PROPERTY DRIVEN PROGRAM SLICING REFINEMENT

Sukriti Bhattacharya and Agostino Cortesi
Ca' Foscari University of Venice, Via Torino 155, 30170 Venezia, Italy
{*sukriti,cortesi*}*@unive.it*

Keywords: Abstract Interpretation, Program slicing, Semantics, Static analysis.

Abstract: A slice is usually computed by analyzing how the effects of a computation are propagated through the code, i.e., by inferring dependencies. The aim of this paper is to further refine the traditional slicing technique by combining it with a static analysis in Abstract Interpretation based framework. This results into a deeper insight on the strong relation between slicing and property based dependency.

1 INTRODUCTION

Program slicing is the study of meaningful subprograms. Typically applied to the code of an existing program, a slicing algorithm is responsible for producing a program (or subprogram) that preserves a subset of the original programs behavior. A specification of that subset is known as a slicing criterion, and the resulting subprogram is a slice. Generally speaking, by applying a slicing technique on a program P with a slicing criterion C (i.e. a line of code in P), we get a program P' that behaves like P when focussing only on the variables in C. The sliced program P' is obtained through backward computation from P by removing all the statements that do not affect neither directly nor indirectly the values of the variables in C.

Very often, we are interested on a specific property of the variables in the slicing criterion, not on their exact actual values.

In this direction (Bhattacharya, 2011), our aim is to further refine the traditional slicing technique, (Weiser, 1984) by combining it with a static analysis in Abstract Interpretation (Cousot and Cousot, 1977) based framework that looks for the statements affecting a fixed property of variables of interest rather than values. This results into a deeper insight on the strong relation between slicing and property based dependency (Mastroeni and Zanardini, 2008) (Mastroeni et al., 2010) (Cortesi and Halder, 2010).

The resulting proposal is a fixed point computation where each iterate has two phases. First, the control flow analysis is combined with a static analysis in a Abstract Interpretation based framework. Hence, each program point of the program is enhanced with information about the abstract state of variables with respect to the property of interest. Then, a backward program slicing technique is applied to the augmented program exploiting the abstract dependencies.

2 ABSTRACT SEMANTICS

The essential issue in program slicing is to define what semantic relationship must exist between a program and its slice in order that the slice is considered valid. Mark Weiser (Weiser, 1984) defined the semantic relationship that must exist between a program and its slice in terms of state trajectories. In this section we provide the abstract semantics of the trajectories. We consider the WHILE language for our discussion (Nielson et al., 1999). The set of concrete states Σ consists of functions $\sigma : \mathsf{V} \to \mathcal{V}$ which maps the variables to their values from the semantic domain $\mathbb{Z}_\perp$ where, $\perp$ represents an undefined or uninitialized value and $\mathbb{Z}$ is the set of integers. If a program has k variables $x_1, ..., x_k$, we can represent states as tuples, i.e., $\sigma = \langle x_1, ..., x_k \rangle$ and $\Sigma = \mathcal{V}^k$.

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

$\mathcal{D}$ be an abstract domain on concrete values and α and γ are *abstraction* and *concretization* functions, respectively. The related abstract semantics on expressions , $\mathcal{H}[\![a]\!]\varphi$, is applied to abstract states $\varphi = \langle d_1, ..., d_k \rangle \in \mathcal{D}^k$ and is defined as the best correct approximation of $\mathcal{E}[\![a]\!]\sigma$ as depicted in Table 1.

In Table 1 $\widehat{op_a}$ is the abstract operation in $\mathcal{D}$ that

Table 1: Approximation of arithmetic expressions.

$$\mathcal{H}[\![a]\!]\varphi = \begin{cases} d_i & if \quad a = x_i \in \mathsf{Var} \\ \alpha(n) & if \quad a = n \in \mathsf{Num} \\ \mathcal{H}[\![a_1]\!]\varphi \ \widehat{op_a} \ \mathcal{H}[\![a_2]\!]\varphi & if \quad a = a_1 \ op_a \ a_2 \end{cases}$$

safely approximate op_a, when we construct the abstract semantics of programs, we need to define abstract operations over the abstract domain, that approximate the corresponding concrete operations over the concrete domain. The idea is that the abstract calculation *simulates* the concrete calculation, and the concretization of the abstract calculation is a correct approximation of the values in the concrete result.

For example consider the following code fragment in Figure 1 and consider the abstract domain where the addition and multiplication are influenced according to the well known *rule of signs*

```
1. x=2;
2. y=-5;
3. z = (x+3)*y;
```

Figure 1: Sample code fragment.

The sign of the variable z can be computed in the abstract domain of *Sign* by,

$$\begin{array}{lcl} \mathcal{H}[\![x+3*y]\!]\varphi & = & (\mathcal{H}[\![x]\!]\varphi \mathbin{\widehat{+}} \mathcal{H}[\![3]\!]\varphi) \mathbin{\widehat{*}} \mathcal{H}[\![y]\!]\varphi \\ & = & (+ \mathbin{\widehat{+}} \alpha(3)) \mathbin{\widehat{*}} - \\ & = & (+ \mathbin{\widehat{+}} +) \mathbin{\widehat{*}} - \\ & = & + \mathbin{\widehat{+}} - \\ & = & - \end{array}$$

The abstract semantics $\mathcal{H}_b[\![b]\!]\varphi$ of boolean expression b is defined as the best correct approximation of $\mathcal{B}[\![b]\!]\sigma$ in Table 2, where $\widehat{op_r} : \mathcal{D} \times \mathcal{D} \to \{TRUE, FALSE, ?\}$ is the abstract operation that safely approximate op_r and ? ($undefined$) signifies that, the abstract domain is not accurate enough to evaluate the condition. For instance, abstract operations $\widehat{<}$ on *Sign* domain is depicted in Table 3,

Table 2: Approximation of boolean expressions.

$$\mathcal{H}_b[\![b]\!]\varphi = \begin{cases} TRUE & if \quad b = TRUE \\ & OR \\ & b = a_1 \ op_r \ a_2 \ \ AND \\ & \mathcal{H}[\![a_1]\!]\varphi \ \widehat{op_r} \ \mathcal{H}[\![a_2]\!]\varphi = TRUE \\ FALSE & if \quad b = FALSE \\ & OR \\ & b = a_1 \ op_r \ a_2 \ \ AND \\ & \mathcal{H}[\![a_1]\!]\varphi \ \widehat{op_r} \ \mathcal{H}[\![a_2]\!]\varphi = FALSE \\ ? & \text{undefined otherwise} \end{cases}$$

Table 3: Abstracting $<$ operator.

$\widehat{<}$	$\top$	$\bot$	$+$	0	$-$
$\top$	?	?	?	?	?
$\bot$	?	?	?	?	?
$+$	?	?	?	FALSE	FALSE
0	?	?	TRUE	?	FALSE
$-$	?	?	TRUE	TRUE	?

3 ABSTRACT TRAJECTORY

We now define the abstract trajectory semantics for WHILE.

- For *skip* statement:
  $\tau^{\mathcal{D}}[\![l : skip]\!]\varphi = \langle (l, \varphi) \rangle$
  $\langle (l, \varphi) \rangle$ represents the singleton sequence consisting of the pair (l, φ). i.e statement level alone with the properties of the variables.
- For assignment statement:
  $\tau^{\mathcal{D}}[\![l : x = a]\!]\varphi = (l, \varphi[x \leftarrow \mathcal{H}[\![a]\!]\varphi])$
  Where $\mathcal{H}[\![a]\!]\varphi$ means the *new* value resulting from evaluating expression a in abstract domain and $\varphi[x \leftarrow \mathcal{H}[\![a]\!]\varphi]$ is the abstract state φ *updated* with the maplet that takes variable x to this new abstract value.
- For sequences of statements: $\tau^{\mathcal{D}}[\![l : S_1; S_2]\!]\varphi = \tau[\![S_1]\!]\varphi \diamond \tau^{\mathcal{D}}[\![S_2]\!]\varphi'$
  Where φ' is the abstract state obtained after executing S_1 in φ and $\diamond$ means concatenation.
- For *if* statement:

  $$\tau^{\mathcal{D}}[\![l : if \ b \ then \ S_1 \ else \ S_2]\!]\varphi = \langle (l, \varphi) \rangle \diamond$$

  $$\begin{cases} \bot & if \ \mathcal{H}_b[\![b]\!]\varphi = ? \\ \tau^{\mathcal{D}}[\![S_1]\!] & if \ \mathcal{H}_b[\![b]\!]\varphi = TRUE \\ \tau^{\mathcal{D}}[\![S_2]\!] & if \ \mathcal{H}_b[\![b]\!]\varphi = FALSE \\ (\tau^{\mathcal{D}}[\![S_1]\!]\varphi) \sqcup (\tau^{\mathcal{D}}[\![S_2]\!]\varphi) & \text{otherwise} \end{cases}$$

  The first element is the label of the *if* in the current abstract state. The rest of the trajectory is the trajectory of one of the branches depending on the abstract execution of the boolean expression evaluated in the current abstract state.
- For *while* statement:

  $$\tau^{\mathcal{D}}[\![l : while \ b \ then \ S]\!]\varphi =$$

  $$\begin{cases} \lambda & if \ \mathcal{H}_b[\![b]\!]\varphi = FALSE \\ \langle l_i, \sqcup_{i \geq 0}(\varphi_i) \rangle & otherwise \end{cases}$$

  If the predicate b evaluated to be *FALSE* there would be a empty trajectory at l other wise

a fixpont iteration on the abstract state of each statements with in the loop body where $\varphi_0 = \varphi$ and $\varphi_{i+1} = \tau^{\mathcal{D}}[\![S]\!]\varphi_i$

Definition 1. (Restriction of a state to a set of variables w.r.t a given property) Given a abstract state, φ with respect to a property, ρ and a set of variables, $\mathcal{V} \in \mathsf{Var}$, $\varphi|_V^{\rho}$ restricts φ so that it is defined by ρ only for variables in $\mathcal{V}$.

Definition 2. (Projection of a abstract trajectory to a slicing criterion w.r.t a given property) For a program point p' and a abstract state φ, the projection of the abstract trajectory sequence element (p',φ) to the slicing criterion (p,V) w.r.t property ρ is

$$(p',\varphi)|^{\rho}_{(p,V)} = \begin{cases} (p',\varphi|_V^{\rho}) & if \quad p' = p \\ \lambda & otherwise \end{cases}$$

where λ denotes the empty string.

The projection of the abstract trajectory $\tau^{\mathcal{D}}$ to the slicing criterion (p,V) w.r.t a property ρ is

$$Proj_{(p,V)}(\tau^{\mathcal{D}}) = \langle (p_0,\varphi_0)|^{\rho}_{(p,V)}, (p_1,\varphi_1)|^{\rho}_{(p,V)}, ..., (p_k,\varphi_k)|^{\rho}_{(p,V)} \rangle$$

Definition 3. (Property driven program slicing) A property driven slice P_ρ of a program P on a slicing criterion (p,V) and with respect to a given property ρ is any executable program with the following two properties:

- P' can be obtained from P by deleting zero or more statements.
- Whenever P halts on an input state φ with a abstract trajectory $\tau^{\mathcal{D}}$ then P' also halts on φ with trajectory $\tau^{\mathcal{D}'}$ where,

$$\mathscr{R}ed(Proj_{(p,V)}(\tau^{\mathcal{D}})) = \mathscr{R}ed(Proj_{(p,V)}(\tau^{\mathcal{D}'})).$$

Where $\mathscr{R}ed$ is defined in Table 4, given a abstract trajectory $\tau^{\mathcal{D}} = \langle (p_0,\varphi_0),(p_1,\varphi_1),...,(p_k,\varphi_k)\rangle$ $\mathscr{R}ed$ is obtained by applying the following reduction algorithm,

Table 4: $\mathscr{R}ed$.

```
begin
i=0;
while(i < n){
  j=1;
  while(p_{i+j} = p_i)  &&  (φ_{i+j} = φ_i)
    remove (p_{i+j}, φ_j) from the trajectory
  i=i+j;
}
```

Table 5: Property driven slicing on *Sign*.

St.No.	Original Program	Sliced Program
1	$x = 5;$	$x = 5;$
2	$y = 3;$	$y = 3;$
3	$z = y - x;$	$z = y - x;$
4	$if(x > z)\{$	$if(x > z)$
5	$y = x + z^2;$	
6	$w = y * z;\}$	$w = y * z;$
7	$else\{$	
8	$y = x^2 + z;$	
9	$w = y * z;\}$	
10	$printf("\%d", w);$	$printf("\%d", w);$

Consider Table 5 for an illustration of the above definitions,

The abstract state trajectory of program P with respect to *Sign* property is denoted as τ^{Sign} and the abstract state trajectory of the sliced program P_{Sign} with respect to the property *Sign* on slicing criteria $C = (10,w)$ is denoted as $\tau^{Sign'}$.

$\tau^{Sign} =$ $\langle (1,\{\bot,\bot,\bot,\bot\}),\ (2,\{\bot,+,\bot,\bot\}),\ (3,\{\bot,+,-,\bot\}),$
$(4,\{\bot,+,-,-\}),\ (5,\{\bot,+,-,-\}),\ (6,\{-,+,+,-\}),$
$(10,\{-,+,+,-\})\rangle$

$\tau^{Sign'} =$ $\langle (1,\{\bot,\bot,\bot,\bot\}),\ (2,\{\bot,+,\bot,\bot\}),\ (3,\{\bot,+,-,\bot\}),$
$(5,\{\bot,+,-,-\}),\ (6,\{\bot,+,+,-\}),\ (10,\{-,+,+,-\})\rangle$

Notice that,

$$\mathscr{R}ed(Proj_{(10,w)}(\tau^{Sign})) = \mathscr{R}ed(Proj_{(10,w)}(\tau^{Sign'}))$$

4 DATAFLOW BASED PROPERTY DRIVEN PROGRAM SLICING

This notion of dependencies often loses some information, because syntactic occurrence is not enough to get the real idea of relevancy. For instance (Mastroeni and Zanardini, 2008), the value assigned to x does not depend on y in the statement $x = z + y - y$, although y occurs in the expression. The syntactic approach may fail in computing the optimal set of dependencies, since it is not able to rule out this kind of *false dependencies*. This results in obtaining a slice which contains more statements than needed.The first step towards a generalization of the way of defining slicing is to consider *semantic dependencies*, where intuitively a variable is relevant for an expression if it is relevant for its evaluation.

Definition 4. (Semantic dependency) Let $x,y \in \mathsf{Var}$, then the semantic dependency between the expression e and variable x is defined formally as,

$\exists \sigma_1,\sigma_2 \in \Sigma. \forall y \neq x. \sigma_1(y) = \sigma_2(y) \wedge$
$\mathcal{E}[\![e]\!]\sigma_1 \neq \mathcal{E}[\![e]\!]\sigma_2$.

This semantic notion can then easily generalized

in what we will call *abstract dependency*, where a variable is relevant to an expression if it affects a given property of its evaluation. More precisely, This notion of dependency is parametric on the properties of interest. Basically, an expression e depends on a variable x w.r.t. a property ρ if changing x, and keeping all other variables unchanged with respect to ρ, may lead to a change in e with respect to ρ.

Definition 5. (Abstract dependency) Let $x,y \in \mathsf{Var}$, then the abstract dependency between the expression e and variable x with respect to an abstract domain ρ (property) is defined formally as,

$\exists \varphi_1, \varphi_2 \in \Sigma^\rho . \forall y \neq x . \varphi_1(y) = \varphi_2(y) \wedge$
$\mathcal{H}[\![e]\!]\varphi_1 \neq \mathcal{H}[\![e]\!]\varphi_2$.

Dataflow based property driven program slicing is a fixed point computation where each iterate has two phases, first, the control flow analysis is combined with a static analysis in a abstract interpretation based framework. Hence, each program point of the program is enhanced with information about the abstract state of variables with respect to the property of interest. Then, a backward program slicing technique is applied to the augmented program exploiting the abstract dependencies.

4.1 Phase 1: Static Analysis

Our representation of programs are *def/use* graphs. The objective of a static analysis based on Abstract Interpretation is to assign sets of possible abstract values to edges of a *def/use* graph. The *def/use* graph consists of five different node types which represent program points:

1. A designated start and end node representing the beginning and end point of a *def/use* graph.
2. Expression nodes representing different expression types found in a concrete semantic model.
3. Condition nodes representing forks in a control flow, i.e. this type of nodes has one incoming and two outgoing edges.
4. Join nodes merging two paths of the *def/use* graph, i.e. these nodes have two incoming and one outgoing edge.

Like the classical approach, our analysis also begins at the start node of the *def/use* graph and traverses the graph during its static program analysis phase. Depending on the encountered node type, a particular set of rules which is based on Abstract Interpretation is applied.

Based on the *def/use* graph, the classical approach begins with the construction of a complete transition system for the five node types. It defines how an abstract state is transferred into one state to another state at program point p:

$$\mathscr{T}_p : \wp(\Sigma^\mathcal{A}) \rightarrow \wp(\Sigma^\mathcal{A})$$

The transition system $\mathscr{T}$ is used to construct a system of equations which define the assignment of abstract states to program points. A solution is found by a *fixed-point* iteration. It begins with the least possible assignment $\mathscr{T}(\bot)$ where $\bot$ is the least element representing $\emptyset$. The *fixed-point* iteration continues as long as a further application of $\mathscr{T}$ does not compute a new state: $\mathscr{T}^{n-1} = \mathscr{T}^n$.

Now we will define $\mathscr{T}$ for the different types of edges in a *def/use* graph. For any edge $\mathfrak{e} \in E$ we shall denote its predecessor edges as $\mathfrak{e}_{pre}$. For merge nodes, which have two incoming edges, the second is denoted $\mathfrak{e}_{pre'}$. In the following, $\mathscr{T}$ is given for every type of program point with respect to a given abstract domain ρ. $\forall \varphi_\rho \in \Sigma^\rho$ denotes the abstract states associated to program variables at each program point.

Start Edge. At the start edge $\mathfrak{e}$, nothing is known about the values of variables. Having said this, the natural definition of an abstract state associated with the initial state should be as follows:

$$\mathscr{T}_\mathfrak{e}(\varphi_\rho) = \bot$$

Assignment Edge. An assignment edge is an edge which emerges from an assignment node. Let, an assignment node has an assignment $x := a$ associated with it, where $x \in \mathsf{Var}$ and $a \in \mathsf{AExp}$, then $\mathscr{T}_\mathfrak{e}(\varphi_\rho)$ should be equal to the previous abstract state with the variable x updated to the abstract value of e (Table 1), as follows:.

$$\mathscr{T}_\mathfrak{e}(\varphi_\rho) = \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho[x \leftarrow \mathcal{H}[\![a]\!]\varphi_\rho])$$

Merge Edge. The problem of Abstract Interpretation is that a termination of the fixed-point iteration can not be guaranteed. Due to the nature of Abstract Interpretation which iteratively simulates each state transition, the fixed-point iteration can consume a significant amount of time for loops with large iteration counts. To overcome both problems, the widening operator ∇ (Cortesi and Zanioli, 2010) can be applied. Its application typically enlarges the abstract states during the fixed-point iteration leading to a correct but also over-approximated solution which might become infeasible as result for many applications. Thus, a narrowing operator $\triangle$ was introduced (Cortesi and Zanioli, 2010) to restrict the over-approximation afterwards.

A merge edge is an edge emerging from a merge node. A merge node combines the analysis results of the two incoming edges. The least abstract value which is correct with respect to both

incoming values is the supremum of the these. In addition, if the merge node is the entry of a loop, then that is a good place to put the widening based on the abstract domain. Thus, the abstract transition function for merge nodes is

$$\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \begin{cases} \mathscr{T}_{\mathfrak{e}}(\varphi_\rho) \nabla (\mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \sqcup \mathscr{T}_{\mathfrak{e}_{pre'}}(\varphi_\rho)) \\ if\ loop\ merge \\ \\ \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \sqcup \mathscr{T}_{\mathfrak{e}_{pre'}}(\varphi_\rho) \\ otherwise \end{cases}$$

Conditional Edges. The conditional node has two outgoing edges. Conditionals are resolved by only boolean expressions with relational operators Op_r, so for an abstract domain it is necessary to have abstract version of all relational operators $\widehat{Op_r}$ (Table 2).

$$\mathscr{T}_{\mathfrak{e}}(\varphi_\rho) = \begin{cases} \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \wedge \mathscr{H}_b[\![b]\!]\varphi_\rho = TRUE \\ \\ \mathscr{T}_{\mathfrak{e}_{pre}}(\varphi_\rho) \wedge \mathscr{H}_b[\![b]\!]\varphi_\rho = FALSE \end{cases}$$

The Abstract Interpretation may establish certain properties of a program through which we can identify infeasible statements of the program which will not be taken into account for program execution by predicting predicates present in conditional statements. By the following rules we modify the program P in order to simplify the control dependence, taking into account only the statements that have impact on the property of interest ρ.

Table 6: Rules for conditional nodes.

Rule 1 For S::= l : $if\ b\ then\ S_1\ else\ S_2$

$$\begin{cases} (a) P' = P[S\ /\ S_1] & if\ \mathscr{H}_b[\![b]\!]\varphi_\rho = TRUE \\ (b) P' = P[S\ /\ S_2] & if\ \mathscr{H}_b[\![b]\!]\varphi_\rho = FALSE \\ (c) P' = P[S\ /\ S] & \text{No replacement otherwise} \end{cases}$$

Rule 2 For S::= l : $while\ b\ do\ S_1$

$$\begin{cases} (a) P' = P[skip\ /\ S] & if\ \mathscr{H}_b[\![b]\!]\varphi_\rho = FALSE \\ (b) P' = P[S\ /\ S] & No \text{ replacement otherwise} \end{cases}$$

Let's apply the rules in Table 6 on the following code fragments, In Table 7, P' is obtained by applying rule 1(a) on P by statically analyzing the program in *Parity* domain. Notice P' contains less statements than P. Therefore, the above rules can often generate a reduced CFG by statically analyzing the associated program with respect to a certain property ρ.

Table 7: Application of rule 1(a) on program P.

P	$\varphi\{w,x,y,z\}$	P'
Input z;	$(\bot, \bot, \bot, \bot)$	*Input z*;
$y = 15$;	$(\bot, \bot, \bot, \top)$	$y = 15$;
$x = 2 * z$;	$(\bot, \bot, O, \top)$	$x = 2 * z$;
$if(x! = y)$	$(\bot, E, O, \top)$	
$w = x + y$;	$(\bot, E, O, \top)$	$w = x + y$;
else	$(\bot, E, O, \top)$	
$w = x - y + 1$;	$(\bot, E, O, \top)$	
Output w;	$(\top, E, O, \top)$	*Output w*;

4.2 Phase 2: Slicing Algorithm

This section introduces a backward slicing algorithm that uses the extracted information from *phase* 1 at each program point. While traditional slicing algorithms are typically syntactical dependency based, this property driven approach must rely on semantics dependencies and abstract dependencies. In fact, the more abstract the property, the greater the loss of precision of the syntactic approach with respect to the actual semantic.

Algorithm: Property Driven Program Slicing
Input:

- ☐ $\mathcal{G}_P$: Statically analyzed (Phase 1) def/use graph of the program P.
- ☐ $C = (n, V)$: slicing criterion.
- ☐ ρ: Given property of interest.

Directly Relevant Variables ($R^0_{(C,\rho)}$)

- ☐ The set of directly relevant variables at slice node, n, is simply the slice set, V.
- ☐ The set of directly relevant variables at every other node i, is defined in terms of the set of directly relevant variables of all nodes j leading directly from i to j ($i \longrightarrow_{\mathcal{G}_{P'}} j$) in $\mathcal{G}_P$. $R^0_{(C,\rho)}(i)$ contains all variables x such that, either,

$$\begin{cases} (a)\ x \in R^0_{(C,\rho)}(j) - \mathrm{def}(i) \\ \\ (b)\ if\ (\mathrm{def}(i) \cap R^0_{(C,\rho)}(j) \neq \emptyset)\ then \\ \quad (\forall y \neq x \in use(i)) \wedge (\forall \varphi^i_\rho, \varphi^j_\rho \in \Sigma^{\mathfrak{A}}) \\ \quad\quad if(\varphi^i_\rho(y) = \varphi^j_\rho(y)) \wedge (\varphi^i_\rho(\mathrm{def}(i)) \neq \varphi^j_\rho(\mathrm{def}(i))) \\ \quad then \\ \quad\quad R^0_{(C,\rho)}(i) = R^0_{(C,\rho)}(i) \cup \{x\} \end{cases}$$

The directly relevant variables of a a node are the set of variables at that node upon which the slicing

Table 8: Program P after Phase 1.

Stmt. No.	Code	x	y	l	p	m	k	c	w
1	scanf(”%d”, &y);	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
2	x=2*y+1	$\perp$	$\top$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
3	l=x+1;	O	$\top$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
4	p=l;	O	$\top$	E	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
5	m=x+l;	O	$\top$	E	E	$\perp$	$\perp$	$\perp$	$\perp$
6	k= m+(x%2)-m;	O	$\top$	E	E	O	$\perp$	$\perp$	$\perp$
7	if(k!=0){	O	$\top$	E	E	O	O	$\perp$	$\perp$
8	x=p+1;	O	$\top$	E	E	O	O	$\perp$	$\perp$
9	x=x+1;	O	$\top$	E	E	O	O	$\perp$	$\perp$
10	c=x+p;}	E	$\top$	E	E	O	O	$\perp$	$\perp$
	else{	E	$\top$	E	E	O	O	E	$\perp$
11	x=x-1;								
12	p=l+1;								
13	c=x-p;}								
14	w=x+p	E	$\top$	E	E	O	O	E	$\perp$
15	printf(”%d”, c);	E	$\top$	E	E	O	O	E	E
16	printf(”%d”, w);	E	$\top$	E	E	O	O	E	E

criterion is transitively dependent based on a given property ρ.

Directly Relevant Statements ($S^0_{(C,\rho)}$)

In terms of the directly relevant variables, a set of *directly relevant statements* is defined:

$if\ (\text{def}(i) \cap R^0_{(C,\rho)}(j) \neq \emptyset)\ then$

$S^0_{(C,\rho)} = S^0_{(C,\rho)} \cup \{i\}$

Indirectly Relevant Variables ($R^{k+1}_{(C,\rho)}, K \geq 0$)

In calculating the indirectly relevant variables, control dependency is taken into account.

for each predicate node b in $\mathcal{G}_P{}'$ do

$if\ (b \cap S^0_{(C,\rho)}) \neq \emptyset$

$B^K_{(C,\rho)} = B^K_{(C,\rho)} \cup \{b\}$

$B^K_{(C,\rho)}$ is the set of all predicate nodes that control a statement in $S^0_{(C,\rho)}$.

$R^{K+1}_{(C,\rho)}(i) = R^K_{(C,\rho)}(i) \cup \bigcup_{b \in B^K_{(C,\rho)}} R^0_{(b,use(b),\rho)}(i)$

Indirectly Relevant Statements ($S^{k+1}_{(C,\rho)}, K \geq 0$)

Adding predicate nodes to $S^0_{(C,\rho)}$ includes further indirectly relevant statements in the slice:

$if\ (def(i) \cap R^{k+1}_{(C,\rho)}(j) \neq \emptyset)\ then$

$S^{k+1}_{(C,\rho)} = S^k_{(C,\rho)} \cup B^K_{(C,\rho)} \cup \{i\}$

Let us consider the following code in Table 8. Notice that, statements 7 and 11 to 13 can be ignored by Rule 1(a) discussed in Table 6.

Table 9 shows the comparison between the value based slice and property driven slice with respect to slicing criterion C=(16, w) and a property $\rho = Parity$. Since the property of x at statement 2 does not depend on the property of y, statement 1 is irrelevant. The property of x stays same before and after the execution of statement 8, for that reason statement 8 is also irrelevant in this context. And statement 6 and statement 10 are deleted from the slice due to the traditional slicing rules.

5 CONCLUSIONS

The proposed slicing algorithm does not allow any huge alteration on the traditional algorithm, it just emphasizes on the abstract dependencies rather than on value based dependencies and has some significant advantages over the traditional slicing algorithms.

On the practical side, property driven program slicing is interesting since, in general, the slicing based on a property of some variables is smaller than the slicing technique based on the exact value of the same variables, since, properties propagate less than concrete values, some statements might affect the values but not the property. This can make debugging and program understanding tasks easier, since a smaller portion of the code has to be inspected when searching for some undesired behavior.

ACKNOWLEDGEMENTS

Work partially supported by RAS L.R. 7/2007 Project TESLA.

Table 9: Property driven slice of P, $P^{Parity}_{(16,w)}$, w.r.t $\rho = Parity$ and C=(16,w), and value based slice of P, $P_{(16,w)}$, w.r.t $\rho = Parity$ and C=(16,w).

Stmt. No.	P	$P^{Parity}_{(16,w)}$	$P_{(16,w)}$
1	scanf(”%d”, &y);		scanf(”%d”, &y);
2	x=2*y+1;	x=2*y+1;	x=2*y+1;
3	l=x+1;	l=x+1;	l=x+1;
4	p=l;	p=l;	p=l;
5	m=x+l;		m=x+l;
6	k= m+(x%2)-m;		k= m+(x%2)-m;
7	if(k!=0){		if(k!=0){
8	x=p+1;		x=p+1;
9	x=x+1;	x=x+1;	x=x+1;}
10	c=x+p;}		
	else{		else{
11	x=x-1;		x=x-1;
12	p=l+1;		p=l+1;}
13	c=x-p;}		
14	w=x+p;	w=x+p;	w=x+p;
15	printf(”%d”, c);		
16	printf(”%d”, w);	printf(”%d”, w);	printf(”%d”, w);

REFERENCES

Bhattacharya, S. (2011). Property driven program slicing and water marking in the abstract interpretation framework. *PhD Thesis, Ca' Foscari University of Venice, Italy*.

Cortesi, A. and Halder, R. (2010). Dependence condition graph for semantics-based abstract program slicing. *In proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, ACM Press*, (1):4–17.

Cortesi, A. and Zanioli, M. (2010). Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems and Structures*, 37(1):24–42.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM Symp. on Principles of Programming Languages*, pages 238–252.

Mastroeni, I., , and Nikolic'., D. (2010). Abstract program slicing: From theory towards an implementation. *Formal Methods and Software Engineering, LNCS 6467*, pages 452–456.

Mastroeni, I. and Zanardini, D. (2008). Data dependencies and program slicing: from syntax to abstract semantics. *Proceedings of ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 123–134.

Nielson, F., Nielson, H., and Hankin, C. (1999). Principles of program analysis. *Springer Verlag*.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.