Abstract Program Slicings *

IN SANG CHUNG
School of Computer Engineering
Hansung University, Seoul, Korea
insang@hansung.ac.kr

BYEONG-MO CHANG
Department of Computer Science,
Sookmyung Women's University, Seoul, Korea
chang@cs.sookmyung.ac.kr

BYEONG MAN KIM
School of Computer & Software Engineering,
Kumoh National University of Technology, Kumi, Korea
bmkim@se.kumoh.ac.kr

JANG-WU JO
Department of Computer Engineering
Pusan University of Foreign Studies, Pusan, Korea
jjw@taejo.pufs.ac.kr

Abstract

In this paper, we present a new slicing technique named abstract program slicing that allows a decomposition of a program for the set of initial states. We apply abstract interpretation to the derivation of slices from existing programs. Abstract interpretation allows us to yield safe information about the run-time behavior of the program without having to run it for all input data. Thus, we can statically compute safe approximations of program slices on the slicing criterion.

KEY WORDS
Software Engineering, Program Slicing, Abstract Interpretation

1. Introduction

Program slicing has been used to provide solutions to many software engineering areas including reverse engineering, testing, debugging, reuse and complexity analysis. Program slicing is carried out with respect to a slicing criterion (p, V), where p is a program location and V is a subset of the program's variables. The major goal of program slicing is to extract those statements which directly or indirectly affect the values of variables at the slicing criterion [1].

Even though various slightly different notions of program slices have been proposed, static slice and dynamic slice provide the basis for many different definitions of slices proposed in the literature. The difference between static and dynamic slicing is that *dynamic* slicing assumes a fixed input for a program, whereas static slicing does not make assumptions regarding the input. A dynamic slicing criterion specifies the input, and distinguishes between different occurrences of a statement in the execution history [2, 3]. The availability of run-time information makes dynamic slices smaller than static slices.

Another interesting slicing definition has been introduced in the literature, called *conditioned slicing* [4]. Conditioned slicing is a generalization of both static and dynamic slicing. The conditioned slicing criterion augments the static criterion with an input condition, which captures a set of initial states. In [5], symbolic execution and theorem proving techniques are used in order to compute a conditioned slice. Using symbolic execution, state information from the slicing criterion is propagated to all points in the program. This information is passed to the theorem prover, which identifies the statements which are never executed under the condition imposed by the slicing criterion. However, the computed conditioned slice can be overly conservative in cases where the theorem prover is not able to decide the truth of propositions put to it.

In general, computing *exact* program slices on a slicing criterion involves the semantic analysis of programs such as determination of a suitable induction hypothesis or fixed point calculations of the invariant conditions associated with each program point. It is often the case that these tasks need human interaction or could be time-consuming although tools such as theorem provers are employed [6, 7].

In this paper, we present a novel slicing technique named *abstract program slicing* that produces *approximate* conditioned program slices. Abstract program slicing, however, does not rely on theorem proving in reasoning about the effect of the conditions mentioned in the slicing criterion. Instead, we apply abstract interpretation to the derivation of slices. Abstract interpretation allows us to yield "safe" information about the run-time behavior of the program without having to run it for all input data [8]. The calculation will give approximate information, while guaranteeing that the information is *safe*. By "safe", we mean that the results of all possible real executions of the program are included in the calculated results. This characteristic of abstract interpretation enables us to compute

*This work was supported in part by the Ministry of Information and Communication of Korea(Support Project of University Research '2000 supervised by IITA), KISTEP(grant No. 39-N6-02-01-A-02) and the Basic Research Program of the Korea Science & Engineering Foundation(grant No. 2000-1-30300-009-2) .

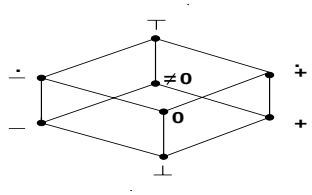

Figure 1. A lattice of abstract values for sign analysis

slices which are safe approximations of program slices on the conditioned slicing criterion.

2. Abstract Interpretation

Abstract interpretation focuses on a class of properties of program executions, which is usually defined by a *collecting semantics*. Intuitively, the collecting semantics is the most precise semantics to provide a sound and complete proof method for the class of properties of interest. Thus, it can be conceived as a reference semantics which provides a basis for proving the soundness of all other approximate or abstract semantics. The correspondence between collecting semantics and abstract semantics can be expressed with the use of Galois connections, denoted by a pair of monotone functions (α, γ), where there is a best way to approximate any concrete property by an abstract one.

Suppose that we have a concrete domain $(D, \sqsubseteq)$, which is a lattice. Then, we define an abstract domain $(D^{\mathcal{A}}, \sqsubseteq^{\mathcal{A}})$ approximating the concrete domain, where $d_1^{\mathcal{A}} \sqsubseteq^{\mathcal{A}} d_2^{\mathcal{A}}$ means that $d_1^{\mathcal{A}}$ is a more precise value than $d_2^{\mathcal{A}}$. The purpose of the abstraction function $\alpha : D \to D^{\mathcal{A}}$ is to map elements in D to approximating elements in $D^{\mathcal{A}}$, while respecting the partial order. The fact that $d^{\mathcal{A}}$ is a valid approximation of d can be expressed as: $\alpha(d) \sqsubseteq^{\mathcal{A}} d^{\mathcal{A}}$. The concretization function $\gamma : D^{\mathcal{A}} \to D$ is a meaning function, which maps an abstract value in $D^{\mathcal{A}}$ to its concrete meaning in D. This mapping also should respect the partial order. The fact that $d^{\mathcal{A}}$ is a valid approximation of d can also be stated as: $d \sqsubseteq \gamma(d^{\mathcal{A}})$.

When these two soundness conditions are equivalent, we have a Galois connection which is defined as a pair of functions (α, γ) such that the partial order is preserved while going back and forth between the two lattices. The intuition behind Galois connection is that the best such abstract approximation is defined by α. Formally, a *Galois connection* between D and $D^{\mathcal{A}}$ is a pair (α, γ) of monotone functions $\alpha : D \to D^{\mathcal{A}}$ and $\gamma : D^{\mathcal{A}} \to D$ such that

$$\forall d \in D : \forall d^{\mathcal{A}} \in D^{\mathcal{A}} : \alpha(d) \sqsubseteq^{\mathcal{A}} d^{\mathcal{A}} \Leftrightarrow d \sqsubseteq \gamma(d^{\mathcal{A}})$$

For example, an abstract interpretation may use abstract values $+$, $-$ and 0 to describe negative and positive integers, and zero rather than using concrete integer values.

When we construct the approximate or abstract semantics of programs, we need to define abstract operations over the abstract domain, that approximate 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, by abstracting operations like addition or multiplication according to the "rule of signs", the abstract interpretation may establish certain properties of a program such as"whenever this loop body is entered, a variable x is assigned a positive value".

The concrete collecting semantics of a program P is characterized by the least fixpoint $lfp(F_P)$ of the semantic function $F_P : \wp(State) \to \wp(State)$ where $State$ is the set of all possible program states. The concrete collecting semantics can be infinite, so we compute a safe approximation (that is, abstract semantics) of the concrete semantics by abstract interpretation. The abstract semantics of a program P is characterized by the least fixpoint $lfp(F_P^{\mathcal{A}})$ of $F_P^{\mathcal{A}}$, and it is a safe approximation of the concrete semantics if $F_P^{\mathcal{A}}$ is a safe approximation of F_P. This can be illustrated by a Galois connection $(\alpha_\sigma, \gamma_\sigma)$ between the concrete domain $(\wp(State), \subseteq)$ and the abstract domain $(AbsState, \sqsubseteq)$, where $State$ and $AbsState$ represent a set of possible concrete states and a set of abstract states, respectively. Once an abstract domain $AbsState$ is selected, we can derive the abstract semantic function $F_P^{\mathcal{A}}$ from the concrete one. The abstract semantic function can be built-up on a standard way from the abstract versions of primitive operations, which are program independent. Then, it is well known that if an abstract semantic function $F_P^{\mathcal{A}}$ is a safe approximation : $\alpha_\sigma \circ F_P \circ \gamma_\sigma \sqsubseteq F_P^{\mathcal{A}}$ of a semantic function F_P, then the abstract semantics of P defined by the least fixed point of $F_P^{\mathcal{A}}$ is a safe approximation of the the concrete collecting semantics, that is: $lfp(F_P) \subseteq \gamma_\sigma(lfp(F_P^{\mathcal{A}}))$

3. Abstract Interpretation Based Program Slicing

3.1 Abstract program slices

A traditional static slicing criterion is a set of variables at a certain program point. As a result, traditional static slices are computed in such a way that they preserve the behavior of the original program at a program point for a subset of the program's variables with respect to *any* possible program executions. Because, however, our slicing model restricts the concept of preserving behavior to a specified set of program executions, we need to take into account the specification of initial states in the definition of slicing criteria.

Definition 1 *Let ϕ be a predicate on the input variables V_{in} which describes assumption about the input values. A (concrete) slicing criterion of a program P is a triple $C = (\phi, p, V)$, where p is a program point in P and V is a subset of the variables in P of interest.*

Definition 2 *Let D and $D^{\mathcal{A}}$ be connected by a Galois connection (α, γ). Then, an abstract slicing criterion of a program P is defined by a triple $C^{\mathcal{A}} = (\phi^{\mathcal{A}}, p, V)$, where $\phi^{\mathcal{A}} = \alpha(\phi)$ for a (concrete) slicing criterion $C = (\phi, p, V)$.*

That is, $\alpha(\phi)$ is the representation of the least(best) upper approximation of the initial condition $\phi \in D$. For example, let ϕ be the input condition on variables x and y defined by $\phi = (x > y + 3) \wedge (y \geq 0)$

If we assume that the lattice of sign shown in Fig. 1 is taken as an upper approximation of the predicates $\mathcal{P} \in \mathcal{Z} \rightarrow \mathcal{B}$ where $\mathcal{Z}$ is the set of integers. Then, we can compute $\alpha(\phi)$ by α_1 and α_2 defined by

$$\begin{array}{ll} \alpha_2 = \lambda \mathcal{P} \in \mathcal{Z} \rightarrow \mathcal{B}. & \text{if } \mathcal{P} = \lambda \mathcal{X}.\text{false then } \bot \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} = 0 \text{ then } 0 \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} < 0 \text{ then } - \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} > 0 \text{ then } + \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} \leq 0 \text{ then } \dot{-} \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} \geq 0 \text{ then } \dot{+} \\ & \text{elseif } \mathcal{P} \Rightarrow \lambda \mathcal{X}.\mathcal{X} \neq 0 \text{ then } \neq 0 \\ & \text{else } \top \text{ fi} \\ \alpha_1 = \lambda \mathcal{P} \in \mathcal{Z}^2 \rightarrow \mathcal{B}. & < \alpha_2(\exists y : \mathcal{P}(x, y)), \alpha_2(\exists x : \mathcal{P}(x, y)) > \end{array}$$

Thus, we have

$$\begin{array}{lll} \alpha_1(\lambda & < x, y > . & ((x > y + 3) \wedge (y \geq 0)) \\ & = & < \alpha_2(\lambda x.(x \geq 3)), \alpha_2(\lambda y.(y \geq 0)) > \\ & = & < +, \dot{+} > \end{array}$$

Consequently,

$$\phi^{\mathcal{A}} \quad = \quad \alpha(\phi) = (x = +) \wedge (y = \dot{+})$$

It is essential to observe that the connection between abstract slicing criterion $C^{\mathcal{A}}$ and concrete slicing criterion C is established by a Galois connection (α, γ). This means that $C^{\mathcal{A}}$ is a *safe* approximation of the corresponding concrete slicing criterion C. Consequently, the (abstract) slice of a program P with respect to $C^{\mathcal{A}}$ is required to include the results of all possible real executions of the concrete slice of P with respect to C for the set of variables V.

Let us assume that p_0, p_1, $\cdots$, p_k are the program points in a program P For $\sigma_i \in State$, let σ_i be a state which is assumed immediately before the execution of p_i. A state σ_i can be restricted to a specified set of variables V, which is denoted by $\sigma_i|V$. We also introduce the notion of *state trajectory* and the projection functions which are useful to demonstrate the properties of slices [1].

Definition 3 *A state trajectory of a program for initial state σ is a finite sequence of ordered pairs $\tau = \langle (p_0, \sigma_0), (p_1, \sigma_1), \cdots, (p_k, \sigma_k) \rangle$, where $\langle p_0, p_1, \cdots, p_k \rangle$ is the program path to be traversed during execution.*

Definition 4 *Let $\tau = \langle \tau_0, \tau_1, \cdots, \tau_k \rangle$ where $\tau_i (0 \leq i \leq k)$ is the i-th pair of the sequence τ, i.e., (p_i, σ_i). Then,*

$$\mathrm{Proj}'_{(p,V)}(\tau_i) = \begin{cases} \lambda & \text{if } p_i \neq p \\ \langle (p_i, \sigma_i|V) \rangle & \text{if } p_i = p \end{cases}$$

Definition 5 Proj *is defined as an extension of* Proj' *to entire trajectories:*

$$\mathrm{Proj}_{(p,V)}(\tau) = \mathrm{Proj}'_{(p,V)}(\tau_0) \cdots \mathrm{Proj}'_{(p,V)}(\tau_k)$$

Let us now consider the relation between state trajectories and condition ϕ of a slicing criterion C. Assume that the abstract condition $\alpha(\phi)$ is characterized by an abstract state $\sigma^{\mathcal{A}}$. It is important to observe that each abstract state $\sigma^{\mathcal{A}}$ corresponds to a set of concrete states $\sigma \in State$ through the function $\gamma_\sigma : AbsState \rightarrow \wp(State)$. Thus, each state σ such that $\sigma \in \gamma_\sigma(\sigma^{\mathcal{A}})$ identifies a state trajectory τ. An abstract slice is any subset of the program statements that preserve the original behavior on each of these trajectories.

Definition 6 *Let a program P' be a portion of a program P, denoted by $P' \preceq P$, if it can be obtained by deleting zero or more statements from P. An abstract slice of a program P on a slicing criterion $C = (\phi, p, V)$ is an executable program P' such that $P' \preceq P$ and whenever P halts on initial state σ where $\sigma \in \gamma \circ \alpha(\phi)$ with state trajectory τ, P' also halts on σ with state trajectory τ' and* $\mathrm{Proj}_{(p,V)}(\tau) = \mathrm{Proj}_{(p,V)}(\tau')$.

One key factor for program slicing is *safety*. That is, we have to guarantee that the resulting abstract slices preserve the behavior of the original program for the slicing criterion C. The "safety" property can be restated in terms of the relation of the collecting semantics between the program P and its slice P' on the slicing criterion C as shown by Theorem 1. The proof of Theorem 1 is straightforward from Definition 6 and the characteristics of abstract interpretation.

Theorem 1 *Let P' be an abstract slice of a program P on a slicing criterion $C = (\phi, p_i, V)$. Then, for input condition ϕ, the following holds:*

$$\textit{if } (p_i, \sigma_i) \in lfp(F_P), \textit{ then } \exists (p_i, \sigma') \in lfp(F_{P'})$$

such that $\sigma_i|V = \sigma'|V$.

3.2 Computation of abstract slices

In order to extract abstract slices from a program P, we also need to identify control and data dependence relations among program points of P [9]. To the ends, we take advantage of dynamic properties of a program obtained during abstract interpretation so that more precise dependences can be captured than those used for computing static slices. The dynamic properties produced by abstract interpretation enable us to identify which program blocks are infeasible for a set of initial states characterized by a given input condition. The statements which are never executed for the set of initial states can be safely ignored when dependences are to be discovered. Identification of infeasible statements can be done by computing abstract collecting

```
int n,i,a[100],sum,asum,psum,nsum;
int pos,neg,zero;

1  read(n);
2  read(a);
3  sum = asum = psum = nsum = 0;
4  pos = neg = zero = 0;
5  i = 1;
6  while (i <= n) do
7    if (a[i] > 0) then
8      psum = psum + a[i];
9      pos = pos + 1;
10   else
11   if (a[i] < 0) then
12     nsum = nsum + a[i];
13     neg = neg+1;
14   else
15     if (a[i] == 0) then
16       zero = zero + 1;
17       if (psum > -nsum) then
18         sum = psum+nsum;
19       else
20         sum = 0;
21       fi
22     fi
23   fi
24 fi
25 i = i +1;
26 end
27 if (neg ==0) then
28   sum = asum = psum;
29 else
30   sum = psum + nsum;
31   asum = psum - nsum;
32 fi
33 print(sum);
34 print(asum);
```

Figure 2. An example program for slicing

semantics of the program P. For example, consider an example program in Figure 2. Let $(L^{int}, \sqsubseteq_{int})$ be an abstract domain for intervals defined as

$$L^{int} = \bot \cup \{[l,u] | l \in Z \cup \{-\infty\} \wedge u \in Z \cup \{+\infty\} \wedge l \leq u\}$$

We employ the abstract domain L^{int} for the program variables "i"and "n", and L^{sign} for the other program variables in Figure 2. We can intuitively compose those abstract domains for the program variables by componentwise abstraction as in [10]. We associate each program point with an abstract state by the system of equations, which is the least fixed point equation $\bar{X} = F_P^{\mathcal{A}}(\bar{X})$. For the program variables ⟨n, i, a, sum, asum, psum, nsum, pos, neg, zero⟩, then, we have

$$\begin{array}{lcl}
X_1 & = & \langle \top, \top, \cdots, \top \rangle \\
X_2 & = & [read(n)]^{\mathcal{A}}(X_1) \\
X_3 & = & [read(a)]^{\mathcal{A}}(X_2) \\
X_4 & = & \langle X_3[n], X_3[i], X_3[a], 0, 0, 0, 0, 0, 0, 0 \rangle \\
X_5 & = & [i = 1]^{\mathcal{A}}(X_4) \sqcup X_{25} \\
X_6 & = & \langle X_5\,[n], [i <= n]^{\mathcal{A}}(X_5)[i] \sqcap \\
 & & [-\infty, max(X_5[n])], X_5[a], X_5[\text{sum}], \cdots, X_5[\text{zero}] \rangle \\
 & & \cdots
\end{array}$$

where, for example, $[i <= n]^{\mathcal{A}}$ denotes the abstract test primitive, which must satisfy,

$$[i <= n]^{\mathcal{A}}(X) \sqsupseteq \alpha_\sigma(\{\sigma \in \gamma_\sigma(X) | \sigma(i) \leq \sigma(n)\})$$

$$\begin{array}{lcl}
X_1 & = & \langle \top, \top, \cdots, \top \rangle \\
X_2 & = & \langle [100,100], \top, \cdots, \top \rangle \\
X_3 & = & \langle [100,100], \top, a, \top \cdots, \top \rangle \text{ where } a = \langle +, \cdots, + \rangle \\
X_4 & = & \langle [100,100], \top, a, 0, 0, 0, 0, 0, 0, 0 \rangle \\
X_5 & = & \langle [100,100], [1, +\infty], a, 0, 0, 0, 0, 0, 0, 0 \rangle \\
X_6 & = & \langle [100,100], [1,100], a, 0, 0, +, 0, +, 0, 0 \rangle \\
 & & \cdots \\
X_{26} & = & \langle [100,100], [101, +\infty], a, 0, 0, +, 0, +, 0, 0 \rangle \\
 & & \cdots
\end{array}$$

Figure 3. Solution for the system of equations

Abstract interpretation is finitely computable if the semantic function is monotone and the abstract domain is finite or ascending chain finite. Otherwise, it might have an infinite computation. For this case, speed-up techniques like widening [8] can be employed to ensure finiteness when necessary. In this way, we can compute an approximate solution to the original system of equations in finite time as well known in [8, 10]. If the input condition ϕ is given by "$\forall 1 \leq i \leq n : a[i] > 0$", then its abstraction $\phi^{\mathcal{A}}$ is "$\forall 1 \leq i \leq n : a[i] = +$". Under this input condition, the solution to the modified fixed point equations is summarized in Figure 3 Once we have computed an abstract collecting semantics of a program for an input condition, we can identify infeasible statements of the program which will not be taken into account for program slicing. In order to define the transformation rules which simplify the original program, we introduce a few definitions. The abstract boolean restriction function $\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}}$ can be defined as:

$$\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}} = \alpha_\sigma\{\sigma \in \gamma_\sigma(\sigma^{\mathcal{A}}) \mid b \; is \; true \; in \; \sigma\}$$

where $\alpha_\sigma(\Sigma) = \bot_\sigma$ if $\Sigma = \emptyset$. Note that if there is no possible value in the abstract state $\sigma^{\mathcal{A}}$, the resulting state will be $\bot_\sigma$.

Let $\sigma^{\mathcal{A}}$ be an abstract state which is assumed before the execution of the program component S. Then, the transformation rules which enables us to find a simpler program P^T from the program P are as follows:

Rule1 If S is a conditional statement of the form **if** b **then** S_1 **else** S_2 **fi** and $\mathcal{R}^{\mathcal{A}}[\neg b]\sigma^{\mathcal{A}} = \bot_\sigma$ then $P[\textbf{skip}/S_2]$, where $P[S_i/S]$ represents the replacement of S by S_i and **skip** represents a null statement.

Rule2 If S is a conditional statement of the form **if** b **then** S_1 **else** S_2 **fi** and $\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}} = \bot_\sigma$ then $P[\textbf{skip}/S_1]$.

Rule3 If S is a conditional statement of the form **if** b **then** S_1 **else** S_2 **fi** and $\mathcal{R}^{\mathcal{A}}[\neg b]\sigma^{\mathcal{A}} \neq \bot_\sigma$ and $\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}} \neq \bot_\sigma$ then S is not replaced with either S_1 or S_2. That is, $P[S/S]$.

Rule4 If S is a while-loop statement of the form **while** b **do** S_1 **end** and $\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}} = \bot_\sigma$ then $P[\textbf{skip}/S_1]$.

Rule5 If S is a while-loop statement of the form **while** b **do** S_1 **end** and $\mathcal{R}^{\mathcal{A}}[b]\sigma^{\mathcal{A}} \neq \perp_\sigma$, then S is unchanged. That is, $P[S/S]$.

Let us apply the rules to the program in Figure 2 on the input condition $\forall i : \ 1 \leq \text{i} \leq n, \text{a[i]} > 0$ and $\text{n} = 100$. Recall that Figure 3 shows the abstract states $\sigma_i^{\mathcal{A}}$ before the execution of each program statement i of the program in Figure 2 on the input condition. Consider X_5 which shows the abstract state $\sigma_5^{\mathcal{A}}$ before the execution of the predicate in line 5. Because $\sigma_5^{\mathcal{A}}(i) = [1, +\infty]$, but $\sigma_5^{\mathcal{A}}(n) = [100, 100]$, two outcomes of the predicate in line 5 are possible. This means that we could not discard the loop body according to Rule 5. The abstract state $\sigma_6^{\mathcal{A}}$, however, is given by:

$$\langle [100, 100], [1, 100], a, 0, 0, +, 0, +, 0, 0 \rangle$$

This implies that the predicate in line 6 will always evaluate to **True**. Therefore, the $false$ branch can be discarded according to Rule 1. Similarly, we can observe that the $false$ branch of the predicate in line 26 will never be executed because the program variable "neg" is equals to 0 in the corresponding abstract state $\sigma_{26}^{\mathcal{A}}$. Therefore, we can also apply Rule 1 to this case which allows us to delete the lines 29 and 30.

3.3 Abstract program dependence graph

Abstract program dependence graph for P, denoted by $G_P^{\mathcal{A}}$, is a program dependence graph which is defined in terms of a program P^T obtained from P by the applications of the transformation rules. The nodes of $G_P^{\mathcal{A}}$ is the program components such as the assignment statements and control predicates that occur in P^T, rather than in P. As in the traditional program dependence graph [9], the edges of $G_P^{\mathcal{A}}$ represent control dependences and data dependences. We denote a control dependence edge from node u to v by $u \rightarrow_{c^{\mathcal{A}}} v$ and a data dependence edge by $u \rightarrow_{d^{\mathcal{A}}} v$ in $G_P^{\mathcal{A}}$.

The definitions of control dependence and data dependence in $G_P^{\mathcal{A}}$ make no difference from the one used in the traditional program dependence graph [9]. However, the existence of a data dependence edge between two nodes of $G_P^{\mathcal{A}}$ such as $u \rightarrow_{d^{\mathcal{A}}} v$ implies the existence of the data dependence from u to v. The opposite, however, is not necessarily true. In order to clarify this, let us consider the following data flow concepts.

The location or the l-value of a variable v is an instance of v when it is evaluated as a target of an assignment statement. And the current content of the location is referred to as the r-value of v. Let $D(p_i)$ be the set of variables whose l-values are used at node p_i and $U(p_i)$ be the set of variables whose r-values are used at node p_i. These sets differ from their static counterparts in that they are of dynamic nature to a certain extent. As an example, consider a program fragment:

```
        ...
p_l1 : a[i] = a[j]*k;
p_l2 : i= i+2;
p_l3 : z = a[i];
        ...
```

Suppose that $\sigma_{l_1}^{\mathcal{A}}(i)$ is $[2..3]$ and $\sigma_{l_1}^{\mathcal{A}}(j)$ is $[3..5]$. Then, the following variables are used and defined:

$$D(p_{l_1}) = \{a[2], a[3]\} \text{ and } U(p_{l_1}) = \{a[3], a[4], a[5], i, j, k\}$$

Similarly, we can compute $D(p_{l_3})$ and $U(p_{l_3})$ by considering the effect of the assignment statement at p_{l_2}:

$$D(p_{l_3}) = \{z\} \text{ and } U(p_{l_3}) = \{a[4], a[5], i\}$$

In contrast, traditional static slicing treats an entire array as a single variable. This is due to the fact that static slicing fails to take into account any information about particular array elements. As a result, the slice can be unnecessarily large. It is important to observe that the dependences used in static slicing are defined for all possible initial states whereas abstract slicing defines dependences in terms of the specified set of initial states. Furthermore, abstract interpretation enables us to determine which array elements might be used or modified at every point of program execution. For example, the traditional static analysis of the program fragment would lead to

$$D(p_{l_1}) = \{a\},\ U(p_{l_1}) = \{a, i, j, k\},$$

$$D(p_{l_3}) = \{z\},\ U(p_{l_3}) = \{a, i\}$$

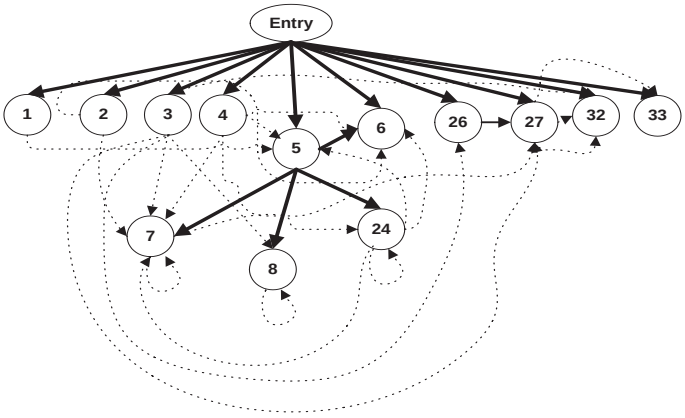

Figure 4. An example abstract program dependence graph

From the static point of view, the data dependence from p_{l_1} to p_{l_3} are represented on its program dependence graph whereas the data dependence from p_{l_1} to p_{l_3} does not exist in the corresponding abstract program dependence graph. If, however, the program fragment lies in the loop and the widening operation is performed at the index variable "i", then the precision might be reduced.

From this discussion, an abstract slice of a program P with respect to a slicing criterion $C = (\phi, p, V)$ can be defined as a collection of statements corresponding to the

```
int n,i,a[100],sum,asum,psum,nsum;
int pos,neg,zero;

1       read(n);
2       read(a);
3       sum = psum = neq = 0;
4       i = 1;
5       while (i <= n) do
6            if (a[i] >0) then
7               psum = psum + a[i];
23          fi
24              i = i +1;
25      end
26      if (neq = 0) then
27              sum  = psum;
31      fi
32      print(sum);
```

Figure 5. An example abstract slice

nodes v of $G_P^{\mathcal{A}}$ on which p has a transitive control or data dependence, i.e., $v \rightarrow^*_{c^{\mathcal{A}},d^{\mathcal{A}}} p$. Figure 4 shows the abstract program dependence graph resulting from the application of the transformation rules to the program in Figure 2 and Figure 5 shows its abstract slice on the slicing criterion $C = (\phi, 32, \{\text{sum}\})$, where $\phi = \forall i : 1 \leq \text{i} \leq n, \text{a[i]} > 0$ and n is 100. In the present case, the resulting abstract slice is identical to the dynamic slice on the slicing criterion $(\forall i : 1 \leq \text{i} \leq n, \text{a[i]} = 1, 32, \{\text{sum}\})$. We can further reduce the size of the slice by considering that the predicates in line 6 and line 26 always evaluate to **True**. That is, the predicates and the statements which might affect their outcomes could be deleted.

4. Future Work

While the notion of abstract slices is new and our research offers improvements weak sides of traditional slicing techniques, there are some issues that are worthy of further research. In theoretical aspects, we need to further identify other variants of abstract slices and their characteristics and the relations between them. For example, abstract slicing might yield more focused and precise slices by fully utilizing the information present in the specification. In the present case, we only consider the precondition which specifies a subset of initial states. Usually, a secification is denoted by a pre-post condition pair where the postcondition represents the desired behaviour of the component one wants to extract[11]. We are necessitated to map the postcondition to the specification of the slicing criterion. This can be done by the backward (approximate) semantic analysis in determining at each program point an invariant property which are the ascendants of the output states satisfying a given postcondition[12]. Yet another important research area is the application of abstract slicing to the fields such as restructuring, program comprehension, and software reuse. We need to extend abstract slicing to the interprocedural level for a wider application.

References

[1] M. Weiser, Program slicing, *IEEE Trans. on Software Engineering*, vol. 10, no. 4, 1984, 352-357.

[2] H. Agrawal and J. Horgan, Dynamic program slicing, In *Proc. of the ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, 1990, 246-256.

[3] B. Korel and J. Laski, Dynamic program slicing, *Information Processing Letters*, vol. 29, no. 3, 1988, 155-163.

[4] G. Canfora, A. Cimitile, and A. De Lucia, Conditioned Program Slicing, *Information and Software Technology*, vol. 40, 1998, 595-607.

[5] S. Danicic, C. Fox, M. Harman, and R. Hierons, ConSIT: A Conditioned Program Slicer, *Proc. of Conf. on Software Maintenance*, S. Jose, CA, U. S. A., IEEE CS Press, 2000, 216-226.

[6] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, "Program Slicing Based on Specification", *2001 ACM Syposium on Applied Computing*, 2001, 605-609.

[7] W. K. Lee, I. S. Chung, G. S. Yoon, and Y. R. Kwon, Specification-based Program Slicing and Its Applications, *Journal of Systems Architecture*, 47(2001), 427-443.

[8] P. Cousot and R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. of 4th ACM Symp. on Principle of Programming Languages*, ACM Press, 1977, 238-252.

[9] S. Horwitz, T.Reps, and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Trans. on Programming Languages and Systems*, vol. 12, no. 1, Jan. 1990, 35-46.

[10] P. Cousot and R. Cousot, Systematic design of program analysis frameworks, *Proc. of 4th ACM Symp. on Principle of Programming Languages*,ACM Press, 1979, 269-282.

[11] A. Cimitile, A. De Lucia, and M. Munro, A Specification Driven Slicing Process for Identifying Reusable Functions, *Journal of Software Maintenance: Research and Practice*, vol. 8, no. 3, 1996, 145-178.

[12] F. Bourdoncle, Abstract debugging of higher-order imperative languages, *Proceedings of ACM Symp. on PLDI*, 1993, 46-55.