yields to *pos* for all atomic values of $x \in \{pos, neg\}$. 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.

Abstract semantics: Expressions and statements. Consider the IMP language [41]. 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 $v_i \in \mathbb{V}$, $i = 1, \ldots, k$ 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

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

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 the 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 $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho : E[\![e]\!]^\rho(\epsilon) = \rho(\{E[\![e]\!](\langle u_1, \ldots, u_k \rangle) \mid \forall i.\ u_i \in \rho(v_i)\})$.

3. Semantic relevancy of statements

In this section, we introduce the notion of semantic relevancy of statements in both concrete and abstract domains. The motive behind this is to determine whether the execution of a statement affects a concrete/abstract property of the variables of interest.

Definition 2 (*Concrete Semantic Relevancy*). Given a program P and a concrete property ω on states, the statement s at program 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 Definition 3.

Definition 3 (*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 1. Consider the concrete property $\omega_{(x\neq 4)} : \Sigma \rightarrow \{true, false\}$, 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 2. Consider the program P_{ssa} in Fig. 1(b). 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.

Example 3. Consider the program P_{ssa} in Fig. 1(b) 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 $\{y \in Z : y \textit{ is even}\}$, ODD represents $\{y \in Z : y \textit{ is odd}\}$, 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 $\sigma \in \Sigma_8$ with an equal number of variable values belonging to both ODD and $EVEN$ sets, yields to a state σ' where the value of x moves from one set to another. Observe that the statement at program point 10, on the other hand, is irrelevant *w.r.t.* ω.

As clearly observed by Weiser's thesis [39], semantic relevancy is not computable in general, as it is the case for many non-trivial behavioral properties of programs. We can only approximate the semantic relevancy, and the actual accuracy of slicing algorithms heavily depends on how good this approximation is. Observe that there is always a safe approximation, i.e. the most conservative one.

The notion of semantic relevancy in an abstract domain is defined in Definition 4. The abstract semantic relevancy can be defined in relational as well as non-relational abstract domains.

Definition 4 (*Abstract Semantic Relevancy*)**.** Let P be a program and ρ be a closure operator over $\wp(\mathbb{V})$, 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.

In other words, the statement s at program point p is semantically irrelevant *w.r.t.* an abstract property ρ if no changes take place in the abstract states $\epsilon \in \Sigma_p^\rho$, when s is executed over ϵ.

Example 4. Consider the statement $s \triangleq x = x+2$ at program point 10 of the program P_{ssa} depicted in Fig. 1(b). 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.

Definition 4 is not parametric on variables. Below we provide a parametric definition for the abstract statement relevancy. This definition may be useful, combined with independence analysis, to further refine the slicing when focusing just on a proper subset of program variables in the slicing criterion.

Definition 5 (*Abstract Semantic Relevancy for a Set of Variables*)**.** Let P be a program and ρ be a closure operator over $\wp(\mathbb{V})$, 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

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

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

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 4 is just a rephrasing of Definition 2.

In a similar way, we can lift the notion of semantic relevancy to blocks and to control statements also (for details, see [16]).

The well-known collecting semantics (also known as static semantics) [10] of programs can help in obtaining the set of states possibly reaching each program point in the program. The relevancy computation of a statement s at p *w.r.t.* a concrete/abstract property is, thus, performed by simply checking whether the execution of s changing the property of any of the states at p generated from the collecting semantics. Note that the only statements that can affect the property of states are the "*assignment*" statements of the form $v = e$, where v is a variable and e is an arithmetic expression.

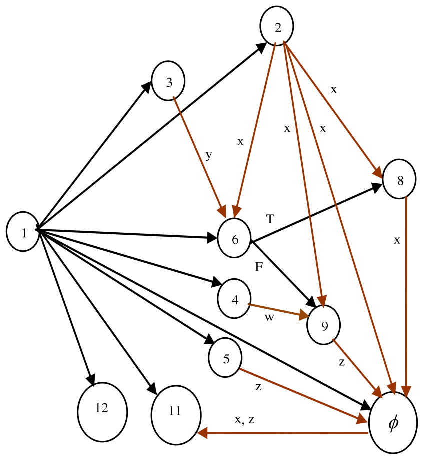


Fig. 3. G^r_{pdg}: *Semantics-based abstract PDG of* P_{ssa} *w.r.t. PAR.*

The notion of statement relevancy has many interesting application areas. For instance, if we are analyzing a speed control engine and we are just interested in the portion of the 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 either directly or indirectly its sign can immediately be disregarded.

Given a program and its syntactic PDG, we can obtain a more precise semantics-based (abstract) PDG by disregarding from the syntactic PDG all the nodes that corresponds to irrelevant statements *w.r.t.* a concrete/abstract property of interest. For instance, Fig. 3 depicts the semantics-based abstract PDG G^r_{pdg} which is obtained by disregarding from G_{pdg} in Fig. 2 two nodes corresponding to the statements 7 and 10 which are irrelevant *w.r.t. PAR*.

It is worthwhile to mention that the computation above can be optimized in particular cases by applying slicing on the syntactic PDG first, and then, computing statement relevancy and semantic data dependences on this sliced program.

4. 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 Fig. 4. In this proposed algorithm, we combine (i) the notion of semantic relevancy of statements, and (ii) the notion of semantic data dependences of expressions.

We use the notation ϵ_{ij} to denote the jth 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 dependences for all expressions in P_{rel} by following the algorithm of Mastroeni and Zanardini [28]. 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.

5. 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 [37]. A DCG 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. The first component e^R refers to *Reach Sequences*, whereas the second component 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

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 $\epsilon_{ij}$ and determine (whenever it is possible) if it can be considered semantically irrelevant;
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. $\rho$, 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. $\rho$, 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. $\rho$:
                        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. $\rho$:
                        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. $\rho$:
                        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 dependences for all expressions in $P_{rel}$ w.r.t. $\rho$ 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. $\rho$, as obtained in previous steps;
```

Fig. 4. Algorithm to generate semantics-based abstract PDG.

e.tgt. It is worthwhile to note that e^A is relevant only for DDG edges and it is $\emptyset$ for CDG edges. Example 5 illustrates briefly how to compute annotations over the edges in a PDG.

Example 5. Consider the edge $2 \xrightarrow{x} 8$ in the semantics-based abstract PDG of Fig. 3. For this edge, we see that node 8 does not post-dominate the node 2. Thus, the reach sequence for the edge is $(2 \xrightarrow{x} 8)^R = \{1 \xrightarrow{true} 6 \xrightarrow{true} 8\}$. This means that the condition at 6 must be *true* in the execution trace to ensure that both 2 and 8 are executed, and the data x assigned at 2 can reach 8. For the edge $2 \xrightarrow{x} \phi$, the reach sequence $(2 \xrightarrow{x} \phi)^R$ is empty, meaning that no condition has to satisfy for the x assigned at 2 to reach ϕ, because ϕ post-dominates 2 and once 2 is executed ϕ is also executed. To compute the avoid sequences for the edge $2 \xrightarrow{x} \phi$, we consider two data dependence edges $e_1 = 2 \xrightarrow{x} \phi$ and $e_2 = 8 \xrightarrow{x} \phi$ with ϕ as target. By following the algorithm of [37], we get the avoid sequences $(2 \xrightarrow{x} \phi)^A = \{1 \xrightarrow{true} 6 \xrightarrow{true} 8\}$. This reflects the fact that the "*if*" condition at 6 must be *false* in order to guarantee that the definition of x at 2 is not re-defined at 8 and can reach ϕ. Table 1 depicts the DCG annotations for all DDG edges of the semantics-based abstract PDG in Fig. 3.

Sukumaran et al. [37] described the semantics of DCG annotations in terms of execution semantics of the program over the concrete domain.

Below, we first define the abstract semantics of DCG annotations in an abstract domain. 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 $(p_i, \epsilon_{p_i}),\ i = 1, 2, 3, \ldots$ 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".

Table 1
DCG annotations $\langle e^R, e^A\rangle$ for DDG edges e of G^r_{pdg} in Fig. 3.

e	e^R	e^A
$2 \xrightarrow{x} 6$	$\emptyset$	$\emptyset$
$3 \xrightarrow{y} 6$	$\emptyset$	$\emptyset$
$2 \xrightarrow{x} 8$	$1 \xrightarrow{true} 6 \xrightarrow{true} 8$	$\emptyset$
$2 \xrightarrow{x} 9$	$1 \xrightarrow{true} 6 \xrightarrow{false} 9$	$\emptyset$
$4 \xrightarrow{w} 9$	$1 \xrightarrow{true} 6 \xrightarrow{false} 9$	$\emptyset$
$2 \xrightarrow{x} \phi$	$\emptyset$	$1 \xrightarrow{true} 6 \xrightarrow{true} 8$
$8 \xrightarrow{x} \phi$	$\emptyset$	$\emptyset$
$5 \xrightarrow{z} \phi$	$\emptyset$	$1 \xrightarrow{true} 6 \xrightarrow{false} 9$
$9 \xrightarrow{z} \phi$	$\emptyset$	$\emptyset$
$\phi \xrightarrow{x,z} 11$	$\emptyset$	$\emptyset$

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 6 (*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 7 (*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 8 (*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 $(p_{s_1} \xrightarrow{labs_1} p_{s_2} \xrightarrow{labs_2} \dots p_{s_n} \xrightarrow{labs_n} p_j) \in e^R$ where $p_{s_1}, p_{s_2}, \dots, p_{s_n}$ correspond to conditional statements, the trace ψ contains $\iota_k : (p_{s_k}, \epsilon_{p_{s_k}})$ for $k = 1, \dots, n$ and $\iota_1 < \iota < \iota_2 < \dots < \iota_n$ and $\bigwedge_{1\leq k\leq n}$ $[\![p_{s_k}.cond = lab_{s_k}]\!](\epsilon_{p_{s_k}})$ yields to "true" or "unknown".

Definition 9 (*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 $\psi \vdash^{A}_{\iota} e$) if ψ contains $\iota : (p_i, \epsilon_{p_i})$, and for each $(p_{s_1} \xrightarrow{labs_1} p_{s_2} \xrightarrow{labs_2} \dots p_{s_n} \xrightarrow{labs_n} p') \in e^A$ 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:

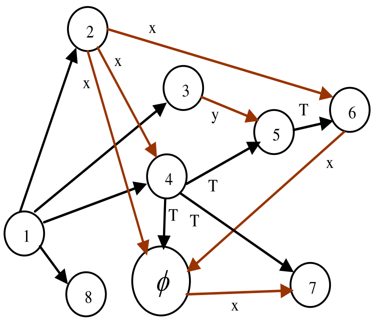
- ψ' does not contain $(p_{s_k}, \epsilon_{p_{s_k}})$ for $1 \leq k \leq n$, or
- $\exists\, k : 1 \leq k \leq n$: ψ' contains $(p_{s_k}, \epsilon_{p_{s_k}})$ such that $[\![p_{s_k}.cond = lab_{s_k}]\!](\epsilon_{p_{s_k}})$ yields to false.

Definition 10 (*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 $\psi \vdash^{\rho}_{\iota} e$) is defined as

$$\psi \vdash^{\rho}_{\iota} e \triangleq (\psi \vdash^{R}_{\iota} e) \wedge (\psi \vdash^{A}_{\iota} e).$$

```
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}.

Fig. 5. The program P_{ssa} and its DCG.

Definition 11 (*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

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

Theorem 5.1. *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. See [16]. □

Example 6. Consider the program P and its PDG depicted in Fig. 5(a) and (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 [37], 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 Fig. 5(c).

Consider the abstract domain *SIGN*. The initial state of P is defined 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

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

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 12 (*PhiSeqs*). A ϕ-sequence η_ϕ is a DDG path of the form: $n_1 \rightarrow \phi_1 \rightarrow \phi_2 \rightarrow \cdots \rightarrow \phi_k \rightarrow n_2$, 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 $\eta = e_1e_2 \dots e_n$ 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 13 (*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 14 (*Execution Satisfying a Dependence Path*). A series of program dependences represented by a dependence path $\eta = e_1e_2 \dots e_n$ is said to be satisfied by an execution ψ over an abstract domain ρ (written as $\psi \vdash^{\rho} \eta$) if

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

Theorem 5.2. *Given a ϕ-sequence $\eta_\phi = e_1e_2 \dots e_n$ 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. See [16]. □

Example 7. Consider the dependence path $\eta = 2 \xrightarrow{x} 6 \xrightarrow{x} \phi \xrightarrow{x} 7$ in the graph of Fig. 5, and the following execution trace over the abstract domain *SIGN*:

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

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 $\iota_1 : \langle 1, (\bot, \bot)\rangle\ \iota_2 : \langle 2, (\bot, \bot)\rangle \dots \iota_4 : \langle 4, (+, +)\rangle\ \iota_5 : \langle 5, (+, +)\rangle\ \iota_6 : \langle 6, (+, +)\rangle \dots$ 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 $\dots \iota_6 : \langle 6, (+, +)\rangle\ \iota_7 : \langle \phi, (+, +)\rangle \dots$.
- $\psi \vdash^{SIGN}_{\iota_7} (\phi \xrightarrow{x} 7)$ 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 $\dots \iota_7 : \langle \phi, (+, +)\rangle\ \iota_8 : \langle 7, (+, +)\rangle \dots$.

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

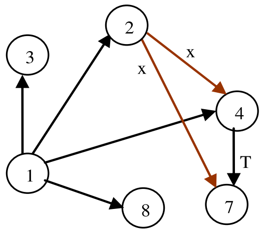
Let η be a dependence path in a DCG. Suppose an execution trace ψ over an abstract domain ρ satisfies η (denoted $\psi \vdash^{\rho} \eta$). If we compute semantic relevancy *w.r.t.* ρ and we disregard the irrelevant entries from both η and ψ, we see that the satisfiability of the refined path is also preserved, as depicted in Theorem 5.3.

Theorem 5.3. *Given a program P and its DCG G_{dcg}. Let ψ be an execution trace of P over an abstract domain ρ, and $\eta = e_1e_2 \dots e_le_{l+1} \dots e_h$ 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 $\eta' = e_1e_2 \dots e_q \dots e_h$, where $e_q : p_i \xrightarrow{x} p_k$, and a trace ψ' respectively. Then,*

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

```
1.    start
2.    x = input;
3.    y = input;
4.    if (x > 0){
7.         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}.

Fig. 6. Program P'_{ssa} and its DCG after relevancy computation.

Proof. See [16]. □

Example 8. Look at Fig. 5 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*:

$$\begin{aligned}\psi \;&= \iota_1 : \langle 1, (\bot, \bot)\rangle\; \iota_2 : \langle 2, (\bot, \bot)\rangle\; \iota_3 : \langle 3, (+, \bot)\rangle\; \iota_4 : \langle 4, (+, +)\rangle\; \iota_5 : \langle 5, (+, +)\rangle \\ &\quad \iota_6 : \langle 6, (+, +)\rangle\; \iota_7 : \langle \phi, (+, +)\rangle\; \iota_8 : \langle 7, (+, +)\rangle\; \iota_9 : \langle 8, (+, +)\rangle.\end{aligned}$$

Note that $\psi \vdash^{SIGN} \eta$, as already shown in Example 7.

Fig. 6(a) and 6(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 Fig. 6(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:

$$\begin{aligned}\psi' \;&= \iota_1 : \langle 1, (\bot, \bot)\rangle\; \iota_2 : \langle 2, (\bot, \bot)\rangle\; \iota_3 : \langle 3, (+, \bot)\rangle\; \iota_4 : \langle 4, (+, +)\rangle\; \iota_8 : \langle 7, (+, +)\rangle \\ &\quad \iota_9 : \langle 8, (+, +)\rangle.\end{aligned}$$

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

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 $1 \xrightarrow{true} 4 \xrightarrow{true} 7 \in (2 \xrightarrow{x} 7)^R$: $[\![1.cond = true]\!](\bot, \bot)$ and $[\![4.cond = true]\!](+, +)$ yields to true. Thus, $\psi' \vdash^R_{\iota_1} (2 \xrightarrow{x} 7)$.

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

5.1. 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 15.

Definition 15 (*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 $\forall \psi$: $\psi \nvdash^{\rho} \eta$, where ψ is an abstract execution trace.

The refinement algorithm of a DCG from syntactic to semantic one is depicted in Fig. 7. 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.

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 $\forall\psi: \psi \nvdash^{\rho} (p \xrightarrow{lab} q)$ 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 $\forall\psi: \psi \nvdash^{\rho} e$ 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 $\phi$-sequences $\eta_\phi = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \dots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ 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
```

Fig. 7. Algorithm to generate semantics-based abstract DCG.

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.

6. Slicing algorithm

We are now in position to formalize our proposed slicing algorithm, depicted in Fig. 8, that takes a program P and an abstract slicing criterion $\langle p, v, \rho\rangle$ 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 4, and then by converting it into semantics-based abstract DCG by following the algorithm REFINE-DCG depicted in Section 5.

Observe that the sub-DCG G_{sdcg} which is obtained in step 4 by applying slicing criterion 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 Fig. 9(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.

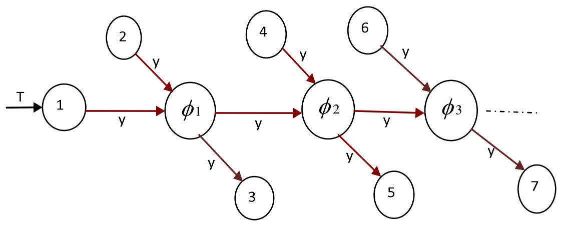
- $\eta_1 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} 3$
- $\eta_2 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_2 \xrightarrow{y} 5$
- $\eta_3 = 1 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_2 \xrightarrow{y} \phi_3 \xrightarrow{y} 7.$

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

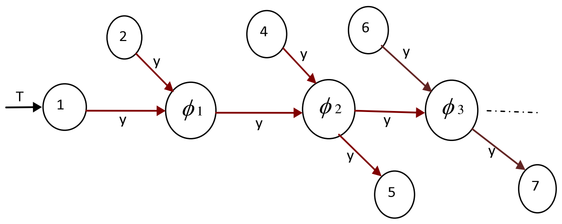
Given a slicing criterion C. In algorithm GEN-SLICE, suppose the sub-DCG generated after applying C does not include the node 3, as depicted in Fig. 9(b). Now if we apply step 5 on the sub-DCG, we see that all the ϕ-sequences emerging from node 1 (ϕ_2 and ϕ_3) are not semantically realizable. Therefore, we can remove the edge $1 \xrightarrow{y} \phi_1$ from it as depicted in Fig. 9(c). The further application of the slicing criterion 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.

```
Algorithm 3: GEN-SLICE
Input: Program P and an abstract slicing criterion ⟨p, v, ρ⟩
Output: Abstract Slice w.r.t. ⟨p, v, ρ⟩
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 criterion $\langle p, v\rangle$ on $G^{s}_{dcg}$ by following PDG-based slicing techniques [31] and generate a sub-DCG $G_{sdcg}$ that includes
   the node corresponding to the program point $p$ as well.
5. Refine $G_{sdcg}$ into a more precise one $G'_{sdcg}$ by performing the following operation for all nodes $q \in G_{sdcg}$:
   $\forall \eta_\phi = (q \xrightarrow{x} \phi_1 \xrightarrow{x} \dots \xrightarrow{x} \phi_j \xrightarrow{x} p_i)$ 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 criterion $\langle p, v\rangle$ on $G'_{sdcg}$ that results in the desired slice.
```

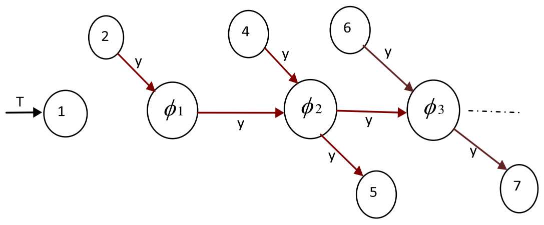
Fig. 8. Slicing algorithm.



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



(b) Sub-DCG after applying slicing criterion C.



(c) Refinement of Sub-DCG.

Fig. 9. Refinement of sub-DCG during slicing.

7. Illustration of the proposal with an example

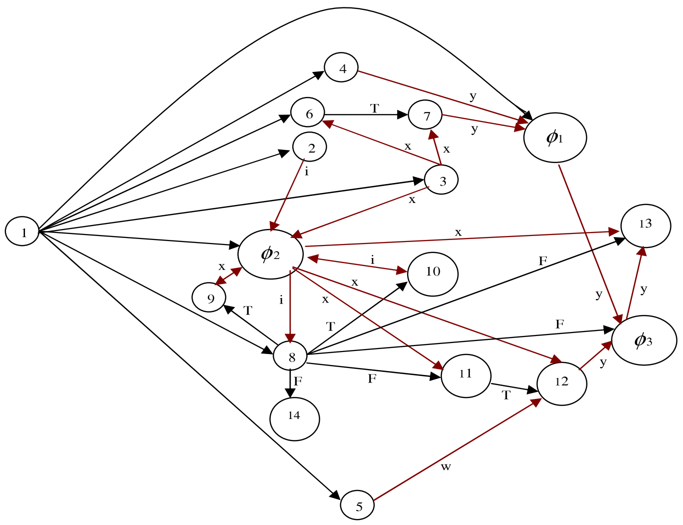
In this section, we illustrate the proposed slicing technique with an example by showing the combination of all the phases of computation (statement relevancy, semantic data dependences, conditional dependences and slicing) step by step. We

```
1.    start
2.    $i = -2$;
3.    $x = input$;
4.    $y = input$;
5.    $w = input$;
6.    if $(x \geq 0)$
7.        $y = 4x^3$;
8.    while$(i \leq 0)${
9.        $x = x \times 2$;
10.       $i = i + 1$; }
11.   if $(x \leq 0)$
12.       $y = x^2 + 4w \bmod 2$;
13.   print$(x, y)$;
14.   stop
```

(a) *Program P.*

```
1.    start
2.    $i_1 = -2$;
3.    $x_1 = input$;
4.    $y_1 = input$;
5.    $w = input$;
6.    if $(x_1 \geq 0)$
7.        $y_2 = 4 \times (x_1)^3$;
$\phi_1$    $y_3 = f(y_1, y_2)$;
      while(
$\phi_2$    $(x_2, i_2) = f((x_1, i_1), (x_3, i_3))$;
8.    $i_2 \leq 0$
      ){
9.        $x_3 = x_2 \times 2$;
10.       $i_3 = i_2 + 1$; }
11.   if $(x_2 \leq 0)$
12.       $y_4 = (x_2)^2 + 4w \bmod 2$;
$\phi_3$    $y_5 = f(y_3, y_4)$;
13.   print$(x_2, y_5)$;
14.   stop
```

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



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

Fig. 10. The traditional Program Dependence Graph (PDG).

show that our proposal results in a more precise slice (according to Definition 1) than the one proposed by Mastroeni and Zanardini [28].

Example 9. Consider the program P and the corresponding traditional Program Dependence Graph (G_{pdg}) for its SSA correspondent code, as depicted in Fig. 10.

Suppose, we are interested only in the sign of the program variables and the abstract domain *SIGN* is represented by $SIGN = \{\perp, +, 0, -, 0^+, 0^-, \top\}$ where 0^+ denotes $\{x \in Z : x \geq 0\}$, 0^- denotes $\{x \in Z : x \leq 0\}$, and Z is the set of integers. Consider the abstract slicing criterion $\langle 13, y, SIGN \rangle$ where 13 is the program point and y is the variable used at 13.

Computation of Statement Relevancy. At program point 9, the variable x may have any abstract value from the set $\{+, 0, -\}$. Since the abstract evaluation of the assignment statement $x_3 = x_2 \times 2$ at 9 does not change the sign property of x, the statement at 9 is irrelevant *w.r.t. SIGN*. After disregarding the node corresponding to this statement from the syntactic PDG G_{pdg}, we get a refined semantics-based abstract PDG G^r_{pdg} depicted in Fig. 11.

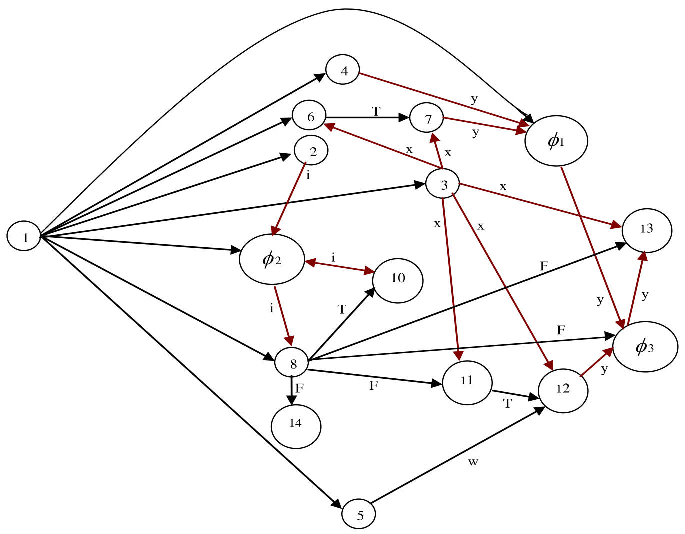


Fig. 11. G^r_{pdg}: *PDG of* P_{ssa} *after computing Statement Relevancy* $w.r.t.$ *SIGN — Observe that node* 9 *does not appear anymore*.

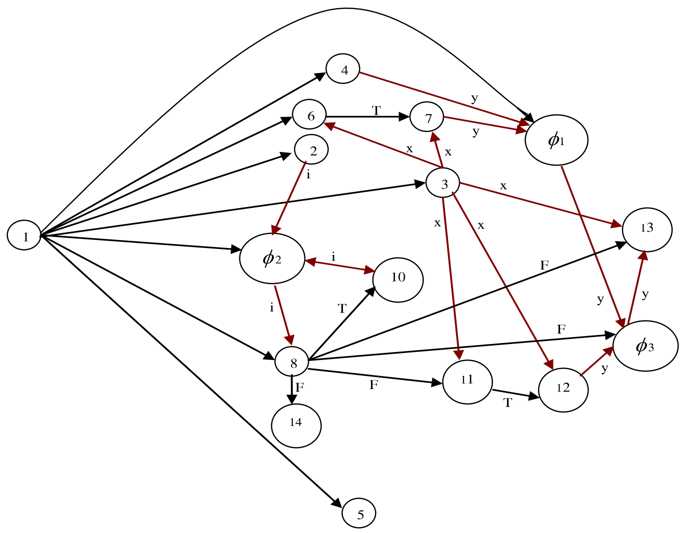
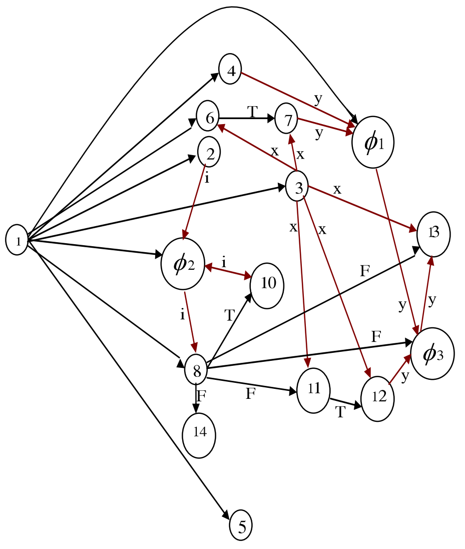


Fig. 12. $G^{r,d}_{pdg}$: *PDG of* P_{ssa} *by computing Statement Relevancy first, and then Semantic Data Dependences* $w.r.t.$ *SIGN*.

Computation of semantic data dependences. The computation of semantic data dependences [28] for all expressions in P_{ssa} *w.r.t. SIGN* reveals the fact that there is no semantic data dependence between y_4 and w at statement 12, as "$4w$ *mod* 2" always yields to 0. Therefore, by disregarding the corresponding data dependence edge $5 \xrightarrow{w} 12$ from G^r_{pdg} obtained in previous phase, we get a more refined semantics-based abstract PDG $G^{r,d}_{pdg}$ depicted in Fig. 12.

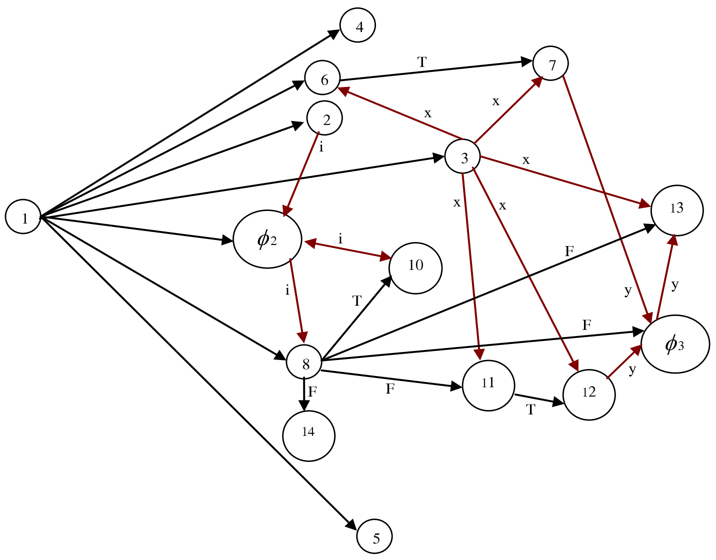
Computation of conditional dependences. Given a semantics-based abstract PDG $G^{r,d}_{pdg}$ obtained so far, we can easily convert it into the corresponding DCG G_{dcg} depicted in Fig. 13(a). Let us consider the node 4 and its associated ϕ-sequence $\eta_\phi = 4 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_3 \xrightarrow{y} 13$ in G_{dcg} representing the flow of definition at 4 to 13. Since the abstract values of x may have any value from the set $\{+, 0, -\}$ at program point 3, 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. For all execution traces 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. In short, η_ϕ is semantically unrealizable i.e. $\forall \psi : \psi \not\vdash^{SIGN} \eta_\phi$.

Therefore, if we execute the algorithm `REFINE-DCG` on G_{dcg}, since no semantically realizable ϕ-sequence exists from the node 4 to any target node t such that y defined at 4 can reach t, we remove the edge $4 \xrightarrow{y} \phi_1$ from G_{dcg}, resulting in a more precise semantics-based abstract DCG G^s_{dcg} as depicted in Fig. 13(b).



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

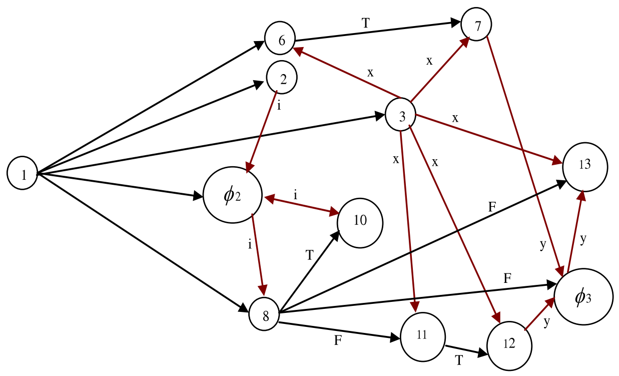
(a) G_{dcg}: *DCG after computing annotations on* $G^{r,d}_{pdg}$.



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

Fig. 13. Semantics-based abstract DCG.

Slicing w.r.t. ⟨13, y, SIGN⟩. Given the semantics-based abstract DCG G^s_{dcg} in Fig. 13(b). We apply PDG-based backward slicing technique [31] on it *w.r.t.* $\langle 13, y\rangle$ and generate a sub-DCG G_{sdcg} as depicted in Fig. 14(a). Observe that we cannot refine it anymore. Therefore, slicing of G_{sdcg} again *w.r.t.* $\langle 13, y\rangle$ yields a slice shown in Fig. 14(b).



(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.    $if\,(x \geq 0)$
7.          $y = 4x^3;$
8.    while$(i \leq 0)\{$
10.         $i = i + 1;\ \}$
11.   $if\,(x \leq 0)$
12.         $y = x^2 + 4w \bmod 2;$
```

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

Fig. 14. Slicing *w.r.t.* $\langle 13, y, SIGN\rangle$.

```
1.    start
2.    $i = -2;$
3.    $x = input;$
4.    $y = input;$
6.    $if\,(x \geq 0)$
7.          $y = 4x^3;$
8.    while$(i \leq 0)\{$
9.          $x = x \times 2;$
10.         $i = i + 1;\ \}$
11.   $if\,(x \leq 0)$
12.         $y = x^2 + 4w \bmod 2;$
```

Fig. 15. Slice *w.r.t.* $\langle 13, y, SIGN\rangle$ by Mastroeni and Zanardini [28].

Observe that if we apply only the slicing technique of Mastroeni and Zanadini [28] on the program P, we get the slice depicted in Fig. 15. According to the Definition 1, the slice obtained in Fig. 14(b) is more precise than Mastroeni and Zanadini's one obtained in Fig. 15.

8. 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.

8.1. Semantic relevancy: Soundness

When we lift the semantics-based program slicing from the concrete domain to an abstract domain, we are losing 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 8.1 (*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 $\sigma = \langle v_1, \dots, v_k \rangle \in \Sigma$ and $\epsilon = \langle \rho(v_1), \dots, \rho(v_k) \rangle \in \Sigma^\rho$. 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 $\Pi(\rho)$ [28]. The concretization of the abstract state ϵ is represented by $\gamma(\epsilon) = \{\langle u_1, \dots, u_k \rangle \mid \forall i.\ u_i \in \rho(v_i)\}$. We denote the jth concrete state in $\gamma(\epsilon)$ by the notation $\langle u_1, \dots, u_k \rangle^j$ 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:

$$\begin{aligned} S[\![s]\!]^\rho(\epsilon) &= \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \left\{S[\![s]\!]\langle u_1, \dots, u_k\rangle^j\right\}\right) \\ &= \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \left\{\langle u'_1, \dots, u'_k\rangle^j \mid S[\![s]\!]\langle u_1, \dots, u_n\rangle^j = \langle u'_1, \dots, u'_k\rangle^j\right\}\right) \\ &= \left\langle \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_1\}\right), \dots, \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_k\}\right)\right\rangle \end{aligned}$$

where u'^j_i denotes the concrete value of the ith variable $x_i \in VAR$ in the state obtained after the execution of the statement s over the jth 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

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

Therefore,

$$S[\![s]\!]^\rho(\epsilon) = \left\langle \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_1\}\right), \dots, \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_k\}\right)\right\rangle = \epsilon.$$

Then, by def. of ϵ,

$$\left\langle \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_1\}\right), \dots, \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_k\}\right)\right\rangle = \langle \rho(v_1), \dots, \rho(v_k)\rangle\,. \tag{1}$$

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

$$\left\langle \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_1\}\right), \dots, \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u'^j_k\}\right)\right\rangle = \left\langle \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u^j_1\}\right), \dots, \rho\left(\bigcup_{j\in[1..|\gamma(\epsilon)|]} \{u^j_k\}\right)\right\rangle. \tag{2}$$

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 $\Pi(\rho)$ [28]. Thus, $\forall x_i \in VAR :\ \rho(\sigma(x_i)) = \rho(v_i)$ is atomic.

Therefore, from Eqs. (1) and (2), we get

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

This allows us to conclude that for each ith program variables $x_i \in VAR$ (where $i \in [1...k]$) in all the jth concrete states (where $j \in [1..|\gamma(\epsilon)|]$), the concrete value u'^j_i which is obtained after the execution of s over those concrete states have the same property as the concrete value u^j_i 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 $w \triangleq \forall \sigma \in \Sigma_p,\ \forall x_i \in VAR\ :\ \rho(\sigma[x_i]) = \rho((S[\![s]\!]\sigma)[x_i])$. □

8.2. Complexity analysis

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

1. Compute semantic relevancy of program statements *w.r.t.* ρ.
2. Obtain semantic data dependences 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.

8.2.1. Complexity in computing statement relevancy

To compute semantic relevancy of a statement s at program point p *w.r.t.* ρ, we compare each abstract state $\epsilon \in \Sigma^\rho_p$ with the corresponding state $\epsilon' = S[\![s]\!]^\rho(\epsilon)$. If $\forall \epsilon \in \Sigma^\rho_p : \epsilon = \epsilon'$, we say that 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.

Complexity to compute abstract collecting semantics. Given a set of abstract states Σ^{ρ} and a set of program points *Label*, the context vector is defined by $\textit{Context-Vector}^{\sharp} : \textit{Label} \rightarrow \textit{Context}^{\sharp}$, where $\textit{Context}^{\sharp} = \wp(\Sigma^{\rho})$.

The context vector associated with a program P of size n is, thus, denoted by $Cv_P^{\sharp} = \langle Cx_1^{\sharp}, Cx_2^{\sharp}, \ldots, Cx_n^{\sharp}\rangle$, where $Cx_i^{\sharp}$ is the context associated with program point i in P.

Let $F_i^{\sharp} : \textit{Context-Vector}^{\sharp} \rightarrow \textit{Context}^{\sharp}$ be a collection of abstract monotone functions. For the program P, we therefore have

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

Combining the above abstract functions, we get

$$F^{\sharp} : \textit{Context-Vector}^{\sharp} \rightarrow \textit{Context-Vector}^{\sharp}.$$

That is,

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

Each function $F_i^{\sharp}$ includes the transition function defined as follows:

$$Cx_i^{\sharp} = \bigcup_{s_j \in pred(s_i)} \cup_{\epsilon_j \in Cx_j^{\sharp}} S[\![s_j]\!]^{\rho}(\epsilon_j) \tag{3}$$

where, $pred(s_i)$ is the set of predecessors of the statement s_i.

Starting from the initial context vector $Cv_P^{\sharp}=\langle \bot, \ldots, \bot\rangle$ which is the bottom element of the lattice L^n where $L = (\wp(\Sigma^{\rho}), \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, the computation of least fix-point of $F^{\sharp}$ results in the collecting semantics for P. With this collecting semantics, we can easily obtain the abstract states possibly reaching each program point in a program that helps in computing semantic relevancy of all statements *w.r.t.* ρ.

Eq. (3) says that the time complexity of each $F_i^{\sharp}$ depends on the number of predecessors of s_i and the execution time of $S[\![.]\!]^{\rho}$, assuming the number of possible abstract states appearing at each program point as constant. For a "skip" statement, $S[\![skip]\!]^{\rho}$ is constant, whereas for assignment/conditional/repetitive statements, it depends on the execution time for arithmetic and Boolean expressions occurring in those statements. Theoretically, there is no limit to the length of expressions i.e. the number of variables/constants/operations present in the expressions. However, practically, we assume that β is the maximum number of operations (arithmetic or boolean) that can be present in a statement. 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 number 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 number of statements in the program.

The least solution for $F^{\sharp}$ depends on the number of iterations 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 iterations we perform to reach the fix-point. So the time complexity for $\textit{Fix}(F^{\sharp})$ is $O(n^2\beta h)$.

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

Complexity to compute statement 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, $O(n)$.

8.2.2. *Complexity in computing semantic data dependences*

Mastroeni and Zanardini [28] introduced an algorithm to compute semantic data dependences 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 dependences 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 dependences 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 ν 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 $O(\nu)$.
- The time complexity of the atomicity condition $A^U_e(\epsilon)$: In worst case, the time complexity of $A^U_e(\epsilon)$ depends on the time complexity of $E[\![e]\!]^\rho$ and the number of elements in the covering of ϵ. Let a be the number of atomic values in the abstract domain ρ. Since the number of elements in a covering depends on the number of atomic values in the abstract domain, the time complexity of $A^U_e(\epsilon)$ is $O(a\nu)$.
- 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 $A^U_e(\epsilon)$ and the number of elements in the X-covering of ϵ. Thus, the time complexity of $A'_e(\epsilon, X)$ is $O(a^2\nu)$.

In the worst case, the recursive function that uses $A'_e(\epsilon, X)$ executes for all subsets 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 $O(a^2\nu\text{VAR})$, where VAR is the set of program variables. For a program P of size n, the number of expressions that can occur in worst case is n. Thus, finally we get the time complexity to compute semantic data dependences for a program P of size n is $O(a^2\nu n\text{VAR})$.

8.2.3. 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 $O(n)$ [37], where n is the number 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 number 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 $O(n)$ [31], the worst case time complexity to obtain DCG-based slicing is, therefore, $O(n^3h)$.

8.2.4. Overall complexity of the proposal

We may assume that the number of atomic values (a) present in the abstract domain is constant, and that $O(\beta) = O(\nu) = O(\text{VAR}) = O(n)$. Therefore, the overall complexity of our abstract program slicing technique, in case of finite height abstract lattices, is $O(n^3h)$), where n is the number of statements in the program and h is the height of the lattice of context.

9. Towards implementation

In general, the proposed abstract slicing algorithm accepts the following inputs: a program to be sliced, an abstract domain of interest, the types of semantic computations (statement relevancy, semantic data dependences, conditional dependences) to be performed, and the slicing criterion. To implement a slicing tool based on our proposal, we design the following key modules:

1. **ExtractInfo:** The module “ExtractInfo” extracts detailed information about the input programs, i.e. the type of program statements, the controlling statements, the defined variables, the used variables, etc., for all statements in the program, and stores them in a file as an intermediate representation.
2. **FormMatrix:** The module “FormMatrix” generates an incidence matrix for Control Flow Graphs (CFGs) and Program Dependence Graphs (PDGs) of the input programs based on the information extracted by the module “ExtractInfo”.
3. **GenCollectingSemantics** and **GenTraceSemantics:** Given an abstract domain, these modules compute the abstract collecting Semantics and abstract trace semantics of the input programs based on the information extracted by “ExtractInfo” and the CFG generated by “FormMatrix”.
4. **ComputeDCGannotations:** This module computes the DCG annotations (Reach Sequences and Avoid Sequences) for all CDG and DDG edges of the PDG based on the information extracted by “ExtractInfo” and the PDG generated by “FormMatrix”.
5. **ComputeSemanticDep:** It computes statement relevancy, semantic data dependences and conditional dependences of the programs under all possible states reaching each program points of the program (collecting semantics) and by computing the satisfiability of DCG annotations under its abstract trace semantics.

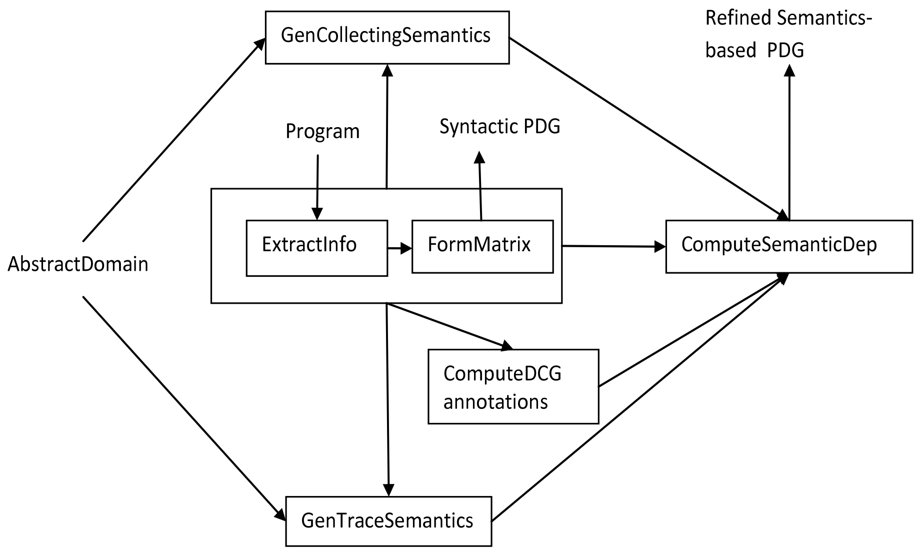


Fig. 16. Interaction between various modules.

Fig. 16 depicts the interaction between different modules mentioned above. Observe that "FormMatrix" generates a syntactic PDG based on which we can perform syntax-driven slicing *w.r.t.* a criterion, whereas "ComputeSemanticDep" refines the syntactic PDG into a semantics based abstract PDG by computing statement relevancy and semantic data dependences under its abstract collecting semantics, and the conditional dependences based on the satisfiability of DCG annotations under its abstract trace semantics. The tool should be more flexible to add any new abstract domain at any point of time. Therefore, the "Abstract Domain Interfaces" should be designed carefully.

10. Related work

All the slicing techniques in the literature make use (implicitly or explicitly) of the notions of data and control dependences in the program. The original static slicing algorithm by Mark Weiser [40] is expressed as a sequence of data-flow analysis problems and the influence of predicates on statement execution, while Korel and Lasky [23] extended it into the dynamic context and proposed an iterative dynamic slicing algorithm based on dynamic data flow and control influence. An improvement of the efficiency of data flow analysis-based program slicing techniques in presence of pointer variables is achieved in [25], by performing an equivalence analysis of memory locations. [25] showed that the use of equivalence classes significantly improves the Harrold and Ci's reuse-driven slicing algorithm [19] to achieve more reuse when applied to programs using pointers.

Over the last 25 years, many slicing techniques have been proposed based on the Program Dependence Graph (PDG) [13,31,34], System Dependence Graph (SDG) [22,36], or Dynamic Dependence Graph (DDG) [1] representations.

De Lucia et al. [26] defined the base vocabulary and slicing criteria for static [40], dynamic [1,23], quasi static [38], simultaneous dynamic [17], and conditioned slicing [6] using formal notation. They defined the conditioned slice as a general purpose slice that lies between static (every possible input considered) and dynamic (only one input considered) slicing, where it is possible to consider a subset of the input states. This is done by specifying a logical formula on the program input, which identifies the set of states satisfying it. In one sense, the semantic view of abstract slicing includes conditioned slicing as a way to specify predicates on states, but mostly uses such predicates as a way to track which path has been taken during the computation, thus increasing the precision of the analysis. A formal mathematical framework that enables us to define and compare different forms of slicing, is developed in [5] based on the projection theory.

Various semantic justification of program slicing have been introduced to prove the correctness of slicing algorithms. These include Finite Trajectory-based semantics introduced by Weiser [39]; Non-standard semantic approaches such as Non-strict (or Lazy) semantics [7,38,12], Transfinite semantics [15,30]; and very recent Trajectory-based Strict semantics [2] which is an extension of [39] to non-terminating programs.

Hong et al. [35] introduced a program slicing approach based on Abstract Interpretation and Model Checking, where the abstract interpretation is considered in the restricted area of predicate abstraction. They used abstract state graphs which are obtained by applying predicate abstraction to programs with predicates and constraints, rather than a flow graph. Since the size of an abstract state graph grows exponentially in the number of predicates, they formulated abstract slicing in terms of symbolic model checking that has been shown to be effective for controlling such state explosion problem.

In traditional PDGs, the notion of dependences between statements is based on the syntactic presence of a variable in the definition of another variable or in a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependences. Mastroeni and Zanardini [28] first introduced the notion of semantic data dependences which fills up the existing gap between syntax and semantics. The semantic data dependences which are computed for all expressions in the program over the states possibly reaching the associated program points, help in obtaining more precise semantics-based PDGs by removing some false dependences from the traditional syntactic PDGs. This is the basis to design an 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.

Zanardini [42] introduced a similar notion of invariance of statements based on the notion "agreement", i.e. the set of conditions that are preserved at a given program point. He defined an abstract slicing framework that includes the possibility to restrict the input states of the program, in the style of abstract conditioned slicing, thus lying between static and dynamic slicing. His semantic approach for abstract slicing has much stronger requirements than we have, as it needs various static analysis components, like an invariance analyzer, weakest precondition calculus, strongest postcondition calculus, and symbolic executor.

The property driven program slicing technique in [4] considers properties of variables' values and is based on the data-flow analysis. Rival [33] discussed abstract dependences representing data properties by means of abstract interpretation, and its applications to alarm diagnosis, together with techniques for analyzing and composing dependences. Most recently, Mastroeni and Nicolić [27] extended the theoretical framework of slicing proposed by Binkley [5] to abstract domain in order to define the notion of abstract slicing, and to represent and compare different forms of slicing in abstract domain. Their proposal towards the implementation of abstract program slicing approaches refers to an abstract state graph representation of the programs.

11. Discussions and conclusions

We acknowledge that there are other possible improvements that deserve to be considered as future works.

For instance, the semantic relevancy at the statement level does not take into account the semantic interaction between statements. For example, if we 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*. This can be seen as possible improvement to our algorithm.

In [28], the problem related to the control dependences is not addressed. For example, consider the following example:

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

Here the semantic data dependence says that the condition in the "*if*" statement is only dependent on y. But it does not say anything about the dependence between w and y. Observe that although w is invariant *w.r.t.* the evaluation of the guard, this is not captured by [28]. The block-level semantic relevancy, rather than statement-level, can resolve this issue of independences. 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 dependences and conditional dependences in the refinement of the PDGs can be applied effectively to the context of static, dynamic, and forward conditioned slicing. Since the allowed initial states are different for different forms of slicing, we compute statement relevancy and semantic data dependences 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 criterion. Similarly, the satisfiability of the dependence paths in the DCG are checked against the traces generated from the allowed initial states only. For instance, in case of forward conditioned slicing, a condition is specified in the slicing criterion to disregard some initial states that do not satisfy it. In case of dynamic slicing, inputs of interest are specified in the slicing criterion. Therefore, the collecting semantics and execution traces are generated based on the allowed initial states specified or satisfying the conditions in the slicing criterion, and are used to compute statement relevancy, semantic data dependences and satisfiability of the dependence paths.

The combination of results on the refinement of dependence graphs with static analysis techniques discussed in this paper 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. The effective applicability of our proposal to other forms of slicing, like backward conditioned slicing, simultaneous slicing, amorphous slicing, etc [5,18,27,38] is also an interesting direction of our ongoing research.

Acknowledgments

This work is partially supported by RAS L.R. 7/2007 Project TESLA. We thank the anonymous reviewers for their useful comments and remarks.

References

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

[2] R.W. Barraclough, D. Binkley, D. Sebastian, M. Harman, R.M. Hierons, Á. Kiss, M. Laurence, L. Ouarbya, A trajectory-based strict semantics for program slicing, Theoretical Computer Science 411 (11–13) (2010) 1372–1386.

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

[4] S. Bhattacharya, Property driven program slicing and watermarking in the abstract interpretation framework, Ph.D. Thesis, Università Ca' Foscari Venezia, 2011.

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

[6] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, Software salvaging based on conditions, in: Proceedings of the 10th International Conference on Software Maintenance, ICSM '94, IEEE Computer Society, Victoria, British Columbia, Canada, 1994, pp. 424–433.

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

[8] Agostino Cortesi, 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, ACM Press, Paphos, Cyprus, 2010.

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

[10] P. Cousot, 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, ACM Press, Los Angeles, CA, USA, 1977, pp. 238–252.

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

[12] Sebastian Danicic, Mark Harman, John Howroyd, Lahcen Ouarbya, A non-standard semantics for program slicing and dependence analysis, Journal of Logic and Algebraic Programming 72 (2) (2007) 191–206.

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

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

[15] Roberto Giacobazzi, Isabella Mastroeni, Non-standard semantics for program slicing, Higher-Order and Symbolic Computation 16 (4) (2003) 297–339.

[16] Raju Halder, Agostino Cortesi, Abstract program slicing on dependence condition graphs, Technical Report, Università Ca' Foscari Venezia, 2011. (http://www.dsi.unive.it/~cortesi/slice_tech_report.pdf).

[17] R.J. Hall, Automatic extraction of executable program subsets by simultaneous program slicing, The Journal of Automated Software Engineering 2 (1) (1995) 33–53.

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

[19] Mary Jean Harrold, Ning Ci, Reuse-driven interprocedural slicing, in: Proceedings of the 20th International Conference on Software Engineering, ICSE '98, IEEE Computer Society, Kyoto, Japan, 1998, pp. 74–83.

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

[21] S. Horwitz, J. Prins, 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, ACM Press, San Diego, California, United States, 1988, pp. 146–157.

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

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

[24] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, in: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '81, ACM Press, Williamsburg, Virginia, 1981, pp. 207–218.

[25] Donglin Liang, Mary Jean Harrold, Equivalence analysis and its application in improving the efficiency of program slicing, ACM Transactions on Software Engineering and Methodology 11 (3) (2002) 347–383.

[26] A. De Lucia, A.R. Fasolino, M. Munro, Understanding function behaviors through program slicing, in: Proceedings of the 4th Workshop on Program Comprehension, WPC '96, IEEE Computer Society, Berlin, Germany, 1996, pp. 9–18.

[27] Isabella Mastroeni, Durica Nikolic, Abstract program slicing: From theory towards an implementation, in: Proceedings of the 12th International Conference on Formal Engineering Methods, ICFEM '10, in: LNCS, vol. 6447, Springer, Shanghai, China, 2010, pp. 452–467.

[28] Isabella Mastroeni, 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, ACM Press, San Francisco, California, USA, 2008, pp. 125–134.

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

[30] Harmel Nestra, Transfinite semantics in program slicing, 11 (4) (2005) 313–328.

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

[32] T. Reps, G. Rosay, Precise interprocedural chopping, in: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT '95, ACM Press, Washington, DC, USA, 1995, pp. 41–52.

[33] Xavier Rival, Abstract dependences for alarm diagnosis, in: Proceedings of the Third Asian Conference on Programming Languages and Systems, APLAS'05, in: LNCS, vol. 3780, Springer, Tsukuba, Japan, 2005, pp. 347–363.

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

[35] Hyoung Seok Hong, Insup Lee, 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, IEEE Computer Society, Budapest, Hungary, 2005, pp. 25–34.

[36] S. Sinha, M. Harrold, 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, ACM Press, Los Angeles, CA, USA, 1999, pp. 432–441.

[37] S. Sukumaran, A. Sreenivas, R. Metta, The dependence condition graph: precise conditions for dependence between program points, Computer Languages, Systems & Structures 36 (2010) 96–121.

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

[39] Mark Weiser, Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, Ph.D. Thesis, University of Michigan Ann Arbor, MI, USA, 1979.
[40] Mark Weiser, Program slicing, IEEE Transactions on Software Engineering SE-10 (4) (1984) 352–357.
[41] Glynn Winskel, The Formal Semantics of Programming Languages: An Introduction, The MIT Press, 1993.
[42] Damiano Zanardini, The semantics of abstract program slicing, in: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM '08, IEEE Press, Beijing, China, 2008, pp. 89–100.