Program Slicing and Its Applications

Slides taken from Prof. Rajib Mall
Outline

- Introduction
- Program Models
- Slicing Algorithms
- Modelling Issues:
  - Object-orientation
  - Exception handling
  - Concurrency
- Model Slicing
- Directions of research
A programmer frequently ponders:

• If I change this statement:
  • What other statements would be affected?
• The values live at this statement:
  • Been defined where?
• How can I abstract code?
An Example Scenario

Consider a 5000 line program
- Line 1347 assigning a value to a variable `var` is modified

Impact on the program...
- Some results and predicates using `var` may now fail...

How to ensure correctness?
- Manual checking
- Tedious and error prone

Try
- Program Slicing!
How Bad is Manual Analysis?

- Studies show:
  - Maintainers spend nearly 50% of their time trying to understand the program.
  - No wonder maintenance is so expensive!
Program Slicing

- Proposed by Mark Weiser in 1981
- Constructs a program abstraction:
  - Inspired by the mental abstractions that programmers unconsciously construct while debugging.
  - Which statements can affect the computation of a variable v at a statement s?
A Few Basic Concepts

- A slice with respect to a slicing criterion (SC) consists of:
  - All statements that might affect SC

- Slicing Criterion (SC) $<p, V>$
  - $p =$ point of interest in the program
  - $V =$ subset of variables used in the program
First Slice Example

Slice

Source program

Resulting Slice

Relevant

Slicing Criterion
Second Slicing Example

\[
\begin{align*}
1 & \text{ begin} \\
2 & \text{ read}(x, y) \\
3 & \text{ total} := 0.0 \\
4 & \text{ sum} := 0.0 \\
5 & \text{ if } x \leq 1 \\
6 & \text{ then } \text{ sum} := y \\
7 & \text{ else begin} \\
8 & \text{ read}(z) \\
9 & \text{ total} := x \times y \\
10 & \text{ end} \\
11 & \text{ write}(\text{total, sum}) \\
12 & \text{ end}
\end{align*}
\]

- SC = \( <11, \text{sum}> \)

\[
\begin{align*}
2 & \text{ read}(x, y) \\
4 & \text{ sum} := 0.0 \\
5 & \text{ if } x \leq 1 \\
6 & \text{ then } \text{ sum} := y
\end{align*}
\]
Some Insights

- A slice deletes zero or more statements.

- Behavior of a slice execution:
  - Identical to the original program:
  - As far as slicing criterion is concerned!
  - A program abstraction!
Slice Variants

- Extensions to the original concept of a slice
  - Types of slices:
    - Static
    - Dynamic
  - Direction of slice:
    - Forward
    - Backward
  - Executability of slice:
    - Executable
    - Non-executable
  - Amorphous, etc.
Static and Dynamic Slices

- **Static Slice**
  - Set of all statements that may affect the value of a variable at a program point:
    - For all possible input values.

- **Dynamic Slice**
  - Set of all statements that actually affect the value of a variable at a program point:
    - For a particular execution.
    - Specific input values
    - A specific execution path
Example

```c
int i, sum, prd;
1. read(i);
2. prd = 1;
3. sum = 0;
4. while (i < 10)
5. sum = sum + i;
6. prd = prd * i;
7. i = i + 1;
8. write(sum);
9. write(prd);
```

- **Static Slice**
  - `{1, 2, 4, 6, 7}`

- **Dynamic Slice**
  - Input i = 15
  - `{2}`

- SC = `<9, prd>`
Static and Dynamic Slices

- Static slices are **usually larger**:
  - May contain statements which may have nothing to do with the slicing criterion!

- Which one is more suited to debugging?
  - Dynamic

- Is static slicing required at all?
  - Yes
  - Complexity metrics
  - Regression test selection
  - Parallelization…
Slicing Direction

- **Backward Slice**
  - Statements which *might affect* the value of variables in V.

- **Forward Slice**
  - Statements which *might be affected* by the values of variables in V.
Example

```c
int i, sum, prd;
1. read(i);
2. prd = 1;
3. sum = 0;
4. while (i<10)
   5. sum = sum + i;
   6. prd = prd * i;
   7. i = i + 1;
8. write(sum);
9. write(prd);
```

- SC = <2, prd>
  - Forward Slice
  - {6, 9}

- SC = <9, prd>
  - Backward Slice
  - {1, 2, 4, 6, 7}
Executability

- An executable slice:
  - Can be compiled and run
  - A program abstraction:
    - Execution behavior of the original program and its slice are indistinguishable.

- Weiser’s original slice definition:
  - Static, backward, and executable.
Relevant Slice

- **Union of two sets:**
  - Statements affecting variable V
  - Predicates that could affect V
    - If they had evaluated differently

- **Input**
  - \((a, b, c) = (6, 5, 4)\)

- **Code Snippet**

```
S1:   read(a, b, c);
S2:   class = scalene;
S3:   if a = b or b = a
S4:   class = isosceles
S5:   if a*a = b*b + c*c
S6:   class = right
S7:   if a = b and b = c
S8:   class = equilateral
S9:   case class of
S10:  right     : area = b*c/2;
S11:  equilateral: area = a*a*sqrt(3)/4;
S12:  otherwise : s = (a + b + c)/2
S13:   area = sqrt(s*(s-a)*(s-b)*(s-c));  
       end
S14:   write(class, area);
```
Amorphous Slice

- Traditional program slicing is syntax preserving.

- For applications such as re-engineering and program comprehension:
  - Syntax retention is not required
  - Retaining semantic property is more important

- Amorphous slice:
  - Not syntax preserving, works on transformed program code.
  - Does not distinguish static and dynamic slices
Example of Amorphous Slice

```
for(i = 0, sum = a[0], biggest = sum; i<19; sum = a[++i])
    if (a[i+1] > biggest)
        { 
            biggest = a[i+1];
            average = sum/20;
        }
```

- Traditional slice on **biggest**
  - All but the last statement.

- Amorphous slice
  ```
  for(i = 1, biggest = a[0]; i<20; ++i)
    { 
        if (a[i]>biggest)
            biggest = a[i];
    }
  ```

Involves loop unrolling.
### Example of Amorphous Slice

<table>
<thead>
<tr>
<th>Original Program P</th>
<th>Traditional Static Slice of P</th>
<th>Amorphous Slice of P</th>
</tr>
</thead>
<tbody>
<tr>
<td>int f(int a[])</td>
<td>int f(int a[])</td>
<td>int f(int a[])</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int sum;</td>
<td>int sum;</td>
<td>int average;</td>
</tr>
<tr>
<td>int i;</td>
<td>int i;</td>
<td>for(i=0,sum=a[0];</td>
</tr>
<tr>
<td>int biggest;</td>
<td></td>
<td>i&lt;20;</td>
</tr>
<tr>
<td>int average;</td>
<td></td>
<td>sum = a[++i])</td>
</tr>
<tr>
<td>for(i=0,sum=a[0],biggest=sum;</td>
<td></td>
<td>for(i=0,sum=a[0];</td>
</tr>
<tr>
<td>i&lt;20;</td>
<td></td>
<td>i&lt;20;</td>
</tr>
<tr>
<td>sum = a[++i])</td>
<td></td>
<td>sum = a[++i])</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>{</td>
</tr>
<tr>
<td>if(a[i+1]&gt;biggest)</td>
<td></td>
<td>int average;</td>
</tr>
<tr>
<td>biggest = a[i+1];</td>
<td></td>
<td>for(i=0,sum=a[0];</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>i&lt;20;</td>
</tr>
<tr>
<td>average = sum/20;</td>
<td></td>
<td>sum = a[++i])</td>
</tr>
<tr>
<td>printf(&quot;biggest %d sum %d average %d&quot;, biggest, sum, average);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Source: Binkley - Sluder
Amorphous Slicing

- Identification of the bug in computing **average**:  
  - Traditional slice is not very helpful!
- Amorphous slice is more helpful:  
  - Shows that **average** is not computed correctly  
  - Does not rely on variable **sum**
Inter-procedural Slice

```c
int add(int x, int y) {
    return x + y;
}

int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("%d\n", sum);
    printf("%d\n", i);
}
```

Find backward slice with respect to `printf("%d\n", i)`

Source: Thomas Reps
Weiser’s Program Slicing

Steps:

- Construct CFG
- Construct slice through analysis of:
  - REF, DEF, and relevant sets of each statement
  - For straight line code, computed in single pass
  - Hausler showed, for programs with loops, number of passes equals to number of assignment statements.
- Inefficient
Model-Based Program Analysis

Program code analysis:
- Inefficient.

Model analysis is advantageous:
- Easier to identify relationships
  - Flow, control and data dependence, etc.
- More efficient
- Independent of programming language
Later developments

Later researchers proposed more efficient techniques:

- Through construction of appropriate program models.
Popular Program Models

- Flow-based graph models:
  - Capture control flow relationships among statements.

- Dependency-based graph models
  - Capture dependencies among program statements.
  - Two notions of dependency:
    - Data dependency
    - Control dependency
Flow Graph

- **Directed graph** $G = (V, E)$
  - $V$ = finite and non-empty set of nodes
  - $E$ = set of edges, and $E \subseteq N \times N$

- **Flow graph** $G_F = (V \cup Start \cup Stop, E)$
  - For $Start$ node in-degree is 0
    - Indicates start of program execution
  - For $Stop$ node out-degree is 0
    - Indicates end of program execution
  - $\exists$ a path from $Start$ to every other node in $G_F$
  - $\exists$ a path from every other node in $G_F$ to $Stop$
Control Flow Graph (CFG)

- CFG is a directed graph:
  - Each node represents a program statement
  - An edge represents a possible flow of control in the direction of the arrow.

- Some nodes can have more than one outgoing edges:
  - E.g. Selection and iteration constructs
  - Edges are labeled **TRUE** or **FALSE**
int a, b, sum;
1. read(a);
2. read(b);
3. sum = 0;
4. while(a<8){
5.    sum = sum + b;
6.    a = a + 1;
}
7. write(sum);
8. sum = b;
9. write(sum);
Data Dependence

- Node \text{n} is data dependent on \text{m}:
  - If there exists a variable \text{var} such that
    - \text{m} defines \text{var}
    - \text{n} uses \text{var}
    - There exists a directed path from \text{m} to \text{n} along which there is no intervening definition of \text{var}

- Data dependence graph \( G = (N, E) \)
  - \((x, y) \in E \) iff \text{x} is data dependent on \text{y}
int a, b, sum;
1. read(a);
2. read(b);
3. sum = 0;
4. while(a<8)
5. \[ \text{sum} = \text{sum} + b; \]
6. \[ a = a + 1; \]
7. write(sum);
8. sum = b;
9. write(sum);
Dominance

Let $x$ and $y$ be two nodes in a flow graph

$x$ dominates $y$ iff

- every path from Start to $y$ passes through $x$

$y$ post-dominates $x$ iff

- every path from $x$ to Stop passes through $y$

$x$ is immediate post-dominator of node $y$ iff

- $x$ is a post-dominator of $y$
- $x \neq y$
- Each post-dominator $z \neq x$ of $y$ post-dominates $x$
int a, b, sum;
1. read(a);
2. read(b);
3. sum = 0;
4. while(a<8)
5. sum = sum + b;
6. a = a + 1;
7. write(sum);
8. sum = b;
9. write(sum);

- Node 2 dominates node 4, node 6, node 8
- Node 8 post-dominates node 3, node 6, and node 7
Control Dependence Graph

- For a program P, suppose
  - G is its CFG
  - x and y are nodes in G
- y is control dependent on node x if
  - There exists a directed path Q from x to y
  - y post-dominates every z in Q (excluding x and y)
  - y does not post-dominate x
Control Dependence Graph

- Control Dependence Graph (CDG) $G = (N, E)$
  - $(x, y) \in E$ iff $x$ is control dependent on $y$
int a, b, sum;
1. read(a);
2. read(b);
3. sum = 0;
4. while(a<8){
5.   sum = sum + b;
6.   a = a + 1;}
7. write(sum);
8. sum = b;
9. write(sum);
Program Dependence Graph

- Program dependence graph (PDG):
  - Union of CDG and DDG
  - Two kinds of dependency edges exist:
    - Control
    - Data

- A PDG can model programs with a single function:
  - Suited for intra-procedural slicing
int a, b, sum;
1. read(a);
2. read(b);
3. sum = 0;
4. while(a<8){
5.   sum = sum + b;
6.   a = a + 1;
}
7. write(sum);
8. sum = b;
9. write(sum);
System Dependence Graph

- PDG cannot model procedure calls:
  - Not suited for inter-procedural slicing

- Horowitz proposed System Dependence Graph (SDG) model:
  - Models a program with procedure calls
  - Extension of a PDG
  - Main idea:
    - Construct PDG for each procedure
    - Connect PDGs with auxiliary dependence edges
Representing Procedure Calls in SDG

- **Calling side:**
  - **Call-site** nodes represent procedure call statements
  - Parameter passing is represented by **actual-in/actual-out** vertices

- **Called procedure side:**
  - Parameter passing is represented by **formal-in/formal-out** vertices
Representing Procedure Calls in SDG

- PDGs are connected using new types of edges to produce SDG:
  - **Call** edges:
    - Link call-site nodes with procedure entry nodes
  - **Parameter-in** edges:
    - Connect actual-in nodes with formal-in nodes
  - **Parameter-out** edges:
    - Connect formal-out nodes with actual-out nodes
  - **Summary** edges:
    - Represent transitive dependences that arise due to function calls
Example Representing Procedure Calls

```
main()
{
    int a, b;
    a = 0;
    b = 1;
    add(a, b);
}

void add(int a, int b){
    a = a + b;
    return;
}
```
SDG Construction Steps

- Parse source code --- one procedure at a time.
  - Construct the CFG for each procedure including main.
- Add actual and formal parameter nodes:
  - Connect using parameter-in, parameter-out edges
- Represent function calls
  - Using call edges
Steps in SDG Construction

- Find data dependencies:
  - Perform data flow analysis of the CFGs
  - Connect data dependent nodes using data dependence edges

- Perform control dependence analysis of the CFGs:
  - Connect using control dependence edges

- Add summary edges:
  - Based on transitive dependencies
void add(int a, int b) {
    a = a + b;
    return;
}

void inc(int z) {
    add(z, 1);
    return;
}

main() {
    int s, i;
    s = 0;
    i = 1;
    while (i < 10) {
        add(s, i);
        inc(i);
    }
    write(s);
}
SDG Example

```plaintext
entry main

s = 0

i = 1

while i < 10

write(s)

```

```
entry inc

izn = i
zi = z
z = zni
zout = z

```

```
entry add

ain = z
a = a + b
aout = a

```

```
add

ain = z
b = bni
a = a + b
aout = a

```

Control Dependence
Data Dependence
Call, Parameter-in,
Parameter-out
Summary Edge
Slicing a PDG

```c
1 main()
2 {
3   int i, sum;
4   sum = 0;
5   i = 1;
6   while(i <= 10) {
7       sum = sum + 1;
8       ++ i;
9   }
10  cout<< sum;
11  cout<< i;
12 }
```

\[ SC = <12, i> \]
Slicing PDG

- From the node representing the slice point:
  - Find all reachable nodes
  - Corresponding statements form the slice
Slicing a PDG

Control Dep. Edge

Data Dep. Edge

Slice Point
Slicing SDG

- Two phase algorithm:
  - Proposed by Horwitz et al.

- **Pass 1**: From the slice point:
  - Traverse backward along all edges except parameter-out edges
  - Mark the reached vertices

- **Pass 2**: From vertices marked in Pass 1
  - Traverse backwards along all edges:
  - Except call and parameter-in edges
Example

```c
main()
{
    int a, b, c;
    a = 0;
    b = 1;
    c = add(a, b);
}

int add(int a, int b)
{
    a = a + b;
    return a;
}
```
main()
{
    int a, b, c;
    a = 0;
    b = 1;
    c = add(a, b);
}

int add(int a, int b)
{
    a = a + b;
    return a;
}
main()
{
    int a, b, c;
    a = 0;
    b = 1;
    c = add(a, b);
}

int add(int a, int b)
{
    a = a + b;
    return a;
}
Complexity Analysis

- Time complexity of slicing:
  - Linear in size of SDG
  - Polynomial in the number of statements
Slicing: A Graph Reachability Problem

- Reachability analysis-based slicing algorithms:
  - Involve traversal along control and data dependence edges
- Both static and dynamic slices can be computed.
- Many algorithms have been proposed
  - Intra and Inter-procedural slicing
Inter-Procedural Slice Using SDG

Pass 1:
- The traversal starts from the desired vertex - goes backwards along all edges except parameter-out edges.

Pass 2:
- Starts from each vertex reached in pass one - goes backwards along all edges except call and parameter-in edges.

The slice is the reached in pass1 and pass2 set of vertices.

Fig 1: Example program and it’s SDG
Dynamic Slicing

- A dynamic slice restricts to the program behavior exhibited in a single program execution.
  - Many statements are not executed
  - Many dependencies may not manifest
- First dynamic slicing (Korel and Laski):
  - Simplistically extended results of static slicing:
    - Mark all model elements executed
      - From static slicing exclude nodes not executed
  - Imprecise, inefficient