Value Slice: A New Slicing Concept for Scalable Property Checking

Shrawan Kumar[1,⋆], Amitabha Sanyal[2], and Uday P. Khedker[2]

[1] Tata Consultancy Services Ltd, Pune, India
shrawan.kumar@tcs.com
[2] IIT Bombay, Mumbai 400076, India
{as,uday}@cse.iitb.ac.in

Abstract. A backward slice is a commonly used preprocessing step for scaling property checking. For large programs though, the reduced size of the slice may still be too large for verifiers to handle. We propose an aggressive slicing method that, apart from slicing out the same statements as backward slice, also eliminates computations that only decide whether the point of property assertion is reachable. However, for precision, we also carefully identify and retain *all* computations that influence the values of the variables in the property. The resulting slice, called *value slice*, is smaller and scales better for property checking than backward slice.

We carry experiments on property checking of industry strength programs using three comparable slicing techniques: backward slice, value slice and an even more aggressive slicing technique called thin slice that retains only those statements on which the variables in the property are data dependent. While backward slicing enables highest precision and thin slice scales best, value slice based property checking comes close to the best in both scalability and precision. This makes value slice a good compromise between backward and thin slice for property checking.

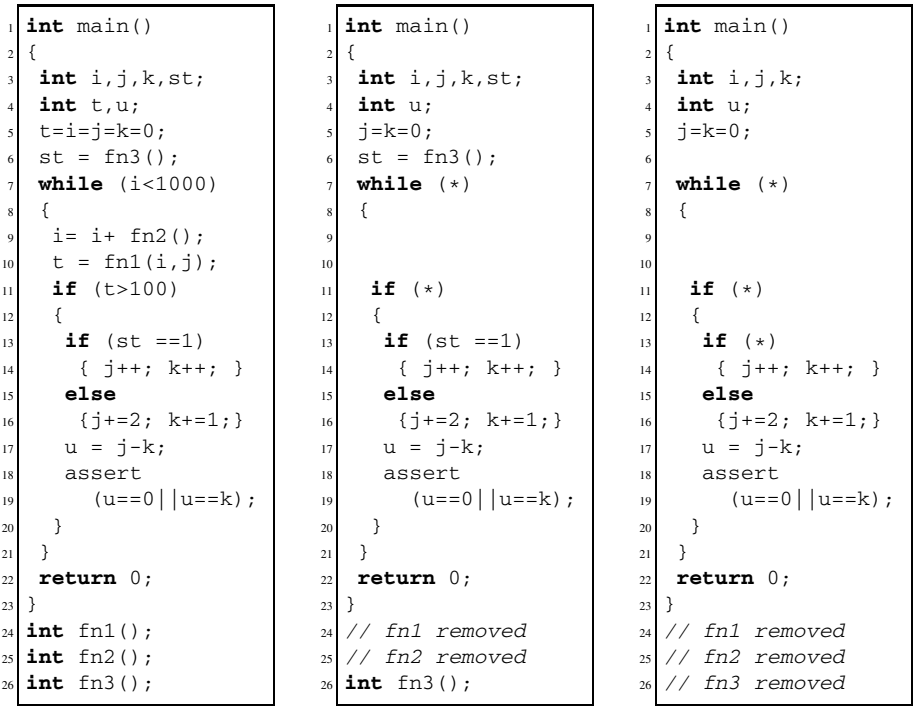
1 Introduction

Given a program and a set of variables at a program point of interest, *program slicing* [19] pares the program to contain only those statements that are likely to influence the values of the variables at that program point. The set of variables and the program point, taken together, is called the *slicing criterion*. Several variants of the original slicing technique, called *backward slicing*, have since been proposed [16]. These have been used for program understanding, debugging, testing, maintenance, software quality assurance and reverse engineering. A survey of applications of program slicing appears in [5]. This paper focuses on the use of slicing for scaling up property checking.

Among slicing techniques, backward slicing is the natural choice for property checking. While computation of backward slice is efficient and scalable, the size of the slice is a matter of concern. Empirical studies [11] have shown that the size of the backward slice on an average is about 30% of the program size. This size is still too large for the analysis of large programs. In addition, the statements sliced out are irrelevant

⋆ Also research scholar at IIT Bombay. This work is part of his doctoral dissertation.

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 101–115, 2015.
DOI: 10.1007/978-3-662-46681-0_7



```
1  int main()
2  {
3   int i,j,k,st;
4   int t,u;
5   t=i=j=k=0;
6   st = fn3();
7   while (i<1000)
8   {
9    i= i+ fn2();
10   t = fn1(i,j);
11   if (t>100)
12   {
13    if (st ==1)
14     { j++; k++; }
15    else
16     {j+=2; k+=1;}
17    u = j-k;
18    assert
19      (u==0||u==k);
20   }
21  }
22  return 0;
23 }
24 int fn1();
25 int fn2();
26 int fn3();
```

(a) Backward slice

```
1  int main()
2  {
3   int i,j,k,st;
4   int u;
5   j=k=0;
6   st = fn3();
7   while (*)
8   {
9
10
11   if (*)
12   {
13    if (st ==1)
14     { j++; k++; }
15    else
16     {j+=2; k+=1;}
17    u = j-k;
18    assert
19      (u==0||u==k);
20   }
21  }
22  return 0;
23 }
24 // fn1 removed
25 // fn2 removed
26 int fn3();
```

(b) Value slice

```
1  int main()
2  {
3   int i,j,k;
4   int u;
5   j=k=0;
6
7   while (*)
8   {
9
10
11   if (*)
12   {
13    if (*)
14     { j++; k++; }
15    else
16     {j+=2; k+=1;}
17    u = j-k;
18    assert
19      (u==0||u==k);
20   }
21  }
22  return 0;
23 }
24 // fn1 removed
25 // fn2 removed
26 // fn3 removed
```

(c) Thin slice

Fig. 1. Usual backward slice, value slice and thin slice

to the asserted property and their elimination does not reduce the load on the verifier significantly. To remedy this, we propose an alternate notion of slicing based on the observation that a backward slice consists of two categories of statements (i) statements that decide whether the slicing criterion will be reached during execution, and (ii) statements that decide the values of variables in the slicing criterion. Our results show that the second category of statements, called *value-impacting*, are often enough for property checking. We also show that the size of the slice consisting of value-impacting statements, called a *value slice*, is about half the size of the backward slice.

An attempt similar to ours called *thin slicing* [17] retains only those statements on which the variables in the slicing criterion are data-dependent. In particular, *all* conditional statements are eliminated. While this does bring down the size of the slice, unlike our method it also eliminates some conditional statements that are value-impacting and thus crucial for property checking.

As a motivating example, Figure 1(a) shows an `assert` involving `u` at line 18. The functions `fn1` and `fn2` are large and complex but without side effects. Clearly, a backward slice with the slicing criterion $\langle 18, \mathtt{u} \rangle$ does not eliminate any statement from the program. SATABS (version 3.0) [8], a robust and scalable predicate abstraction based property checking tool, times out on this program on a limit of 20 minutes.

Observe however that the value of u does not depend on the values of i or t. Since these variables merely decide the reachability of line 18 during execution, the statements computing them are non-value-impacting and thus considered irrelevant. Issues related to reachability are being addressed in an ongoing work and are beyond the scope of this paper.

Figure 1(b) shows a slice of the program that captures the computation of every value of u in the original program. Conditional statements that do not value-impact u are replaced by a $*$ standing for a randomly chosen boolean value. The resulting slice is much smaller in comparison to the backward slice (the entire program). SATABS succeeds in showing that the property is indeed satisfied on the sliced program, and, by implication, on the original program. On the other hand, the thin slice shown in Figure 1(c), while smaller in size, is not useful since the property does not hold on the sliced program. Thus any verifier will produce counterexamples on this slice that will be spurious on the original program.

The contributions of this paper are:

1. We define a new notion of slicing called value slice and propose a worklist based algorithm for its computation. The algorithm is shown to be correct by construction.
2. We describe the results of experiments on property checking based on the three comparable slicing methods—backward, value and thin slices. We show that on both criteria, scalability and precision, value slice based property checking yields results that are close to the best among the three slicing methods.

We conclude that as a slicing technique for increasing the scalability of property checking, value slice represents a sweet spot between backward and thin slice.

2 Background

We shall present our ideas in the context of imperative programs made of assignments, conditional statements (the conditions being without side-effects), *while* loops, and function calls. We allow *break* and *continue* statements in loops. However, we restrict ourselves to goto-less programs with single-entry loops and two-way branching conditional statements; it makes for an easier formal treatment of our method without losing expressibility. We also allow the full range of C-types including arrays and pointers.

Our analysis will be based on a model of the program called the control flow graph (CFG) [1]. A CFG is a pair $\langle N, E\rangle$, where N is a set of nodes representing atomic statements, i.e. assignment statements and conditions (also called predicates) of the program[1]. Further, $(n_1, n_2) \in E$, if there is a possible flow of control from n_1 to n_2 without any intervening node. We use $n_1 \rightarrow n_2$ and $n_1 \xrightarrow{b} n_2$ to denote unconditional and conditional edges, where $b \in \{true, false\}$ indicates the branch outcome. Each statement (or node) is associated with a unique label l that represents the program point just before the statement. Often we shall refer to a node by its label. In addition, each CFG is assumed to have two distinguished nodes with labels $ENTRY$ and $EXIT$. Except for $ENTRY$ and $EXIT$, there is a one-to-one correspondence between the nodes

[1] For the rest of the paper, a statement will mean an atomic statement.

of the CFG and the statements of the program. Thus we shall use the terms statement and node interchangeably.

2.1 Program States and Traces

Let Var be the set of variables in a program P and Val be the set of possible values which the variables in Var can take. A *program state* is a map $\sigma : Var \rightarrow Val$ such that $\sigma(v)$ denotes the value of v in the program state σ. Given $X \subseteq Var$, a *X-restriction* of σ, denoted as $\lfloor\sigma\rfloor_X$, is a map $X \rightarrow Val$ such that $\forall x \in X.\lfloor\sigma\rfloor_X(x) = \sigma(x)$. Finally, an *execution state* is a pair $\langle l, \sigma\rangle$, where σ is a program state and l is the label of a CFG node. The execution of a program is a sequence of execution states starting with $\langle \mathit{ENTRY}, \sigma_0\rangle$, where σ_0 is the initial program state. We assume that the next state is given by a function $\mathcal{T}$, i.e. for each execution state $\langle l, \sigma\rangle$, the next state is $\mathcal{T}(\langle l, \sigma\rangle)$.

A *trace* is a (possibly infinite) sequence of execution states $[\langle l_i, \sigma_i\rangle],\ i \geq 0$, where $l_0 = \mathit{ENTRY}$, σ_0 is an initial program state, and $\langle l_{i+1}, \sigma_{i+1}\rangle = \mathcal{T}(\langle l_i, \sigma_i\rangle)$ for all $i \geq 0$. When the trace sequence is finite and ends with an execution state $\langle \mathit{EXIT}, \sigma\rangle$, it is called a *terminating trace*. We shall only consider terminating traces in the rest of the paper.

2.2 Data and Control Dependence

A definition d of a variable v in a node n is said to be a *reaching definition* [1] for a label l, if there is a control flow path from n to l devoid of any other definition of v. A variable x at label l is said to be *data dependent* on a definition d of x, if d is a *reaching definition* for l. Given a set of variables X and a label l, the set of definitions that the variables in X are dependent on is denoted by $DU(l, X)$.

Backward slicing algorithms are implemented efficiently using post-dominance and control dependence [10,12]. A node n_2 *post-dominates* a node n_1 if every path from n_1 to *EXIT* contains n_2. If, in addition, $n_1 \neq n_2$, then n_2 is said to *strictly post-dominate* n_1. A node n is *control dependent* on an edge $c \xrightarrow{b} n'$, denoted $c \overset{b}{\rightarrowtail} n$, if n *post-dominates* n', and n does not strictly *post-dominate* c. If the label b is not important in a context, it is elided.

The transitive closure of control dependences, i.e. a chain of control dependences starting with the predicate c and edge b and ending with the node n is denoted as $c \overset{b}{\rightsquigarrow} n$. Note that because of `return` and `break` statements, it is possible to have both $c \overset{b}{\rightsquigarrow} n$ and $c \overset{b'}{\rightsquigarrow} n$, where $b \neq b'$.

2.3 Subprogram and Backward Slice

The basis for slicing is a *slicing criterion* defined as a pair $\Upsilon = \langle l, V\rangle$, where l is a statement label and $V \subseteq Var$ is a set of variables. The slicing criterion represents our interest in the values of the variables in V just before the execution of the statement at l. Let $REF(s)$ denote the set of variables referred in a node s. Given a statement s with label l', we will use $LV(s)$ to denote the slicing criterion $\langle l', REF(s)\rangle$.

A *subprogram* of P is a program formed by deleting some statements from P while retaining its structure. This means if a statement enclosed by a predicate c in P is

included in the subprogram, then so is c itself. Given a program P and a slicing criterion with location l, an *augmented program* is obtained by inserting a $SKIP$ (do nothing) statement at l. Clearly, an augmented program has the same behavior as the original program. In the sequel, we shall assume that our programs are augmented. Finally, we shall assume that program points of the same statement in the original program and the slice are represented by the same label.

Assume for the rest of this section that the slicing criterion is $\Upsilon = \langle l, V \rangle$. Given a program P, we define *SC-execution states* to be the execution states of P with label l. For a subprogram to be called a backward slice, there should be a relation between the traces of the program and the slice on the same input when we restrict the traces to their SC-execution states. We call a trace thus restricted as a *sub-trace*. We say that the two sub-traces $[(l, \sigma_i)], 1 \leq i \leq k$ and $[(l, \sigma'_i)], 1 \leq i \leq k'$ are *SC-equivalent* wrt Υ, if $k = k'$, and for each i between 1 and k, $\lfloor \sigma_i \rfloor_V = \lfloor \sigma'_i \rfloor_V$.

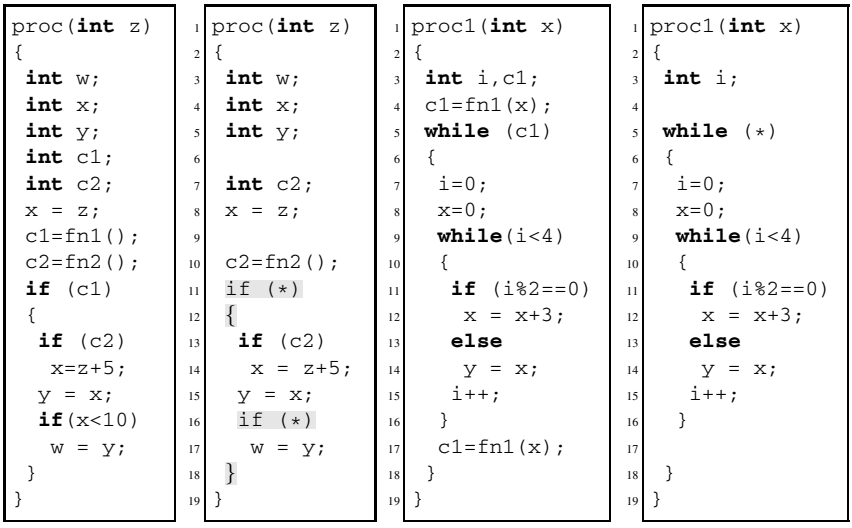
Let $Tr(P, I, \Upsilon)$ denote the sub-trace of a program P on input I for the slicing criterion Υ. We now define P' to be a *backward slice* of P with respect to Υ, if for all inputs I, $Tr(P, I, \Upsilon)$ and $Tr(P', I, \Upsilon)$ are SC-equivalent. As we shall see later, for the same input the sub-traces of the program and its value-slice may not be of the same length. We therefore need a weaker notion of SC-equivalence. We say that a pair of sub-traces $[(l, \sigma_i)], 1 \leq i \leq k$ and $[(l, \sigma'_i)], 1 \leq i \leq k'$ are *weak-SC-equivalent* wrt Υ, if for each i between 1 and $min(k, k')$, $\lfloor \sigma_i \rfloor_V = \lfloor \sigma'_i \rfloor_V$. The value $min(k, k')$ is called the *trace observation window* for the pair of sub-traces.

3 Value Slice

Given a slicing criterion $\langle l, V \rangle$, a value slice is the answer to the question: "Which statements can possibly influence the values of the variables in V observed at l"?

The answer to this question for $P1$ in Figure 2 for the slicing criterion $\langle 17, \{\mathtt{y}\} \rangle$ is as follows: `y` at 17 gets its value from `x` through the assignment at 15. `x`, in turn, gets its value from the definitions at 14 and 8, and both of these can reach 15. Thus 8, 14 and 15 are in the value slice. The predicate `c2` at 13 is also in the value slice, since, of the values generated at 14 and 8, the value that actually reaches line 15 is decided by `c2`. Finally, line 10, where `c2` itself is computed, is also in the value slice. The resulting program is $P2$ without the lines shaded gray.

Although $P2$ (ignoring gray lines) contains all statements required to answer the question posed earlier for the slicing criterion $\langle 17, \{\mathtt{y}\} \rangle$, it is not suitable for property checking. The reason is that apart from the statements that decide the values of variables at the slicing criterion, we also need to explicate the CFG paths along which the computations of these values take place. Therefore, if a statement in the slice is control dependent on a predicate that, by itself, does not influence values of the variables in the slicing criterion, the predicate is retained in the slice in an abstract form. This brings the predicates at lines 16 and 11 into the value slice but replaced by '*' indicating a non-deterministic branch. We call such predicates *abstract predicates*. Note, however, that if none of the statements that are transitively control dependent on a predicate are included in the slice, the predicate itself can be eliminated.



(a) P1 **(b)** P2 **(c)** P3 **(d)** P4

Fig. 2. Various forms of value slices

In the context of property checking, the inclusion of `c2` in a concrete form at line 13 is a crucial difference between value slice and thin-slice[2]. As an example, assume that when $P1$ is executed with v as the initial value of `z`, `c2` evaluates to $false$ and the value reaching `y` at 17 is also v. For the same initial value of `z`, the value slice $P2$ will also assign the same value v to `y`. However, if we abstract `c2` as '*', the resulting program may produce a trace which assigns the value $v+5$ to y at line 17. To avoid such spurious counterexamples, we retain the predicate `c2` at line 13 in a concrete form.

To generalize this point, consider the execution of $P3$ in Figure 2. Assuming that the outer loop executes twice for an input, the sub-trace for $\langle 14, \{\texttt{x}\}\rangle$ is $[\langle 14,3\rangle, \langle 14,6\rangle, \langle 14,3\rangle, \langle 14,6\rangle]$. However, if the predicates of both the `while`s are abstracted, then one of the sub-traces generated is $[\langle 14,3\rangle, \langle 14,3\rangle]$. The two sub-trace do not match in that they are not weak-SC-equivalent. On the other hand, program $P4$ in which only the outer loop predicate is abstracted, produces as a sub-trace zero or more repetitions of $[\langle 14,3\rangle, \langle 14,6\rangle]$. We therefore include the predicate `i<4` in the value slice for the slicing criterion $\langle 14, \{\texttt{x}\}\rangle$. The predicate `i%2==0` is also in the value slice by a similar argument. In summary, for the same input, the sub-traces of a value-slice and the original program are required to be weak-SC-equivalent. Based on these considerations, we now specify the conditions to be satisfied by a value slice.

Definition 1. (Value-slice) *A value slice P^V of a program P for a slicing criterion $\langle l, V\rangle$ satisfies the following conditions:*
1. P^V is a subprogram of P with some predicates in abstract form.

[2] For comparison in the context of property checking, predicate `c2`, which would have been eliminated in the thin-slice, is retained in an abstract form.

2. If P terminates with trace τ on an input, then there should exist a trace τ' of P^V on the same input which is SC-equivalent *to τ.*
3. If P terminates with trace τ on an input, then every trace τ' of P^V on the same input should be weak-SC-equivalent *to τ.*

3.1 Value-Impacting Statements

While the trace-based definition is good from a semantic point of view, we present a definition that will enable us to statically identify the set of statements that should necessarily be in the value slice in concrete form. We call such statements *value-impacting* and define the term shortly. As mentioned in the background section, we shall use the term "node" to also mean atomic statements.

Definition 2. (Value-impacting Node) *A node s* value-impacts *$\Upsilon = \langle l, V \rangle$, if any of the following conditions hold:*
1. s is an assignment in $DU(\Upsilon)$.
2. s is an assignment, and there exists a node t such that t value-impacts *Υ and s is in $DU(LV(t))$.*
3. s is a predicate c from which there exist paths π_1 and π_2 starting with the out-edges of c and ending at the first occurrence of l. Further, there exists a node $t \neq c$ such that t value-impacts *Υ, and (a) t is the first value-impacting node along π_1 (b) t is not the first value-impacting node along π_2.*

A triplet $\langle \pi_1, \pi_2, t \rangle$ due to which a predicate c satisfies rule (3) will be called a *witness* for c being value-impacting. As an illustration, consider the slicing criterion $\langle 14, \{\mathtt{x}\} \rangle$ for $P3$ in Figure 2. Statements 12 and 8 are value-impacting because of rules 1 and 2. Interestingly, the predicates `i%2==0` and `i<4` are value-impacting because of rule 3 with witnesses $\langle \pi_1\colon 11 \xrightarrow{t} 12 \rightarrow 15 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14,\ \pi_2\colon 11 \xrightarrow{f} 14, 12 \rangle$ and $\langle \pi_1\colon 9 \xrightarrow{t} 11 \xrightarrow{f} 14,\ \pi_2\colon 9 \xrightarrow{f} 17 \rightarrow 5 \xrightarrow{t} 7 \rightarrow 8 \rightarrow 9 \xrightarrow{t} 11 \xrightarrow{f} 14, 11 \rangle$. Clearly, if a node s *value-impacts* Υ then there is a path from s to l.

Let $VI(\Upsilon)$ be the set of value-impacting nodes of Υ. Let the set of abstract predicates $AP(\Upsilon)$ consist of predicates that are not by themselves *value-impacting*, but on which other value-impacting nodes are transitively control dependent. We construct a subprogram P^{VS} of P by choosing the statements in $VI(\Upsilon) \cup AP(\Upsilon)$ along with *SKIP* and *ENTRY*. The predicates in $AP(\Upsilon)$ appear in P^{VS} in abstract form. We claim that P^{VS} is a *value slice*. Clearly condition 1 of Definition 1 is satisfied. To show that P^{VS} satisfies conditions 2 and 3, we shall first prove a lemma which shows that if the traces of the original program and the value slice on the same input are restricted to executions states involving value-impacting statements, then they match each other when compared to the extent of the trace with the smaller length. In the lemmas below, AVI denotes the set of concrete statements in P^{VS}. Further, for $s \in AVI$, $AREF(s)$ denotes $REF(s)$ when $s \in VI(\Upsilon)$, V when s is *SKIP* and $\emptyset$ when s is *ENTRY*.

Lemma 1. *Let τ and τ' be traces of the programs P and P^{VS} for an input I. Assume that $\tau_s = [\langle l_i, \sigma_i \rangle]$, $i \geq 1$ and and $\tau'_s = [\langle l'_j, \sigma'_j \rangle]$, $j \geq 1$ are restrictions of τ and τ' to the statements in AVI. Let k be the minimum of the number of elements in τ_s and τ'_s. Then for all $i \leq k$, $l_i = l'_i$ and $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, where $Z_i = AREF(l_i)$.*

Proof. We shall prove the lemma by induction on the common label index i of the two traces. Obviously $i \leq k$, else the lemma is vacuously true.
Base step : $i = 1$. The lemma holds trivially as $l_1 = l'_1 = ENTRY$ and $\sigma_1 = \sigma'_1 = I$.
Induction step: Let the hypothesis be true for i. Since $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, the edges followed from l_i and l'_i in τ and τ' are the same. Assume $l_{i+1} \neq l'_{i+1}$. This is only possible if (a) there is a predicate c in the original program which has been abstracted in the value slice, (b) the path from l_i to l_{i+1} goes through one of the out-edges b_1 of c, and (c) the path from l'_i to l'_{i+1} goes through the other out-edge b_2 of c. Obviously, there are paths π_1 and π_2 from c to l through b_1 and c to l through b_2, and l_{i+1} and l'_{i+1} are the first value-impacting statements on π_1 and π_2 respectively. Therefore, the predicate c is value-impacting and cannot be abstracted in the value-slice, a contradiction. Therefore, $l_{i+1} = l'_{i+1}$.

Now suppose that for some variable $x \in Z_{i+1}$, $\sigma_{i+1}(x) \neq \sigma'_{i+1}(x)$. Then there must be a statement d which provides the value of x at l_{i+1}; x does not get its value from the input I. This implies d is a value-impacting statement. Clearly, d occurs before l_{i+1} and thus it either also occurs before l_i or is l_i itself. By induction hypothesis, d must also be there in τ' and therefore $\sigma_{i+1}(x) = \sigma'_{i+1}(x)$. ■

The following lemma implies that condition 2 of Definition 1 holds for P^{VS}.

Lemma 2. *Let τ be a finite trace for program P for an input I. Let $\tau' = [\langle l_i, \sigma_i \rangle]$, $1 \leq i \leq k$, be the sub-sequence of τ restricted to the nodes in P^{VS}. Then for every prefix of τ' there is a prefix $\tau'' = [\langle l'_i, \sigma'_i \rangle]$ of some trace of P^{VS} for the same input I, such that for all i, $1 \leq i \leq k$, (a) $l_i = l'_i$, (b) if l_i is in $AVI(\Upsilon)$, then $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$, where $Z_i = AREF(l_i)$.*

Proof. Consider a sub-sequence τ' of an arbitrary trace. Let the length of the sub-sequence be k. Let τ'_i be the prefix of τ' with length i. The proof is by induction on the length i of the prefix τ'_i.
Base step: $i = 1$ The lemma holds trivially as $[\langle ENTRY, I \rangle]$ is the only prefix of length 1 for any trace of P as well as P^{VS}.
Induction step: Assume that the statement of the lemma holds for prefixes of τ' of length up to i. Consider a prefix τ'_{i+1} of length $i+1 \leq k$. By induction hypothesis, there exists a trace of P^{VS}, which has a prefix τ''_i of length i and for which statement of the lemma holds with respect to the prefix τ'_i. If the node l_i in τ'_{i+1} (and in τ'_i) is an abstract predicate in $AP(\Upsilon)$, then program control reaching the predicate can take either branch. Otherwise $l_i \in AVI(\Upsilon)$, and $\lfloor \sigma_i \rfloor_{Z_i} = \lfloor \sigma'_i \rfloor_{Z_i}$ by the induction hypothesis. Thus for any edge taken out of l_i in τ', l'_i in τ''_i can be made to take the same edge out. Assume this edge extends τ''_i to τ''_{i+1} by taking l'_i to l'_{i+1}.

We claim that there exists a trace of P^{VS}, having τ''_{i+1} as its prefix, such that $l_{i+1} = l'_{i+1}$. If not, the divergence must be because of some condition c after l_i and before l_{i+1} in τ'. But then $c \rightsquigarrow l_{i+1}$ and therefore $c \in P^{VS}$. This means that there is a trace of P^{VS} such that $l_{i+1} = l'_{i+1}$. Further, by Lemma 1, if $l_{i+1} \in AVI(\Upsilon)$, $\lfloor \sigma_{i+1} \rfloor_{Z_{i+1}} = \lfloor \sigma'_{i+1} \rfloor_{Z_{i+1}}$. ■

Now consider sub-traces of P and P^{VS} for an arbitrary input I. Using Lemma 1, it is easy to show by an induction on the length of the trace observation window that

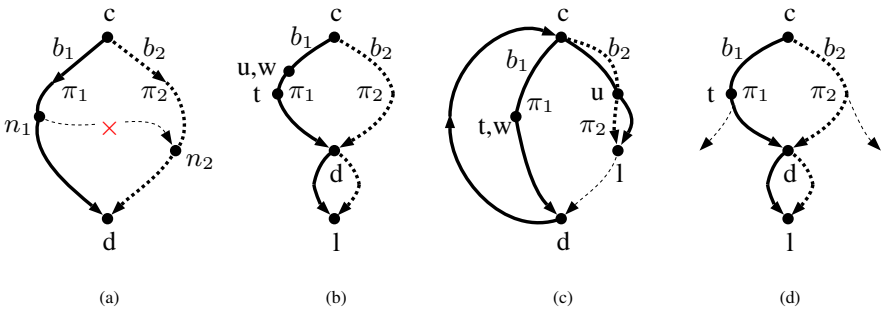


Fig. 3. (a) A property of CFG paths. (b)-(d) Situations that make a predicate value-impacting. In Fig (c), path π_1 is $c \rightarrow t \rightarrow d \rightarrow c \rightarrow u \rightarrow l$.

the sub-traces of P and P^{VS} are weak-SC-equivalent. Therefore the third condition in Definition 1 holds, and we claim that:

Theorem 1. *The abstract subprogram P^{VS} is a* value slice*.*

3.2 Relating Value-Impacting Statements to Data and Control Dependences

Figure 3 shows certain situations that we shall refer to in subsequent discussions. c denotes a predicate having two outgoing edges b_1 and b_2 that start the paths π_1 (solid line) and π_2 (thick dashed line). l denotes the node of the slicing criterion. We begin by mentioning a property of the programs under consideration. In figure (a), d is the first node common to π_1 and π_2. Since our program model does not allow arbitrary jumps, the following property, illustrated in Figure 3 (a), holds:
Prop: Let π_1 and π_2 be disjoint paths from a predicate c to a node d, and let n_1 and n_2 be nodes on these paths distinct from d. Then there cannot exist a path from n_1 to n_2.

It is clear that the most challenging part of value slice computation is the computation of value-impacting predicates. Given a predicate c, we now identify necessary and sufficient conditions for c to value-impact $\Upsilon = \langle l, V \rangle$.

Consider P1 of Fig. 2. The predicate `c2` is value-impacting for the slicing criterion $\langle 15, \{\texttt{x}\}\rangle$. Observe in this case that line 15 is not control dependent on `c2` while the value-impacting assignment at line 14 is control dependent on `c2`. We generalize this observation, illustrated in Figure 3 (b), to obtain the first necessary condition for a predicate c to be value-impacting:

> $cond_1$: If l is not transitively control dependent on c, then a value-impacting node $t \neq c$ is control dependent on c.

Notice that $cond_1$ is also corroborated for the slicing criterion $\langle 17, \{\texttt{y}\}\rangle$, with predicate `c1` as c and the predicate at line 13 as t.

Now consider P3 in which `i<4` is value-impacting for $\langle 14, \{\texttt{x}\}\rangle$. In this case line 14 is transitively control dependent on `i<4` through the true out-edge. The value-impacting assignment for this criterion at line 8 is reachable through the false-edge of predicate

`i<4`, as both are in a cycle $9 \xrightarrow{f} 17 \rightarrow 5 \xrightarrow{t} 7 \rightarrow 8 \rightarrow 9$. The predicate `i%2==0` is also value-impacting for $\langle 14, \{\texttt{x}\}\rangle$, and line 14 is control dependent on `i%2==0` through the false out-edge. Moreover, the value-impacting assignment at line 12 is control dependent on predicate `i%2==0` through the true out-edge. This observation, generalized in Figure 3 (c), gives the second necessary condition for c to be value-impacting:

> $cond_2$: If l is transitively control dependent on c through only one out-edge, say b_2, then there is a value-impacting node $t \neq c$ such that t is not transitively control dependent on c through b_2 and c and t are in a cycle.

There is a third condition $cond_3$ which covers the case when l is transitively control dependent on c through both out-edges, as shown through Figure 3 (d). As mentioned earlier, this happens when some of the branches emanating from a predicate do not merge back due to `return` statements.

> $cond_3$: If l is transitively control dependent on c through both edges, then there is a value-impacting node $t \neq c$ which is transitively control dependent on c through exactly one edge.

Note that the antecedent of exactly one of the three conditions $cond_1$, $cond_2$ and $cond_3$ always holds. Therefore, for the conjunction of these conditions to hold, only the condition with true antecedent needs to hold; the other two will hold vacuously. We will now show that conjunction of $cond_1$, $cond_2$ and $cond_3$ is a necessary and sufficient condition for c to be value-impacting and can thus be used for obtaining a sound and precise value slice. But we first prove a property of the witness of a value-impacting predicate.

Lemma 3. *Let c be a value-impacting node for the slicing criterion $\langle l, V\rangle$ with a witness $\langle \pi_1, \pi_2, u\rangle$. Then, at least one of π_1 or π_2 must have a value-impacting node before any common node appearing on both π_1 and π_2.*

Proof. Let π_1' and π_2' be the disjoint prefixes of π_1 and π_2 ending with a common node d (possibly l itself). Assume that both π_1' and π_2' have no value-impacting statements before d. Obviously, $u \neq d$ otherwise, contrary to our assumption, c will not be value-impacting. Since u is not the first value-impacting on π_2, π_2 must diverge from π_1 after d but before u. The divergence point will have to be a predicate, say c'. It is easy to see that c' will be a value impacting node on π_1 before u, a contradiction. ■

We now show that the conjunction of $cond_1$, $cond_2$ and $cond_3$ is a necessary criterion for a predicate c to be value-impacting.

Lemma 4. *Given a slicing criterion $\Upsilon = \langle l, V\rangle$ and a value-impacting predicate c, conditions $cond_1$, $cond_2$ and $cond_3$ hold.*

Proof.

1. Let $\langle \pi_1, \pi_2, u\rangle$ be the witness for c to be a value-impacting statement for l. Since l is not transitively control dependent on c, the situation must be as depicted in Figure 3 (b), where d is the immediate post-dominator of c. By Lemma 3, at least

one of π_1 or π_2 must have the first value-impacting node w before d. First assume that w lies on the segment of π_1. Obviously, $w = u$ and w must post-dominate the out-edge b_1. In addition, by *Prop*, w can not strictly post-dominate the out-edge b_2. Thus w is the required t. The case of w lying on the segment of π_2 can be argued similarly.

2. Assume that l is transitively control dependent on c through the out-edge b_2 only. Since c is value-impacting, by *Prop*, the situation resembles Figure 3 (c) and the witness is either $\langle \pi_1, \pi_2, w \rangle$ or $\langle \pi_2, \pi_1, u \rangle$. If the witness is $\langle \pi_2, \pi_1, u \rangle$, then there must be a value-impacting node $t \neq u$ in the looping segment c to c of π_1. If the witness is $\langle \pi_1, \pi_2, w \rangle$, then $t = w$ and t must once again be in the c to c segment of π_2. In both cases, t is not transitively control dependent on c through b_2 and is in a cycle with c.
3. Assume that l is transitively control dependent on c through both out-edges b_1 and b_2. Since c is value-impacting, there will be a witness with paths π_1 and π_2 as shown in Figure 3 (d). By Lemma 3, there is a value-impacting statement t on π_1 or π_2 before d. Without loss of generality, we assume that t is on π_1 and it is first value impacting statement on π_1. By *Prop*, t has to be transitively control dependent on c through b_1 and only through b_1. ■

We now show that the conjunction of $cond_1$, $cond_2$ and $cond_3$ is also a sufficient condition.

Lemma 5. *Given a slicing criterion $\Upsilon = \langle l, V \rangle$ and a predicate c such that the conditions $cond_1$, $cond_2$ and $cond_3$ hold, c is value-impacting for Υ.*

Proof. In each case we shall identify a witness for c to be value-impacting for Υ.

1. Assume that l is not transitively control dependent on c and t is control dependent on c through the b_1 edge. Clearly, t post-dominates edge b_1. Consider the first value-impacting statement u between c and t (u may be the same as t). Then the required witness is $\langle \pi_1, \pi_2, u \rangle$ as shown in Figure 3 (b).
2. Assume l is transitively control dependent on c through only one of the edges, say b_2. Also assume that there is a node t that is not transitively control dependent on c through b_2 and that c and t are in a cycle. Then the witness is $\langle \pi_1, \pi_2, t \rangle$, as shown in Figure 3 (c).
3. Now assume that l is transitively control dependent on c through both edges and t is transitively control dependent on c through the b_1 edge only. Then the witness is $\langle \pi_1, \pi_2, t' \rangle$, as shown in Figure 3 (d), where t' is first value impacting node on π_1 and may be same as t. ■

4 Value Slice Computation

Given a program dependence graph (PDG) [10], representing data and control dependences in the program, it is easy to compute value-impacting assignments using Definition 2. In addition, Lemmas 4 and 5 can be used to identify value-impacting predicates. These value-impacting assignments and predicates are augmented with abstract predicates to obtain the value-slice. A minor implementation detail is that a predicate with

```
1: function compVI(l, V)
2: begin
3: lct = tcd(l, true)
4: lcf = tcd(l, false)
5: vi = ∅
6: wl = DU(l, V)
7: while wl is not empty do
8:     choose an element t from wl
9:     ic = iConds(t, lct, lcf)
10:    vi = vi ∪ {t}
11:    wl = (wl \ {t}) ∪
12:         ((ic ∪ DU(LV(t))) \ vi)
13: end while
14: return vi
15: end
```

```
1: function iConds(t, lct, lcf)
2: begin
3: tct = tcd(t, true)
4: tcf = tcd(t, false)
5: dc = cd(t)
6: cnd1 = dc \ (lct ∪ lcf)
7: cnd2^t = (lct \ tct) ∩ incycle(t)
8: cnd2^f = (lcf \ tcf) ∩ incycle(t)
9: cnd3^t = lct ∩ lcf ∩ (tct \ tcf)
10: cnd3^f = lct ∩ lcf ∩ (tcf \ tct)
11: return (cnd1 ∪ cnd2^t ∪ cnd2^f ∪
12:         cnd3^t ∪ cnd3^f)
13: end
```

Fig. 4. Algorithm to compute VI

the reaching definitions of all its variables in VI is retained in concrete form, even if the predicate itself is not in VI. Abstracting the predicate in this case would not result in a decrease in the size of the slice. Note that the precision of the slice depends on the precision of the PDG; given a precise PDG, the value slice exactly matches P^{VS}.

Figure 4 gives an algorithm to compute $VI(\langle l, V\rangle)$. We use $tcd(t, b)$ to denote $\{c \mid c \overset{b}{\rightsquigarrow} t\}$ and $cd(t)$ to denote $\{c \mid c \rightarrowtail t\}$. We compute tcd and cd from the PDG of the program. In addition, $incycle(t)$ is the set of predicates which are in a cycle with t. The worklist wl in the algorithm contains value-impacting statements which have not been explored, i.e. they have not been used to find other value-impacting statements. vi contains value-impacting statements which have been explored. Given a value-impacting statement t, ic is the set of predicates and $DU(LV(t))$ the set of assignments that become value-impacting because of t. ic is computed using the function $iConds$ which encodes $cond_1$, $cond_2$ and $cond_3$ in a straightforward manner. As an example, cnd_1, the encoding of $cond_1$, computes the set of predicates c which become value-impacting because t is directly control dependent on c and l is not transitively control dependent on c.

Assume there are E edges and N nodes in the CFG of which C are predicates. Since a node goes into the worklist at most once, the while loop in $compVI$ iterates at most N times. Further, let there be E_d data dependence and E_c control dependence edges in the PDG, adding to $E_p = E_d + E_c$ edges. The sets lct and lcf can be pre-computed in $O(C)$ time and stored in $O(C)$ space, so that membership of these sets can be checked in constant time. Further, Tarjan's algorithm [18] can be used to find all strongly connected components (SCCs) in a CFG in $O(E + N)$ time, from which we can pre-compute $incycle(t)$. This takes $O(N \times C)$ time and $O(N \times C)$ space. Thus $c \in incycle(t)$ can also be checked in constant time.

It is clear that each data dependent edge will be traversed at most once during the entire run of $compVI$. Similarly, because of dc and cnd_1, each control dependent edge will also be visited at most once during execution of $compVI$. The computation of tct. tcf, cnd_2 and cnd_3 all require $O(C)$ time. So the overall complexity of the algorithm is $O(E+N)+O(N\times C)+O(E_c+E_d) \approx O(N\times C)+O(E_p)$. Note that backward slice

Prg	KLOC	Asserts	Backward Slice			Value Slice				Thin Slice				Scale up (%)			Precision loss (%)	
			Y	N	?	Y	N	N_S	?	Y	N	N_S	?	Back.	Value	Thin	Value	Thin
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)	(q)	(r)	(s)
$icecast$	18	27	3	0	24	8	9	0	10	0	21	8	6	11	63	78	0	38
$navi1$	41	58	39	0	19	38	5	3	15	25	14	10	19	67	74	67	7	26
$navi2$	52	68	44	0	24	52	4	2	12	40	16	10	12	65	82	82	4	18
$navi3$	50	80	59	7	14	55	16	6	9	31	43	32	6	83	89	93	8	43
$navi4$	166	70	17	0	53	28	4	0	38	27	24	7	19	24	46	73	0	14
$navi5$	156	70	16	2	52	24	5	0	41	25	24	10	21	26	41	70	0	20
$navi6$	162	70	25	0	45	42	1	0	27	15	32	26	23	36	61	67	0	55
$navi7$	350	60	11	0	49	18	0	0	42	11	25	3	24	18	30	60	0	8
$navi8$	366	56	20	2	34	38	2	0	16	27	20	12	9	39	71	84	0	26
$navi9$	159	50	13	0	37	22	1	0	27	13	22	13	15	26	46	70	0	37
Average														39	60	74	2	29

Fig. 5. Scalability and precision of property checking based on different kinds of slices. Y and N stand for 'yes' and 'no' answers returned by the property checker. ? stands for 'no decision' and N_S stands for a 'no' that is known to be spurious.

computation has a complexity of $O(E_p)$. Since in the worst case $O(E_p) = O(N \times N)$, the worst case complexity is the same for backward slice and value slice.

5 Implementation and Measurements

We have built a scalable property checking tool based on value slicing[3]. Our implementation supports full version of C including pointers, structures, arrays, heap allocation and function calls. Following custom, the heap is abstracted in terms of allocation points and arrays are summarized to a single abstract element. However, structures are field sensitive: `x.a` and `x.b` are treated as separate entities. Pointers are handled using a flow sensitive but context insensitive points-to analysis. We first construct an intraprocedural PDG for each function, using the algorithm of Billardi and Pingali [4] to construct the control dependence graph. The PDGs are then linked and interprocedurally valid data and control dependences computed using the method by Horwitz et. al. [12].

We carried out our experiments on 3.0 GHz Intel Core2Duo processor with 2 GB RAM and 32 bit OS. We chose SATABS (version 3.0) [8] as the verifier for its robustness and its scalability. We experimented on one open source application, $icecast$, and 60 modules of varying sizes of a proprietary code base of a large automotive navigation system, grouped into nine groups: $navi1$ to $navi9$. Average size of individual modules in these groups varied from 6 KLOC to 61 KLOC. We checked for the "array index out of bounds" property on these programs. The size and the number of asserts for each group of program are shown in the table. For each chosen instance, we computed backward slice, value slice and thin slice. All three slices were submitted to SATABS, with a time-out limit of three minutes and three kinds of outcomes were recorded: Property

[3] Implemented on top of PRISM, a static analyzer generator developed at TRDDC, Pune [14,7].

satisfied (Y), property failed (N), and no decision($?$). The possible reasons for the last outcome are time-out, too many iterations, or SATABS failing due to some other causes.

The Y answers of all three slices are correct by construction of the slice. Similarly, an N answer for the backward slice is also correct. However, in case of a value or thin slice, if an assert with an N answer is also recorded as a Y during property checking with the other two slices, it is also recorded as being a spurious N (N_S in the table). Scalability, given by $(Y + N)/(Y + N+?)$ is the ratio of definite outcomes over all outcomes. Loss of precision is the ratio of outcomes that are known to be spurious over all definite outcomes ($N_S/(Y + N)$). The results are presented in Figure 5.

From the results, it is obvious that both value and thin slice help in scaling up property checking, with thin slice having a small advantage (14%) over value slice. However, compared to the backward slice, the precision drops considerably (29%) in the case of thin slice, while there is only a marginal drop (2%) for value slice. This implies that refinement will be required in many more cases with thin slice as compared to value slice. We also expect refinement cycles to be shorter for value slice because of fewer abstractions. This shows that value slice is a good compromise between backward and thin slices as it provides considerable scalability with only a marginal loss in precision.

6 Related Work

Following the introduction of backward slicing by Weiser [19], several variations of slicing have been proposed. Notable among these are forward slicing [3], chopping [13], and assertion based slicing [6,9,2]. Restricted to the slicing criterion, all these techniques produce slices with behaviours equivalent to the original program. Dynamic slice [15] matches the behaviour of the original program for a run over a specific input.

Thin slicing [17], used for debugging, is the first approach that produces a slice whose behaviour differs from the original program with respect to the slicing criterion. A thin slice retains only those statements that the variables in the slicing criterion are data dependent on and abstracts out all predicates. This approach comes closest to our method. While this results in smaller slices, our experiments show that the slices are too imprecise for property checking. Interestingly, the authors do mention the importance of identifying the predicates that we include in the value slice in a concrete form. However it is done manually during debugging.

7 Conclusion

Slicing is an obvious pre-processing step before submitting a program to a verifier for property checking. For this purpose, backward slice has been the choice so far, since its behaviour exactly matches the behaviour of the original program with respect to the property being checked. In this paper, we have suggested a more aggressive form of slicing called value slice which slices out statements affecting reachability of the assertion point and retains just those statements which influence the values of the property variables. Property checking with value slice is more scalable than backward slice. However, our method also carefully identifies and retains certain predicates due to which property checking with value slice is more precise than an even more aggressive form

of slicing called thin slice. Indeed, our experiments show that on both axes of comparison, scalability and precision, value slice based property checking comes close to the best performer of the three comparable forms of slicing that we have considered.

An overall property checking process could include refinement steps on getting a failure answer. If the counterexample generated by the verifier turns out to be spurious, one can use its trace to choose an abstract predicate that can be concretized. At worst, the refinement process could end in a backward slice. An alternate single step refinement process could use the backward slice directly to determine whether the negative answer is genuine. Our experiments also show that the size of the value slice is on the average about 50% of the size of the backward slice. Thus value slices can also be used for program understanding and debugging.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. Pearson Education, Inc. (2006)
2. Barros, J.B., da Cruz, D., Henriques, P.R., Pinto, J.S.: Assertion-based slicing and slice graphs. In: Proceedings of SEFM (2010)
3. Bergeretti, J.-F., Carré, B.A.: Information-flow and data-flow analysis of while-programs. ACM Trans. Program. Lang. Syst. 7(1), 37–61 (1985)
4. Bilardi, G., Pingali, K.: A framework for generalized control dependence. In: Proceedings of PLDI (1996)
5. Binkley, D.W., Gallagher, K.B.: Program slicing. Advances in Computers 43, 1–50 (1996)
6. Canfora, G., Cimitile, A., De Lucia, A.: Conditioned program slicing. Information & Software Technology 40(11-12), 595–607 (1998)
7. Chimdyalwar, B., Kumar, S.: Effective false positive filtering for evolving software. In: Proceedings of ISEC (2011)
8. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
9. Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: Gaudel, M.-C., Wing, J.M. (eds.) FME 1996. LNCS, vol. 1051, pp. 557–575. Springer, Heidelberg (1996)
10. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
11. Gold, N., Harman, M.: An empirical study of static program slice size. ACM Trans. on Software Engineering and Methodology (TOSEM) 16 (2007)
12. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. SIGPLAN Not. 23, 35–46 (1988)
13. Jackson, D., Rollins, E.J.: Chopping: A generalization of slicing. Technical report, Pittsburgh, PA, USA (1994)
14. Khare, S., Saraswat, S., Kumar, S.: Static program analysis of large embedded code base: an experience. In: Proceedings of ISEC (2011)
15. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. 29(3), 155–163 (1988)
16. Silva, J.: A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44(3), 1–41 (2012)
17. Sridharan, M., Fink, S.J., Bodik, R.: Thin slicing. In: Proceedings of PLDI (2007)
18. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. 1(2), 146–160 (1972)
19. Weiser, M.: Program slicing. In: Proceedings of ICSE (1981)