Program Slicing

Stephan Nawracki

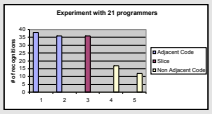
Contents

Introduction

What is program slicing?
Conditions for a program slice
Why is it useful?

What is program slicing?

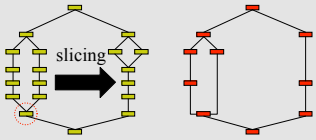
- Analysis technique introduced by Mark Weiser in his PHD thesis (1979)
  - Idea derived when he was observing experienced programmers debugging a program
  - Result: Every experienced programmer uses slicing to debug a program



- Slicing reduces programs to statements relevant for partial computation
  - Irrelevant statements are deleted
- A slice S includes all program statements affecting variables V at position n in Program P
  - Eg the slicing criteria S(a, 10) is a slice including all statements affecting the value of a in line 10

What is program slicing? Cont'd

- Program slicing describes a mechanism which allows the automatic generation of a slice
- All statements affecting the variables mentioned in the slicing criterion becomes a part of the slice
- Variables V at statements s can be affected by statements because:
  - Statements define whether s is executed at all (control dependence)
  - Statements define a variable that is used by s (data dependence)



Source Program Sliced Program

Conditions for a program slice

- A slice S(V,n) is derived from a Program P by deleting statements from P
- The slice must be syntactically correct in terms of the programming language used in P
- The values for variables V received from the slice at statement s have to be the same as the values for V at statement s in program P
- Weiser:
  "First, the slice must have been obtained from the original program by statement deletion. Second, the behaviour of the slice must correspond to the behaviour of the original program as observed through the window of the slicing criterion"

Why is it useful?

- Program slicing is useful in many different stages of the software development lifecycle e.g.
  - Debugging:
    - Slicing visualizes control and data dependencies
    - It highlights statements influencing the slice
  - Testing:
    - Tests may be decomposed and test-work gets faster and more efficient
  - Software quality assurance
    - Safety critical code can be isolated and functions can be implemented redundant and in functional diversity manner
  - Maintenance, …

UCL ComputerScience

Variants of program slicing

Overview
Examples

UCL ComputerScience

Overview

- Many different variants of program slicing exist e.g.
  1. Static slicing
  2. Dynamic slicing
  3. Backward slicing
  4. Forward slicing
  5. Condition or quasi static slicing
  6. Chopping
  7. Interface slicing
- Also many different tools, however
  - Most program slicing tools are written for C but there are also some for C++, COBOL, FORTRAN and Java
  - Most of these have problems with dynamic binding, inheritance, polymorphism and performance (see chapter program slicing software)

UCL ComputerScience

Overview Cont'd

- Static slicing
  - Slices derived from the source code for all possible input values
  - No assumptions about input values.
  - May lead to relatively big slices
  - Contains all statements that may affect a variable for every possible execution
- Dynamic slicing
  - Uses information derived from a particular execution of a program
  - Execution is monitored and slices are computed with respect to program history
  - Relatively small slices
  - Contains all statements that actually affect the value of a variable
- Conditional or quasi static slicing
  - acts as a bridge between the two extremes of static and dynamic slicing
- Backward slicing
  - Contains the statements of the program P which may have some effect in the slicing criterion S(V,n)
- Forward slicing
  - Contains all those statements of P which are affected by S(V,n)

UCL ComputerScience

Example Backward slicing

```
Pass = 0; //There is a bug in the program. Average isn't calculated correctly
Fail = 0;
Count = 0;

while (!eof()) {
        TotalMarks=0;
        scanf("%d",Marks);
        if (Marks >= 40)
                Pass = Pass + 1;
        if (Marks < 40)
                Fail = Fail + 1;
        Count = Count + 1;
        TotalMarks = TotalMarks+Marks ;
}
        printf("Out of %d, %d passed and %d failed\n",Count,Pass,Fail) ;
        average = TotalMarks/Count;
        printf("The average was %d\n",average) ;  // This is the point of interest
        PassRate = Pass/Count*100 ;
        printf("This is a pass rate of %d\n",PassRate) ;
```


UCL ComputerScience

Example Forward slicing

Original:

```
x = 1; /* what happens when this line is changed */
y = 3;
p = x + y ;
z = y -2 ;
if (p==0)
r++ ;
```

Forward slice:

```
/* Change to first line will affect */
p = x + y ;
if (p==0)
r++ ;
```


UCL ComputerScience

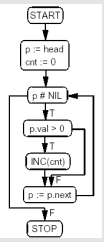
Implementation of program slicing

Introducing control flow graphs
Program slicing as a data flow problem

Introducing control flow graphs

- Control flow graphs are used for data flow analysis
- A control flow graph consists of
  - A set of nodes
  - A set of directed edges
  - A unique entry node START
  - A unique exit node STOP

```
p := head; cnt := 0
WHILE p # NIL DO
   IF p.val > 0 THEN
      INC(cnt)
   END
   p := p.next
END
```



Program slicing as a data flow problem

- Weiser used a control flow graph as an intermediate representation for his slicing algorithm
  - This control flow graph is used for data flow analysis
- A data flow describes change/flow of values of variables from the point of definition to the point they are used
- The relevant parts for slice S are calculated in four steps:

1. Initialize the relevant sets of all nodes to the empty set.
2. Insert all variables of *V* into *relevant(n)*.
3. For *n*'s immediate predecessor *m*, compute *relevant(m)* as:
   relevant(m) := relevant(n) - def(m) (* *exclude all variables that are defined at*m *)

   if relevant(n) _ def(m) # {} then (* *if* m *defines a variable that is relevant at*n *)
      relevant(m) := relevant(m) _ ref(m) (* *include the variables that are referenced at*m *)
      Include m into the slice
   End
4. Work backwards in the control flow graph, repeating step 3 for *m*'s immediate predecessors until the entry node is reached or the relevant set is empty.

Example

- This is a … d has to be extended
  - Proced…
  - Classes…
  - Inherita…
  - …

n	Statement	ref(n)	def(n)	relevant(n)
1	b = 1		b	
2	c = 2		c	b
3	d = 3		d	b, c
4	a = d	d	a	b, c
5	d = b + d	b, d	d	b, c
6	b = b + 1	b	b	b, c
7	a = b + c	b, c	a	b, c
8	print a	a		a

Table 3.1 - Source code with relevant sets, slice for <8, {a}>

Step 2: relevant(8) = {a}
Step 3: relevant(7) = relevant(8) - def(7) = {a} - {a} = {}
relevant(7) = relevant(7) ∪ ref(7) = {} ∪ {b, c} = {b, c}
Since node 7 defines a variable relevant at node 8, it is included into the slice.
Step 3: relevant(6) = relevant(7) - def(6) = {b, c} - {b} = {c}
relevant(6) = relevant(6) ∪ ref(6) = {c} ∪ {b} = {b, c}
Since node 6 defines a variable relevant at node 7, it is included into the slice.
Step 3: relevant(5) = relevant(6) - def(5) = {b, c} - {d} = {b, c}
Step 3: relevant(4) = relevant(5) - def(4) = {b, c} - {a} = {b, c}
Step 3: relevant(3) = relevant(4) - def(3) = {b, c} - {d} = {b, c}
Step 3: relevant(2) = relevant(3) - def(2) = {b, c} - {c} = {b}
relevant(2) = relevant(2) ∪ ref(2) = {b} ∪ {} = {b}
Since node 2 defines a variable relevant at node 3, it is included into the slice.
Step 3: relevant(1) = relevant(2) - def(1) = {b} - {b} = {}
relevant(1) = relevant(1) ∪ ref(1) = {} ∪ {} = {}
Since node 1 defines a variable relevant at node 2, it is included into the slice.

UCL ComputerScience

Program slicing software

Introduction and short description

UCL ComputerScience

Unravel

- By John Lyle, Dolores Wallace, James Graham, Keith Gallagher, Joseph Poole, David Binkley
- Runs on Sun Sparc
- Slices programs in ANSI C
- Has some restrictions (e.g. no goto statements)
  - Just backward slice at the moment
- Performs work in reasonable time

Homepage: http://hissa.nist.gov/unravel/

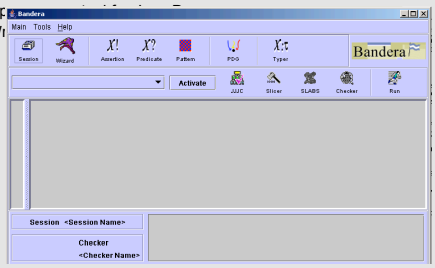
UCL ComputerScience

Wisconsin program slicer

- Was used for C but no longer maintained
  - However commercial tool Codesurfer (http://www.grammatech.com/products/codesurfer/index.html) is derived from the Wisconsin program slicer
- Developed and tested on Sun Sparc
- Forward/backward-slicing, chopping, building and manipulating control flow graphs and program dependency graphs
- Homepage:
- http://www.cs.wisc.edu/wpis/slicing_tool/

Bandera

- Op
- Wr



References

- Kepler, Johannes: “Program Slicing for Object Oriented Programming Languages”
- Weiser, Mark: “Program Slicing”, IEEE Transactions on Software Engineering 10(4):352-357. 1984
- Bandera Manual: „Model-checking Java Programs“
- http://www.brunel.ac.uk/~csstmmh2/exe1.html
- http://bandera.projects.cis.ksu.edu
- http://hissa.nist.gov/unravel/
- http://www.grammatech.com/products/codesurfer/index.html
- http://www.cs.wisc.edu/wpis/slicing_tool/