Dependence Condition Graph for Semantics-based Abstract Program Slicing

Agostino Cortesi and Raju Halder

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

Abstract

Many slicing techniques have been proposed based on the traditional Program Dependence Graph (PDG) representation. In traditional PDGs, the notion of dependency between statements is based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Mastroeni and Zanardini introduced a semantics-based dependency both at concrete and abstract domain. This semantic dependency is computed at expression level over all possible (abstract) states appearing at program points. In this paper we strictly improve this approach by (*i*) considering the semantic relevancy of the statements (not only the expressions), and (*ii*) adopting conditional dependency. This allows us to transform the semantics-based abstract PDG into an semantics-based abstract Dependence Condition Graph (DCG) that enables to identify the conditions for dependence between program points. The resulting program slicing algorithm is stricly more accurate than the Mastroeni and Zanardini's one.

1 Introduction

Program slicing is a well-known decomposition technique that extracts from programs the statements which are relevant to a given behavior. In particular, a slice is an executable program whose behavior must be identical to a specific subset of the original behavior, and is obtained by deleting statements from the original program. The notion of program slice was originally introduced by Mark Weiser [29]. It is a fundamental operation for addressing many software-engineering problems, including program understanding, debugging, maintenance, testing, parallelization, integration, software measurement etc. See, for instance, [7, 17, 21, 1, 19, 16, 14, 12, 8, 23, 24].

Static [29] and dynamic slicing [15] are two different types of slicing techniques where the former is done at compile time while later is performed at runtime with user inputs. A static slice preserves the program's behavior with respect to variables for all program inputs. The slicing criteria is denoted by

$\langle p, v \rangle$ where v is the variable of interest at program point identified by line number p. This is not restrictive, as it can be easily 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, 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 dependencies that occur in a specific execution of the program are taken into account.

Over the last 25 years, many slicing techniques have been proposed based on the traditional Program Dependence Graph (PDG) representation [18, 13, 27, 11, 10, 22, 24, 6, 23, 26]. The PDG makes explicit both the data and control dependencies for each operation in a program. Data dependencies have been used to represent only the relevant data flow relationship of a program while the control dependencies are derived from the actual control flow graph. The sub-graph of the PDG induced by the control dependence edges is called the control dependence graph (CDG) and the sub-graph induced by the data dependence edges is called the data dependence graph (DDG).

In traditional PDGs, the notion of dependency between statements is based on syntactic presence of a variable in the definition of another variable or on a conditional expression. Therefore, the definition of slices at semantic level creates a gap between slicing and dependencies.

Mastroeni and Zanardini in [20] introduced a semantics-based dependency which fills up the existing gap between syntax and semantics. Based on this semantic dependency, a more precise PDG can be obtained by removing the false dependencies from the traditional syntactic PDG. The semantic dependency can also be lifted to an abstract domain where dependencies are computed with respect to some specific properties of interest rather than concrete values. This (abstract) semantic dependency is computed at expression level over all possible (abstract) states appearing at program points.

In [28], Sukumaran et al. introduced Dependence Condition Graph (DCG), a refinement of PDGs based on the notion of conditional dependency. This is obtained by adding the annotations which encode the condition under which a particular dependence actually arises in a program execution.

In this paper we combine these two concepts: semantic data dependency [20] and conditional dependency [28], (*i*) by considering the semantic relevancy of the statements (not only the expressions), and (*ii*) by adopting conditional dependency. This allows us to transform the semantics-based (abstract) PDG into an semantics-based (abstract) DCG that enables to identify the conditions for dependence between program points. The resulting (abstract) slicing algorithm is strictly more precise than the one in [20].

The rest of the paper is organized as follows: Section 2 provides a motivating example. Section 3 recalls some basic ideas about abstract interpretation theory covering the abstract semantics of the expressions, statements and the induced partitioning over the abstract domain. Section 4 introduces the (abstract) semantic relevancy of the statements. In section 5 we extend the semantics-based (abstract) PDGs into DCGs. Finally, in Section 6 we draw our conclusions.

2 A motivating example

In [20], the traditional syntactic based PDG is refined by introducing the notion of semantic dependency both at concrete and abstract level. The semantic dependency computes the data dependency between an expression e and the set of variables $var(e)$ involved in e at program point p. The expression e does not semantically depend on a variable $x \in var(e)$ if the evaluation of e over any two different states σ_1 and σ_2 appearing at p where $\forall y \in var(e) \wedge y \neq x : \sigma_1(y) = \sigma_2(y)$, results the same values for e. Although the presence of variable x in expression e shows the syntactic data dependency of e on x, semantically there is no such dependency. For instance, the expression $e = x + x - 2x + 4$ does not depend semantically on x. The concrete semantic data dependency can easily be lifted to an abstract domain representing specific property of interest. The abstract semantic data dependency describes the semantic dependency on abstract values rather than the concrete values. This (abstract) semantic dependency is used to eliminate some irrelevant dependencies from the traditional PDG, resulting in a more precise slice.

However, if we observe carefully, we see that the semantic dependency derived at expression level [20] does not always result in a more precise PDG for the program.

Example 1 *Consider the program P and the corresponding traditional Program Dependence Graph (G_{pdg})[1] as depicted in Figure 1. Suppose we are interested only in the sign of the program variables, and we consider the abstract domain SIGN = $\{\bot, +, 0, -, 0^+, 0^-, \top\}$ where 0^+ represents $\{x \in Z : x \geq 0\}$ and 0^- represents $\{x \in Z : x \leq 0\}$. After computing the abstract semantic data dependency [20] w.r.t. the sign property, we get the semantics-based abstract PDG shown in Figure 2(a). Observe that, since the semantic dependency w.r.t. SIGN removes the data dependency between y_4 and w at statement 12 (as "$4w \bmod 2$" always yields to 0), the corresponding data dependence edge $5 \xrightarrow{w} 12$ is disregarded from the traditional PDG. The slicing algorithm based on this semantics-based abstract PDG with the criteria $\langle 13, y \rangle$ would return the program depicted in Figure 2(b). At program point 9, the variable x_2 may have any abstract value in $\{+, 0, -\}$. Since the evaluation of the expression $x_2 \times 2$ over all these possible abstract values yields to the dependency of the expression $x_2 \times 2$ on x_2 w.r.t. SIGN, the dependency is included in the semantics-based abstract PDG by the edge linking node 9 with node q_2.*

However, the execution of statement 9 does not affect at all the sign of x. This "false positive" is due to the fact that the semantic dependency in [20] is defined at expression level. The abstract semantics of the program should say that statement 9 is not relevant for the slicing criteria $\langle 13, y \rangle$. Thus, slicing with criteria $\langle 13, y \rangle$ should have the correct, and more precise, slice shown in Figure 3(b), as the sign of x at line 11 and 12 in the original program is the same as that in the input value.

The point we raise is that the semantics-based (abstract) PDG obtained from the (abstract) semantic dependency [20] can be improved w.r.t. accuracy if, before deriving the dependency at expression level, we compute the semantic relevancy of statements in

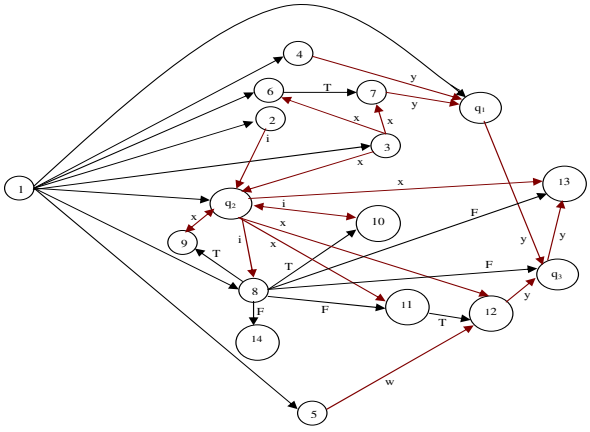
[1] The node in G_{pdg} corresponding to the statement ϕ_i (i^{th} ϕ statement) in P_{ssa} is labeled as q_i.

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

(a) Program P

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

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

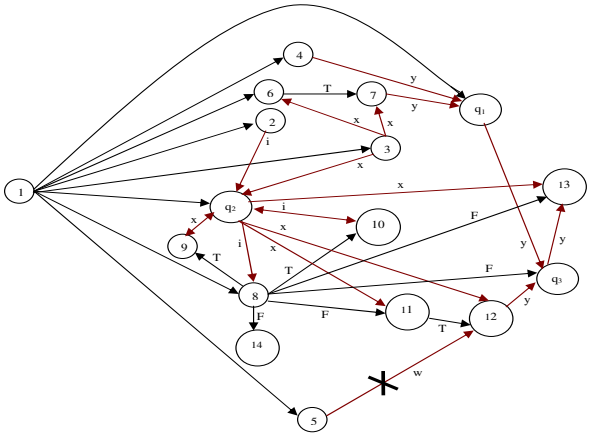


(c) G_{pdg}: PDG of P_{ssa}

Figure 1: The traditional Program Dependency Graph (PDG)

the program. This way, we can refine the PDG, by eliminating the nodes corresponding to the semantically irrelevant statements from the PDG. In a similar way, we can eliminate all nodes corresponding to a conditional block if all the statements in that block are not semantically relevant at all, as illustrated in Section 4. In our example, Figure 3(a) depicts a more precise semantics-based abstract PDG $G^{r,d}_{pdg}$ obtained by computing first the abstract semantic relevancy at statement level which removes the node corresponding to the statement 9, and then by computing the abstract semantic data dependency [20] which removes the data dependency on w at statements 12.

By following [28], we can extend every semantics-based (abstract) PDG obtained so far into the semantics-based (abstract) Dependence Condition Graphs DCG with the annotation $e^b \triangleq \langle e^R, e^A \rangle$ over all the data/control dependence edges e (from e.src to e.tgt): e^R is referred to as Reach Sequence and represents the conditions required to reach e.tgt from e.src, and e^A is referred to as Avoid Sequence and used to avoid re-definitions (in case of control dependence edge, e^A is $\emptyset$). An execution ψ over an abstract domain, is said to satisfy e^b for an data dependence edge e if it satisfies all



(a) G^d_{pdg} : *PDG of* P_{ssa} *after computing Semantic Dependency w.r.t. SIGN* [20]

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

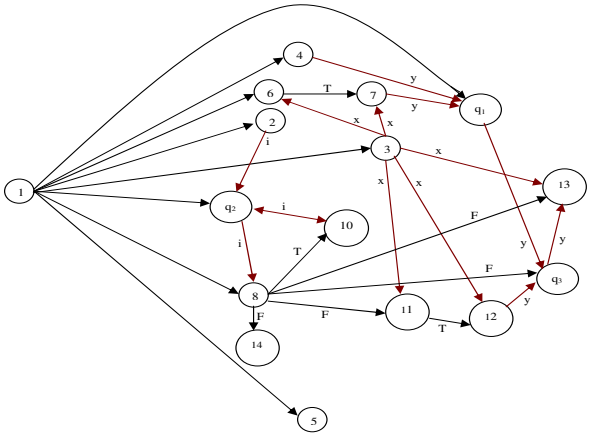
(b) *Slice with criteria* $\langle 13, y\rangle$ *computed from* G^d_{pdg}

Figure 2: Semantics-based abstract PDG and corresponding slice after computing Semantic Dependency w.r.t. SIGN

the conditions in e^R *and at the same time it avoids the conditions represented by* e^A*; this means that the execution* ψ *ensures that the abstract value computed at e.src successfully reaches e.tgt and has an impact on the abstract value of e.tgt. We can extend this to any data dependence PDG path (known as* ϕ*-sequence)* $\eta = e_1 e_2 \dots e_n$*: given a* ϕ*-sequence* $\eta = e_1 e_2 \dots e_n$ *and an execution* ψ *over an abstract domain, we say that* ψ *satisfies* η *(written as* $\psi \vdash \eta$*) if the abstract value computed at* e_1*.src reaches to* e_n*.tgt and has an impact on the abstract value of e.tgt in* ψ*.*

Recall the algorithm to compute DCG reported in [28]. For the program in Figure 1 and its semantics-based abstract PDG $G^{r,d}_{pdg}$ *in Figure 3(a), consider the control dependence path* $p \triangleq 1 \rightarrow 8 \xrightarrow{false} 11$ *and the control dependence edge* $c \triangleq 11 \xrightarrow{true} 12$*. We get the reach sequence for c,* $c^R = \{11 \xrightarrow{true} 12\}$ *which means that to execute the statement 12 successfully the condition at 11 must be true. Now there is one data dependence edge:* $d = 3 \xrightarrow{x} 12$ *that has statement 12 as target. For this data dependence edge, we get the reach sequences as* $d^R = \{1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$*. This means that the condition* $1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$ *has to be satisfied so that 3 and 12 both can get executed.*

To compute the avoid sequence for the edge $d_1 = \phi_1 \xrightarrow{y} \phi_3$*, consider two data*



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

```
1.    start
3.    x=input;
4.    y=input;
6.    if(x ≥ 0)
7.          y = 4x^3;
11.   if(x ≤ 0)
12.         y = x^2 + 4w mod 2;
```

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

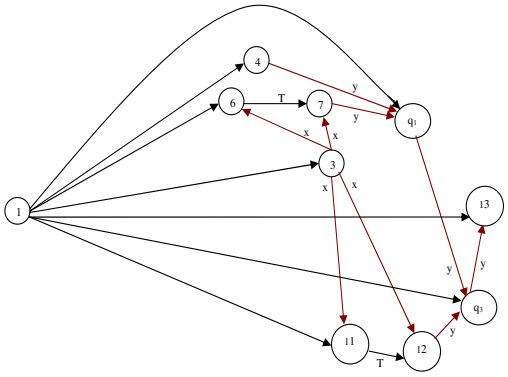
Figure 3: Semantics-based abstract PDG and corresponding slice by computing Statement Relevancy first, and then Semantic Dependency w.r.t. SIGN

dependence edges $d_1 = \phi_1 \xrightarrow{y} \phi_3$ *and* $d_2 = 12 \xrightarrow{y} \phi_3$ *with* ϕ_3 *as target. Following the algorithm of [28], we get* $d^A_1 = (\phi_1 \xrightarrow{y} \phi_3)^A = \{1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12\}$. *This reflects the fact that the "if" condition at 11 must be false in order to guarantee that the definition at* ϕ_1 *is not re-defined at 12 and can reach* ϕ_3. *Similarly, we get* $(4 \xrightarrow{y} \phi_1)^A = \{1 \rightarrow 6 \xrightarrow{true} 7\}$. *Figure 4(a) depicts the DCG annotations over the data dependence edges of* $G^{r,d}_{pdg}$.

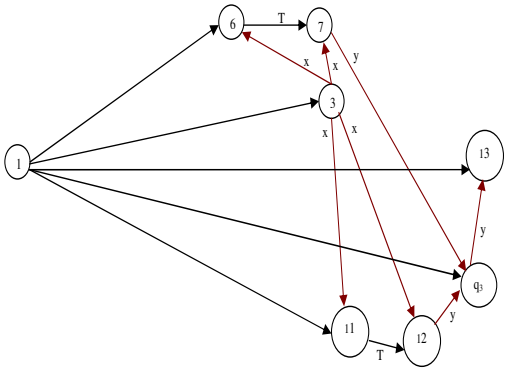
Given the slicing criteria $\langle 13, y\rangle$, *the sub-PDG* $G^{r,d}_s$ *obtained after performing backward slicing technique on* $G^{r,d}_{pdg}$ *is shown in Figure 4(b). Let us consider the PDG path* $\eta = 4 \xrightarrow{y} \phi_1 \xrightarrow{y} \phi_3 \xrightarrow{y} 13$ *in* $G^{r,d}_s$ *which is a* ϕ*-sequence [28] and indicates the flow of definition at 4 to 13. Observe that, since the abstract values of x may have any value from the set* $\{+, 0, -\}$, *there is no such execution* ψ *over the abstract domain SIGN which avoids both* $(4 \xrightarrow{y} \phi_1)^A$ *and* $(\phi_1 \xrightarrow{y} \phi_3)^A$ *simultaneously i.e.* $\forall \psi : \psi \nvdash \eta$. *For each execution over the abstract domain of signs, at least one of the conditions among* $6 \xrightarrow{true} 7$ *and* $11 \xrightarrow{true} 12$ *will be satisfied. This means that the definition at 4 is over-written by 7 or 12 and never reaches 13. Since there exists no semantically realizable PDG path between 4 and 13, we can remove the node 4 from* $G^{r,d}_s$ *as depicted in Figure 4(c) and thus, we get the final more precise slice as shown in Figure 4(d).*

d	d^R	d^A
$4 \xrightarrow{y} \phi_1$	$\emptyset$	$1 \rightarrow 6 \xrightarrow{true} 7$
$7 \xrightarrow{y} \phi_1$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 6$	$\emptyset$	$\emptyset$
$3 \xrightarrow{x} 7$	$1 \rightarrow 6 \xrightarrow{true} 7$	$\emptyset$
$3 \xrightarrow{x} 11$	$1 \rightarrow 8 \xrightarrow{false} 11$	$\emptyset$
$3 \xrightarrow{x} 12$	$1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$	$\emptyset$
$3 \xrightarrow{x} 13$	$1 \rightarrow 8 \xrightarrow{false} 13$	$\emptyset$
$12 \xrightarrow{y} \phi_3$	$\emptyset$	$\emptyset$
$\phi_1 \xrightarrow{y} \phi_3$	$1 \rightarrow 8 \xrightarrow{false} \phi_3$	$1 \rightarrow 8 \xrightarrow{false} 11 \xrightarrow{true} 12$
$\phi_3 \xrightarrow{y} 13$	$\emptyset$	$\emptyset$

(a) *The annotations* $\langle d^R, d^A \rangle$ *over the data dependence edges* d *of the PDG* $G^{r,d}_{pdg}$



(b) $G^{r,d}_s$: *sub−PDG obtained after performing backward slicing on* $G^{r,d}_{pdg}$ *w.r.t.* $\langle 13, y \rangle$



(c) $G^{r,d,c}_s$: *sub−PDG obtained from* $G^{r,d}_s$ *after computing Conditional Dependencies*

```
1.    start
3.    x=input;
6.    if(x ≥ 0)
7.        y = 4x^3;
11.   if(x ≤ 0)
12.       y = x^2 + 4w mod 2;
```

(d) *Slice with criteria* $\langle 13, y \rangle$ *computed from* $G^{r,d,c}_s$

Figure 4: Semantics-based abstract sub-PDG and corresponding slice *w.r.t.* $\langle 13, y \rangle$ after computing Conditional Dependencies *w.r.t.* SIGN

3 Abstract Interpretation

Abstract Interpretation, originally introduced by Cousot and Cousot is a well known semantics-based static analysis technique [3, 4, 5, 9]. Its main idea is to relate concrete and abstract semantics where the latter are focussing only on some properties of interest. Abstract semantics is obtained from the concrete one by substituting concrete domains of computation and their basic concrete semantic operations with abstract domains and corresponding abstract semantic operations. This can be expressed by means of closure operators.

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

Abstract Semantics: Expressions and Statements

As in [20], we consider the IMP language [30]. The statements of a program P act on a set of constants $\mathbb{C} = const(P)$ and set of variables $VAR = var(P)$. A program variable $x \in VAR$ takes its values from the semantic domain $\mathbb{V} = Z_{\mho}$ where, $\mho$ represents an undefined or uninitialized value and Z is the set of integers. The arithmetic expressions $e \in Aexp$ and boolean expressions $b \in Bexp$ are defined by standard operators on constants and variables. The set of states Σ consists of functions $\sigma : VAR \to \mathbb{V}$ which map the variables to their values. For the program with k variables $x_1, \ldots, x_k$, the state is denoted by k-tuples: $\sigma = \langle v_1, \ldots, v_k \rangle$, where $v_i \in \mathbb{V}, i = 1, ..., k$ and hence, set of states $\Sigma = (\mathbb{V})^k$. Given a state $\sigma \in \Sigma$, $v \in \mathbb{V}$, and $x \in VAR$: $\sigma[x \leftarrow v]$ denotes a state obtained from σ by replacing its contents in x by v, *i.e.* define

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

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

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. The partial semantics $P[\![s]\!]_p(\sigma)$ is obtained by collecting all possible states appearing at program point p inside s when s is executed 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 $\epsilon = \langle \rho(v_1), \ldots, \rho(v_k) \rangle \in \Sigma^\rho : E[\![e]\!]^\rho(\epsilon) = \rho(\{E[\![e]\!](u_1, \ldots, u_k) \mid \forall i.\ u_i \in \rho(v_i)\})$.

Similarly, semantics $P[\![s]\!]^\rho_p(\epsilon_0)$ (where $\epsilon_0 = \langle \top, \ldots, \top \rangle$) computes a safe over-approximation of the minimal abstract state $\epsilon'_p \in \Sigma^\rho$ which describes variables

at p:

$$P[\![s]\!]^{\rho}_{p}(\epsilon_0) \triangleq \epsilon_p \geq \epsilon'_p = \rho(\bigcup\{P[\![s]\!]_p(\sigma) \mid \sigma \in \Sigma\})$$

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

Partitions, atoms

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

4 Semantic Relevancy

Given a program P, we define a new semantics-based (abstract) PDG, constructed in two steps:

(1) First, the semantic relevancy of all the statements of the program P is computed. Corresponding to each semantically relevant statement of the program, we create a node in the PDG. The nodes corresponding to the conditional ($if, if-else$) or repetitive statements ($while$) and their control dependencies are inserted if the corresponding blocks are semantically relevant (*i.e.* if at least one statement in the block is semantically relevant).

(2) Second, this PDG is refined by computing the semantic dependency at expression level [20] for all relevant statements to keep only the semantic data dependencies.

Definition 1 *If $\forall \epsilon \in \Sigma^{\rho}$: $P[\![s]\!]^{\rho}_{p}(\epsilon) = \epsilon$, the statement s is not semantically relevant with respect to the abstract domain ρ.*

In other words, the statement s at program point p is semantically irrelevant if no changes take place in the abstract states ϵ occurring at p, when s is executed over ϵ. The statements which do not contribute to any change in the states occurring at that program point are considered semantically irrelevant.

Note that atomicity of the abstract value for each variable in the abstract state ε *w.r.t.* property ρ is one of the crucial requirements during computation of ρ-relevancy of the statements. These atomic abstract values are obtained from induced partitioning. The following example shows how to compute the semantic relevancy for the statements by using covering techniques.

Example 2 *Consider the abstract domain of parity PAR and an abstract state $\varepsilon = \langle even, odd, \top \rangle$ for the variables $x, y, z \in dom(\varepsilon)$. The induced partition for the domain PAR is $\Pi(PAR) = \{even, odd\}$. Since $\top$ is not an atomic state for the domain PAR, we can instead consider a covering $\{\varepsilon_1, \varepsilon_2\}$ for the state ε, where $\varepsilon_1 = \langle even, odd, even \rangle$ and $\varepsilon_2 = \langle even, odd, odd \rangle$. Hence, the relevancy for the statement s at program point p (if ε occurs at p) is computed over ε_1 and ε_2. Observe that, the elements in the cover contains atomic abstract values (atoms, in other word) for the variables.*

Consider the example shown in Figure 1. One possible abstract state at program points 9 and 10 w.r.t. SIGN is $\varepsilon = \langle -, \top, \top, \top \rangle$ *where* $dom(\varepsilon) = \langle i, x, y, w \rangle$*. Since the values for* x, y, w *are provided by the user, it can be any value from the set* $\{+, 0, -\}$ *and is denoted by the top element* $\top$ *of the lattice for SIGN. When we compute the semantic relevancy of statements 9 and 10, the execution over the abstract state* ε *can not reveal the fact of semantic relevancy because of the overapproximated state and lack of precision. Therefore, we compute the semantic relevancy of 9 and 10 over the covering of* ε *i.e.* $\{\langle -, -, -, - \rangle, \langle -, 0, -, - \rangle, \langle -, +, -, - \rangle, \ldots, \langle -, +, +, + \rangle\}$*. Computing over these covering states, we can easily conclude the semantic (ir)relevancy of 9 and 10.*

Treating Conditional Statements

Example 1 illustrated how to obtain a more precise PDG by eliminating the nodes corresponding to the semantically irrelevant statements in the program. In this section we describe how we can disregard any conditional dependency if all statements in that conditional block are not semantically relevant at all.

Example 3 *Consider the program of Figure 5(a) and the property PAR. The traditional PDG of it, is shown in Figure 5(b). At program point 2 and 3, the parity of x and y are odd and even respectively. These properties of x and y remain unchanged during the while loop. Statements 5 and 6, therefore, are not semantically relevant w.r.t. PAR. Thus, we can disregard the nodes corresponding to these statements yielding to the more precise PDG shown in Figure 5(c). Note that the node corresponding to the repetitive statement "while" (node 4) does not appear in Figure 5(c) because there is no semantically relevant statement in the while block.*

We follow the same approach in case of "if" conditional block: the node corresponding to the "if" conditional statement is disregarded from the PDG if the corresponding block has no semantically relevant statement inside.

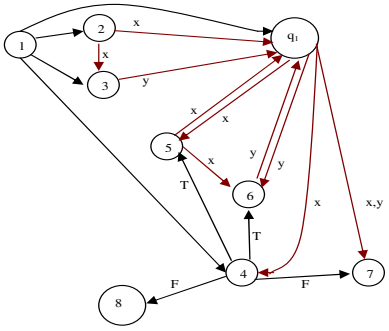
We now discuss about the "$if - else$" conditional block appears in the program. We may distinguish four cases:

1. All statements in "if" block are semantically irrelevant, whereas some/all statements in "*else*" block are semantically relevant *w.r.t.* property ρ.
2. Some/all statements in "if" block are semantically relevant, whereas all statements in "*else*" block are semantically irrelevant *w.r.t.* property ρ.
3. Some/all statements in both "if" and "*else*" blocks are semantically relevant *w.r.t.* property ρ.
4. All statements in both "if" and "*else*" blocks are semantically irrelevant *w.r.t.* property ρ.

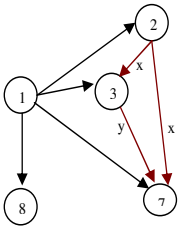
In case 4, we can disregard all the nodes corresponding to the complete "$if-else$" block including the conditional statement from the PDG as all the statements in both "if" and "*else*" blocks are semantically irrelevant. In case 1, we can replace all nodes corresponding to the all statements in "if" block with a single node that corresponds to the statement *skip* with no data flow, whereas in case 2, we can disregard all nodes corresponding to the complete "*else*" block. Observe that, when some or all statements are semantically relevant in either block,

1.	Start
2.	$x_1 = 1;$
3.	$y_1 = 2 \times x_1;$
	while(
ϕ_1	$(x_2, y_2) = f((x_1, y_1), (x_3, y_3))$
4.	$x_2 < 20$){
5.	$x_3 = x_2 + 2;$
6.	$y_3 = y_2 + 2 \times x_3^2;$}
7.	*print*(x_2, y_2);
8.	Stop

(a) Program P



(b) Traditional PDG of the Program P



(c) Semantics-based abstract PDG of P after computing semantic relevancy *w.r.t. PAR*

Figure 5: Treating "*while*" conditional block

only the nodes corresponding to those semantically irrelevant statements (if they exist) will be disregarded from the PDG as in case 3. Lets illustrate with an example shown in Example 4.

Example 4 *Consider the program of Figure 6(a). Observe that the statements 6 and 7 in the "if" conditional block are semantically irrelevant w.r.t. the sign property of the variables. The execution of statements 5 and 6 over all possible abstract states appearing at those program points does not change the sign of y and z. But the "else" block is semantically relevant as it contains semantically relevant statement 9 w.r.t. the sign property. According to case 1, we can't remove the "if" block. Hence, we replace these semantically irrelevant statements in the "if" block with the statement "skip". The corresponding form of the program and semantics-based abstract PDG are shown in Figure 6(b) and 6(c) respectively.*

In [20], the problem related to the control dependency is not addressed: even if they consider (abstract) semantic dependencies, they still could not be able to reach the most precise slice for a given criterion. For example, consider the following example:

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

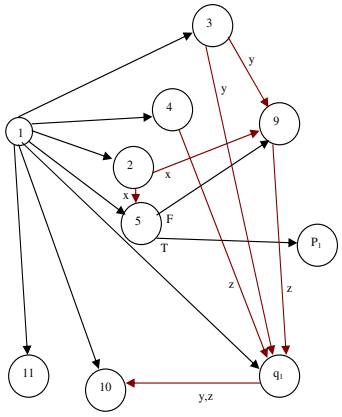
Here the abstract semantic dependency says that the condition of "*if*" statement is only dependent on y. But it does not say anything about the dependency

1.	Start
2.	$x_1 = input;$
3.	$y_1 = 5;$
4.	$z_1 = 2;$
5.	$if(x_1 \geq 0)\{$
6.	$\quad y_2 = y_1 + 2;$
7.	$\quad z_2 = y_2 + x_1;\}$
	else{
8.	$\quad y_3 = y_1 + 5;$
9.	$\quad z_3 = y_3 + x_1\};$
ϕ_1.	$(y_4, z_4) = f((y_2, z_2), (y_3, z_3))$
10.	$print(y_4, z_4);$
11.	Stop

(a) Program P

1.	Start
2.	$x_1 = input;$
3.	$y_1 = 5;$
4.	$z_1 = 2;$
5.	$if(x_1 \geq 0)$
p_1.	$\quad skip;$
	else{
9.	$\quad z_3 = y_1 + x_1\};$
ϕ_1.	$(y_4, z_4) = f((y_1, z_1), (y_1, z_3))$
10.	$print(y_4, z_4);$
11.	Stop

(b) Program P After computing semantic relevancy *w.r.t. SIGN*



(c) *Semantics−based Abstract PDG of P after computing Semantic Relevancy w.r.t. SIGN*

Figure 6: Treatment of "if-else" conditional block

between w and y. Observe that although w is invariant *w.r.t.* the evaluation of the guard, this is not captured by [20].

The semantic relevancy over statement level can resolve this issue of independency. Let us denote the complete "$if - else$" block by s. The semantics of s says that $\forall \sigma_1, \sigma_2 \in \Sigma$ appearing at program point 5, $P[\![s]\!](\sigma_1) = \sigma_1'$ and $P[\![s]\!](\sigma_2) = \sigma_2'$ implies $\sigma_1' = \sigma_2'$. It means that there is no control of the "$if - else$" over the resultant state which is invariant. So we can replace the complete conditional block by the single statement $w = 5$.

Refined Abstract PDGs

We are now in position to formalize the algorithm to construct semantics-based (abstract) PDG $G^{r,d}_{pdg}$ of a program P, as shown in Figure 7. We use the notation s_i to denote i^{th} statement at program point p_i in the program P. The notation ε_{ij} represents the j^{th} abstract state appearing at program point p_i *w.r.t.* property ρ. The input of the algorithm is the program P and output is the semantics-based (abstract) PDG.

Step 3 computes the semantic relevancy *w.r.t.* property ρ for all the statements of the program and at step 6 inserts nodes corresponding to the semantically relevant statements except the conditional (if, $if - else$) and repetitive (*while*) statements. Steps 9, 10, 11 and 12 deal with how to insert a node into

Algorithm 1: REFINE-PDG

Input: Program P

Output: Semantics-based (Abstract) PDG $G^{r,d}_{pdg}$

1. FOR each statement s_i DO
2. FOR all $\varepsilon_{ij} \in \Sigma^{\rho}$ DO
3. Compute Semantic relevancy of s_i;
4. END FOR
5. IF s_i is semantically relevant and is not conditional/repetitive statement THEN
6. Insert the node corresponding to s_i into the PDG;
7. END IF
8. END FOR
9. FOR each "$if-else$" conditional statement DO
10. Case 1: "if" block is semantically irrelevant but has semantically relevant "*else*" block: insert a node into the PDG corresponding to the statement "*skip*" which replaces all statements in the "if" block, and insert a node into the PDG corresponding to the "$if-else$" conditional statement, and obtain the control dependencies;
11. Case 2: "if" block is semantically relevant but has semantically irrelevant "*else*" block: insert a node into the PDG corresponding to the "if"conditional statement only and disregard the "*else*" block completely and obtain the control dependencies;
12. Case 3: Both "$if-else$" blocks are semantically relevant: insert a node into the PDG corresponding to the "$if-else$" conditional statement and obtain the control dependencies;
13. END FOR
14. FOR each "*while*" repetitive or "if" conditional statement DO
15. Case 1: "*while*" block is semantically relevant: insert a node into the PDG corresponding to the repetitive statement "*while*" and obtain the control dependencies;
16. Case 2: "if" block is semantically relevant: insert a node into the PDG corresponding to the "if" conditional statement and obtain the control dependencies;
17. END FOR
18. Apply expression level semantic dependency computation to obtain semantic data dependencies;

Figure 7: Algorithm to generate precise semantics-based (abstract) PDG

the PDG corresponding to the "$if-else$" conditional statement based on the semantic relevancy of the "if" and "*else*" blocks and follow the rules as discussed before. Steps 14, 15 and 16, similarly, deal with how to insert a node into the PDG corresponding to the "if" conditional statement and "*while*" repetitive statement based on the semantic relevancy of the corresponding blocks. Finally, semantic data dependencies are included into the PDG at step 18. 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 Graphs (DCGs)

In this section, we extend the semantics-based (abstract) PDG into a Dependence Condition Graph DCG [28]. A DCG is built from a PDG by annotating each edge e of the PDG with information e^b whose semantic interpretation encodes the condition for which the dependence represented by that edge actually arises in a program execution. The annotation e^b on any edge e (from $e.src$ to $e.tgt$) is a pair $\langle e^R, e^A \rangle$. The first component e^R is referred to as Reach Sequence, and represents the condition that should be true for an execution to ensure that the target $e.tgt$ of e is executed once the source $e.src$ of e is executed, if e is a control edge, or that $e.tgt$ is reached from $e.src$ if e is a data edge. The component e^A is referred to as Avoid Sequence which is only relevant for data edges (for control edges it is $\emptyset$), and captures the possible condition under which the assignment at $e.src$ can be overwritten before it reaches to $e.tgt$.

Sukumaran et al. [28] also described the semantics of DCG in terms of execution semantics of the program over concrete domain. This concrete semantics of the DCG can easily be lifted to the abstract domain with respect to the property of interest. We skip the details of the abstract semantics of the DCG for brevity. The (abstract) semantics of the DCG says about the condition under which any execution ψ over an abstract domain *w.r.t.* property ρ satisfies any PDG path $\eta = e_1 e_2 \dots e_n$, $n \geq 1$ (written as $\psi \vdash \eta$). For any ϕ-sequence η [28], any execution ψ over an abstract domain will satisfy η if the abstract value computed at $e_1.src$ reaches to $e_n.tgt$ and has an impact on it in ψ.

Theorem 1 *For a ϕ-sequence $\eta = e_1 e_2 \dots e_n$ and execution ψ over an abstract domain, we say that $\psi \vdash \eta$ if the abstract value computed at $e_1.src$ reaches to $e_n.tgt$ and has impact on the abstract value of $e.tgt$ in ψ.*

Consider a semantics-based (abstract) PDG $G^{r,d}_{pdg}$ with the DCG annotations over the data/control edges. If we are given a slicing criterion $\langle p, v \rangle$, the corresponding sub-PDG $G^{r,d}_{s}$ can be obtained by applying the PDG-based slicing techniques [23] on $G^{r,d}_{pdg}$. This sub-PDG $G^{r,d}_{s}$ can be made more precise by removing the node t from the sub-PDG if all the data dependence paths η_t of the form of data dependence edge $d = t \xrightarrow{x} t'$ or of the form of ϕ-sequence with t as the source node in $G^{r,d}_{s}$ are semantically unrealizable under the (abstract) semantics of the DCG annotations *i.e.* $\forall \eta_t$ and $\forall \psi$: $\psi \nvdash \eta_t$. Thus, the corresponding node t will not appear in the slice. In Figure 8, we formalize the algorithm to compute the precise slice based on DCG where the input is the semantics-based (abstract) PDG $G^{r,d}_{pdg}$ with all the DCG annotations and the slicing criteria $\langle p, v \rangle$.

The results on the dependency graph discussed so far have an impact on the different forms of static slicing: backward slicing, forward slicing, and chopping. The backward slice with respect to variable v at program point p consists of the program points that affect v. Forward slicing is the dual of backward slicing. The forward slice with respect to variable v at program point p consists of the program points that are affected by v. Chopping is a combination of backward slicing 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 sink

Algorithm 2: REFINE-subPDG

Input: Semantics-based (Abstract) PDG $G^{r,d}_{pdg}$ with DCG annotations and slicing criteria $\langle p, v\rangle$
Output: (Abstract) Slice

1. Apply the PDG-based slicing technique on $G^{r,d}_{pdg}$ *w.r.t.* $\langle p, v\rangle$ and obtain the sub-PDG $G^{r,d}_s$;

2. Perform step 2(a) for all nodes t in $G^{r,d}_s$ and obtain precise sub-PDG $G^{r,d,c}_s$:

2(a). FOR all the data dependence paths η_t of the form of data dependence edge $d = t \xrightarrow{x} t'$ or of the form of ϕ-sequence with t as the source node: if $\forall\psi$, $\forall\eta_t$: $\psi \nvdash \eta_t$, remove the corresponding node t from $G^{r,d}_s$;

3. Compute the slice based on $G^{r,d,c}_s$ *w.r.t.* $\langle p, v\rangle$;

Figure 8: Slicing based on DCG

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 dependency graph is slightly restrictive in the sense that the dependency graph permits slicing of a program with respect to program point p and a variable v that is defined or used at p, rather than *w.r.t.* arbitrary variable at p. PDG-based backward program slicing is performed by walking the graph backwards from the node of interest in linear time [23]. The walk terminates at either entry node or already visited node. Thus for a vertex n of the PDG, the slice *w.r.t.* the variable v at n is a graph containing all vertices that can reach directly or indirectly to n and affect the value of v via flow or control edges.

In case of forward slicing technique based on PDG representation, similarly, we traverse the graph in forward direction rather than backward from the node of interest. We can use the standard notion of *chop* of a program with respect to two nodes s and t in slicing technique [14, 25]: $chop(s, t)$ 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 as follows [28]: $AC(s, t)$ is defined to be true if there exists at least one execution ψ over abstract domain 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 $\forall\psi$ and $\forall\eta \in chop(s, t) : \psi \nvdash \eta$ which indicates that $chop(s, t)$ is empty.

6 Conclusion

The combination of results on the refinement of dependency 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. This is the topic of our ongoing research.

Acknowledgement

Work partially supported by Italian MIUR COFIN'07 project "SOFT".

References

[1] A. Cimitile, A. De Lucia, and M. Munro. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings. of International Conference on Software Maintenance*, pages 124–133. IEEE Computer Society Washington, DC, USA, 1995.

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

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

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press.

[5] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the 29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pages 178–190, Portland, OR USA, January 16–18 2002. ACM Press.

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

[7] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.

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

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

[10] Rajeev Gopal. Dynamic program slicing based on dependence relations. In *Proceedings. Conference on Software Maintenance*, pages 191–200, Sorrento, Italy, Oct 1991.

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

[12] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of Conference on Software Maintenance*, pages 299–308, Orlando, FL, USA, Nov 1992. IEEE Computer Society.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Progr. Lang. Systems*, 12(1):26–60, January 1990.

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

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

[16] Bogdan Korel and Jurgen Rilling. Dynamic program slicing in understanding of program execution. In *5th International Workshop on Program Comprehension (WPC '97)*, pages 80–89, 1997.

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

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

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

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

[21] Markus Muller-Olm and Helmut Seidl. On optimal slicing of parallel programs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 647 – 656, Hersonissos, Greece, 2001. ACM press.

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

[23] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Softw. Eng. Symp. on Practical Softw. Dev. Environments*, pages 177–184, New York, USA, 1984. ACM Press.

[24] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, 1990.

[25] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proc 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, June 1995.

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

[27] S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proc. 21st International Conf. on Software Engineering*, May 1999.

[28] S. Sukumarana, A. Sreenivasb, and R. Metta. The dependence condition graph: Precise conditions for dependence between program points. *Computer Languages, Systems & Structures*, 36(96–121):577–581, April 2010.

[29] Mark Weiser. Program slicing. *IEEE Trans. on soft. Eng.*, SE-10(4):352–357, July 1984.

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