Property Driven Program Slicing

Agostino Cortesi, Sukriti Bhattacharya
Dipartimento di Informatica
Università Ca' Foscari di Venezia, Venezia, Italy

Abstract

In this paper we are trying to combine the traditional proram slicing technique proposed by Mark Weiser[1] along with the static analysis and dataflow analysis results on a given program to remove the statements that do not affect a given property. Hence we named it Property Driven Program Slicing. In order to do that we apply a forward static analysis to extract properties of each variable in each program point, followed by a backward slicing algorithm to collect the slice based on the static analysis based informations.The aim of the present paper is to further refine the traditional slicing technique by combining it with abstract interpretation and data flow analysis.This results into a deeper insight on the strong relation between slicing and property based dependency.

1 Introduction:

Program slicing is a technique to extract program parts with respect to some special computation. Since Mark Weiser [1] first proposed the notion of slicing in 1979, many applications of this technique have been studied in the literature. Slicing was first developed to facilitate debugging, but it has been found helpful in many aspects of the software development life cycle, including software testing, software measurement, program comprehension, software maintenance, software reengineering, program parallelization and so on [2]. Different tools have also been developed, like CodeSurfer, Unravel, Indus, Bandera, and Kaveri, just to name a few.

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.

The aim of the present paper is to further refine the traditional slicing technique by combining it with abstract interpretation and data flow analysis [3,4]. This results into a deeper insight on the strong relation between slicing and property based dependency.

The idea can be summarized as follows. Very often, we are interested on a specific property of the variables in the slicing criterion C, not on their exact actual values. For instance, when condering security properties, it may be sufficient to consider just the parity of the variable containing a cyphertext, or just its sign, or just if it stays into a given range. Therefore, when performing slicing, not only the statements that do not affect Vars(C) can be discarded, but also the statements that do not have any impact, respectively, on their parity, on their sign or on their range.

The resulting proposal is a fixed point computation where each iterate has two phases. First, by a forward (dataflow) static analysis, the control flow analysis of the program is enhanced with information about the state of variables with respect to the property of interest. Then, following a backward computation, we enhance the slicing technique by using such additional information to identify relevancy of the statement to be kept (or removed) in the sliced program.

2 Preliminaries:

Given a program P, we denote by Vars(P) the set of program variables in P and by exp(Vars(P)) the set of expressions in P. We denote the expression evaluation by eval: exp(Vars(P))$\rightarrow$ Values, A property, ψ is a decidable boolean function $\wp$((Vars(P)),eval) $\rightarrow$ {True, False}.

Definition 2.1 (Congruence with respect to a given property)

Given two programs P_1 and P_2 and a set K$\in$ Vars(P_1) $\cap$Vars(P_2), P_1 and P_2 are said to be K-congruent with respect to ψ if for each execution of P_1 and P_2 with the same input, the final value of the variables in K are the same.

Example:

Consider for instance

P_1: *begin input(x); x=2*x; y=5; z=x+y*; *end*, and

P_2: *begin input(y); x=4; y=x/3+2; z=x+1*; *end.*

$P_1$is {x,y,z} congruent to P_2 with respect to *parity*, since at the end of the computation both programs behave the same wrt parity: x is even, y is odd and z is odd.

Definition 2.2 (Property driven program slicing)

Let P be a program, s be a statement of P, and V$\subseteq$ Vars(s). For statement s and a given property ψ, the slice P' of program P with respect to the slicing criterion (s,V) and ψ, is an executable program such that:

1. P'can be obtained by deleting zero or more statements from P.
2. P' is V-congruent to P with respect to ψ.

Example:

Consider, for instance, the following C program fragment , We want to slice P with respect to the slicing criteria <5;{c}> and the *sign* property.

Stmt. NO.	Original Program (P)	Sliced program
1	scanf(%d,"&b"); /* b≠0 */	scanf(%d,"&b");
2	b=b*(-2);	b=b*(-2);
3	c=b*2;	c=b*2;
4	d=d+1;	
5	d=c/b;	d=c/b;
6	c=d;	c=d;

Statement 4 in P is removed in the slice P′, since it is no more relevant when considering the *sign* property and the slicing criterion, as the sign of c in line 6 is not affected by such statemet.

3 Algorithm:

At the macro level, our algorithm is fixed point iteration calternating analysis and slicing, starting form a program P and a slicing criterion C and a property ψ:

```
loop {
        P_old=P;
        A = Analysis(P, ψ);
        P′= Slicer(P, C, ψ, A);
      }until P′=P_old
```

More in detail, *Slicer* considers P as an ordered set of statements (all statements that should not be removed), and the set N to hold all variables that have been used. *Slicer* performs a fixpoint iteration: by using information collected in the analysis phase.

For any statement s, *STMT(s)* denotes the succeeding statements.After the inizialization phase,statements possibly affecting a variable in N are added to P, and the used variables of these statements are added to N, until these sets do not grow any more. Clearly, this process terminates since there are only a finite number of variables and assignments in a program.Variables may be read or written through pointers. We assume, for each statement s and pointer variable p, that there is a points-to [5] set *PTS(p, s)* which contains the set of variables possibly pointed to by p in the statement s. The *DEFS* function calculates a set of locations where the result of an assignment may be stored. If the left-hand side of the assignment contains de-referenced pointers, then *DEFS* uses *PTS* to calculate possible locations where the result might be stored.$LIVE_s^{out}$ holds the variables live [4], after the execution of statement s.

Each program flow graph edge has an associated flag, the *ExecutableFlag*. This flag is initially *FALSE* for all edges. ExecutableFlag value of program flow graph edges are marked *TRUE* by symbolically executing the program, beginning with the start node. Whenever an assignment node is executed, the *ExecutableFlag* value of the out-edge in the program flow graph leaving that node is marked as *TRUE* and added to the worklist. Whenever a conditional node is executed, the expression controlling the conditional is evaluated and we determine which branch (es) may be taken. If the expression evaluates to $\perp$ (not predictable) then all branches may be taken. The *ExecutableFlag* value corresponding to these branches are set to be *TRUE*. Otherwise, only one branch can be taken, and the associated *ExecutableFlag* value is set to be *TRUE*.

The Slicer algorithm can be sketched as follow:

```
Slicer(P$_0$, C, $\psi$, A={PTS,DFG$^*$})
INPUT:
P$_0$: The code to be sliced (set of statements)
C: The slicing criterion <n,V>
$\psi$: A given property
PTS: Mapping from pointer variables and statement numbers to points-to-sets
DFG$^*$: Dataflow graph of the program where each node associated with
      $\Pi_v^{in}(\psi_s)$ (variables' value before executing statement s wrt $\psi$)
      $\Pi_v^{out}(\psi_s)$ (variables' value after executing statement s wrt $\psi$)
OUTPUT:
P: A program slice which is V-congruent with $\psi$ and s.
BEGIN
N=V
P={n}
N$_{old}$={$\emptyset$}
while N $\not\supseteq$Nold do
   N$_{old}$ = N
   for each statement s from n to 1 in P$_0$ do
     if ExEflg(STMT(s)) = = TRUE then
       for each v $\in$DEFS (PTS,s,STMT(s)) do
         if v$\in$ N $\wedge\Pi_v^{in}(\psi_s)\neq\Pi_v^{out}(\psi_s)$ $\wedge$ v $\in LIVE_s^{out}$ then
            P = P$\cup${s}
            N= N $\cup$ USES(PTS, s, STMT(s))
         endif
        endfor
     endif
   endfor
```

Termination:

This process terminates since there are only a finite number of variables and assignments in a program and at each step of the while loop the set N is strictly increasing and is bounded by Vars(P).

Correctness:

The correctness of this algorithm relies on the following facts

- Underlaying static analysis
- The traditional slicing algorithm [1] is assumed to be correct.
- SLICE $_{Property}$$\sqsubseteq$SLICE$_{Tradition}$
  $\forall$statement s $\in$SLICE$_{Tradition}$ $\setminus$ SLICE $_{Property}$ the impact on ψ(s)=NULL where SLICE$_{Tradition}$represents the slice generated by tradition slicing algorith proposed by Mark Weiser [1] on a program P. SLICE$_{Property}$ represents the slice generated by property driven slicing algorithm proposed here on the same program program P.

4 Conclusions:

This work combines static analysis and program slicing, i.e it refines slicing with respect to a property of interest. It can also be combined with the abstract slicing approach in [6]. We also notice one thing, *Conditional Constant Propagation* in this context can be helpful to get more precise slice[7].An implementation of the complete algorithm for C programs, where simple properties like sign and partity are considered, has already been developed, and preliminary experimental results confirm the scalability of the algorithm.

References

[1] M. Weiser. *Program slicing.* In ICSE '81: Proceedings of the 5th international conference on Software engineering, pp. 439-449, Piscataway, NJ, USA, 1981. IEEE Press.

[2] F. Tip. *A survey of program slicing techniques.* Journal of Programming Languages, vol. 3, pp. 121-189, 1995.

[3] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., 1999.

[4] Patrick Cousot, Radhia Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.* In Conference Record of the Sixth Annual ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, pp. 238-252, Los Angeles, California, 1977. ACM Press, New York.

[5] Lars Ole Andersen . *Program Analysis and Specialization for the C Programming Language* .Ph.D. Thesis , DIKU, University of Copenhagen,May 1994, pp. 111-120.

[6] I. Mastroeni, D. Zanardini, *Data Dependencies and Program Slicing: from Syntax to Abstract Semantics*, Proc. "ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation", San Francisco, CA, USA , 7-8 Gennaio 2008 , 2008 , ACM press, pp. 125-134 .

[7] Mark N.Wegman F. Kenneth Zadech .Constant propagation with conditional branches. ACM, TOPLAS, April, 1991. pp. 183-193.