Symbolic Evaluation/Execution

Computer Science 520-620
Spring 2013
Prof. L. J. Osterweil

Dynamic Analysis
• Dynamic analysis approaches are based on sampling the input space
• Can demonstrate the presence of failures
  — But not the absence of faults
• Support deep detailed examination of one execution
  — But say nothing about other executions
• Infer behavior or properties from executing a sample of test cases
• Reflect actual execution environment

Static Analysis
• Static analysis approaches based on a “global” assessment of system behavior
• Based on understanding the program (artifact) itself
  — Not executions of it
• Based upon an abstract representation of actual execution environment
• Can be used to prove the absence of faults
• For certain classes of faults
  — Seems good for robustness, security, privacy analysis
• Does not address functionality, though

Static Analysis Approaches
• Data Flow Analysis
• Concurrency Analysis
• Symbolic Evaluation
• Formal Verification

Symbolic Evaluation/Execution
• Addresses analysis of program functionality
  — And more
• Creates a functional representation of a path through a program
• For a path \( P_i \) define
  — \( D[P_i] \) to be the domain of path \( P_i \)
  — \( C[P_i] \) as the computation performed in executing \( P_i \)
Functional Representation of an Executable Component

$X$ is the domain of path $P_i$.
Denoted $D[P_i]$

$X = D[P_j] \cup \ldots \cup D[P_i] = D[P]$  \hspace{1cm} The $D[P]$ partition the input space

$D[P_i] \cap D[P_j] = \emptyset$, $i \neq j$

Representing Computation

- Symbolic names represent the input values
- The path value $PV$ of a variable for a path describes the value of that variable in terms of those symbolic names
- The computation of the path $C[P]$ is defined to be the path values of the outputs for the path

Representing Conditionals

- An interpreted branch condition or interpreted predicate is represented as an inequality or equality condition
- The path condition $PC$ describes the domain of the path.
  - It is the conjunction of the interpreted branch conditions
- The domain of the path $D[P]$ is the set of input values that satisfy the PC for the path

Example program

```
procedure Contrived is
  X, Y : integer;
begin
  if X - Y < 0 then
    Z := 0;
  else
    read X, Y;
    if Y > 0 then
      X, Y, Z := integer;
    end if
    if X - Y < 0 then
      Z := 0;
    else
      write Z;
    end if
  end if
end Contrived;
```

Note: "$x_i$" means the $i$th value taken as input to the program

Presenting the results

```
<table>
<thead>
<tr>
<th>Statements</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 X := In_1</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>2 Y := In_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Z := In_3</td>
<td>true</td>
<td>In_1 ≥ 3 ∧ In_2 ≥ 3</td>
</tr>
<tr>
<td>4 Z := 0;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 if Y &gt; 0 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 Y := Y + 1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 if X - Y &lt; 0 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 write Z;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 write Y;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 end if</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7,9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Results (feasible path)

$D[P] = \{ (In_1, In_2) \mid In_1 ≥ 3 ∧ In_2 ≥ 0 ∧ In_2 - In_1 ≥ 5 \}$

$C[P] = PV. Y = In_2 + 5$

$P = 1, 2, 3, 5, 6, 7, 9$

$D[P] = \{ (In_1, In_2) \mid (In_1 ≥ 3 ∧ In_2 ≥ 0 ∧ In_1 - In_2 ≥ 5) \}$

$C[P] = PV. Y = In_2 + 5$
Evaluating another path

procedure Contrived is
X, Y, Z : integer;
1 read X, Y;
2 if X ≥ 3 then
3 Z := X+Y;
else
4 Z := 0;
endif;
5 if Y > 0 then
6 Y := Y + 5;
endif;
7 if X - Y < 0 then
8 write Z;
else
9 write Y;
endif;
end Contrived;

Results (infeasible path)

What about loops?

• Symbolic evaluation requires a complete path description
  
  Example Paths
  • P = 1, 2, 3, 5
  • P = 1, 2, 3, 4, 2, 3, 5
  • P = 1, 2, 3, 4, 2, 3, 4, 2, 3, 5
  • Etc.

What This Capability Is Good For:

– Documentation
– Path Executability
– Fault Detection
– Path Selection
– Test Data Generation

Documentation

• Reduces path condition to a canonical form
• Simpler often determines consistency
  
  PC = ( In1 >= 5 ) \& ( In1 < 0 )
• May want to document path computation in simplified and unsimplified forms
  
  PV.X = In1 + (In1 +1) + (In1 + 2) + (In1 + 3) = 4 *In1 + 6
Path Executability

- strategy = solve a system of constraints
  - theorem prover
    - consistency
    - algebraic, e.g., linear programming
    - consistency and find solutions
  - solution is an example of automatically generated test data

In general it is not possible to solve an arbitrary system of constraints

- One of Hilbert’s Problems
- Some complications:
  - Non-linear constraints
    - $X^2 Y = 10$
    - $X^2 + 3X + Y = 0$
  - Integer constraints
  - Constraints involving non-numbers
    - Character strings
  - Can be dealt with separately (to some extent), but not all together
  - Equivalent to the Halting Problem

Fault Detection

- Implicit fault conditions
  - E.g. Subscript value out of bounds
  - E.g. Division by zero e.g., $Z = Y/0$
  - Create assertion to represent the fault and conjoin with the pc
    - Division by zero $assert(divisor \neq 0)$
    - Determine consistency $PC$ and $PV.divisor = 0$
    - if consistent then error possible
  - Must check the assertion at the point in the path where the construct occurs

Recall the Example program

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X ← $ln_2$</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y ← $ln_2$</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Z ← $ln_2 + ln_2$</td>
<td>true ∧ $ln_2 \geq 3 = ln_2 \geq 3$</td>
</tr>
<tr>
<td>5.6</td>
<td>Y ← $ln_2 \times 5$</td>
<td>$ln_2 \geq 3 ∧ ln_2 &gt; 0$</td>
</tr>
</tbody>
</table>
| 7.9  | Z ← $ln_2 / (ln_2 + 5 - 2)$ = $ln_2 / (ln_2 + 3)$ | $PC: (ln_2 \geq 3 ∧ ln_2 > 0 ∧ ln_2 - (ln_2 + 5) \leq 0)$
|      |              | = $(ln_2 \geq 3 ∧ ln_2 > 0 ∧ (ln_2-ln_2) > 5)$ |

Add in Diagnostic Constraint

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X ← $ln_2$</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y ← $ln_2$</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Z ← $ln_2 + ln_2$</td>
<td>true ∧ $ln_2 \geq 3 = ln_2 \geq 3$</td>
</tr>
<tr>
<td>5.6</td>
<td>Y ← $ln_2 \times 5$</td>
<td>$ln_2 \geq 3 ∧ ln_2 &gt; 0$</td>
</tr>
</tbody>
</table>
| 7.9  | Z ← $ln_2 / (ln_2 + 5 - 2)$ = $ln_2 / (ln_2 + 3)$ | $PC: (ln_2 \geq 3 ∧ ln_2 > 0 ∧ ln_2 - (ln_2 + 5) \leq 0)$
|      |              | = $(ln_2 \geq 3 ∧ ln_2 > 0 ∧ (ln_2-ln_2) > 5)$ |

Diagnostic constraint: $(ln_2+5-2) \leq 0$ or $ln_2 > 3$
Is Division by Zero possible?

- Path Constraint: \((l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5)\)
- Diagnostic Constraint: \(l_{2} = 3\)
- Total Constraint: \(l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5 \land l_{2} = 3\)

Results

Division by zero is impossible

Modified Example program

procedure Contrived is
  X, Y, Z : integer;
  1 read X, Y;
  2 if X \geq 3 then
  3   Z := X + Y;
  4 else
  5   Z := 0;
  6 endif;
  7 if Y > 0 then
  8   Y := Y + 5;
  9 endif;
  10 if X - Y < 0 then
  11   write Z;
  12 else
  13   Z := X / (Y - 8);
  14   write Z;
  15 endif;
  16 end Contrived;

Stmt PV PC
1 \(X \leftarrow l_{1}\) true
2,3 \(Z \leftarrow l_{1} \times l_{2}\) true \& \(l_{1} \geq 3 \land l_{2} \geq 3\)
5,6 \(Y \leftarrow l_{1} + 5\) \(l_{1} \geq 3 \land l_{2} \geq 0\)
7,9,10 \(Z \leftarrow \frac{l_{1}}{(l_{1} + 5 - 8)}\) = \(l_{1}/(l_{1} - 3)\)

PC: \((l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 0)\)
\(= (l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5)\)

Add in Different Constraint

procedure Contrived is
  X, Y, Z : integer;
  1 read X, Y;
  2 if X \geq 3 then
  3   Z := X + Y;
  4 else
  5   Z := 0;
  6 endif;
  7 if Y > 0 then
  8   Y := Y + 5;
  9 endif;
  10 if X - Y < 0 then
  11   write Z;
  12 else
  13   Z := X / (Y - 8);
  14   write Z;
  15 endif;
  16 end Contrived;

Stmt PV PC
1 \(X \leftarrow l_{1}\) true
2,3 \(Z \leftarrow l_{1} + l_{2}\) true \& \(l_{1} \geq 3 \land l_{2} = l_{1}\)
5,6 \(Y \leftarrow l_{1} + 5\) \(l_{1} \geq 3 \land l_{2} = 0\)
7,9,10 \(Z \leftarrow \frac{l_{1}}{(l_{1} + 5 - 8)}\) = \(l_{1}/(l_{1} - 3)\)

PC: \((l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 0)\)
\(= (l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5)\)

Diagnostic constraint: \((l_{1} + 5 - 8) = 0\) or \(l_{2} = 3\)

Is Division by Zero possible now?

- Path Constraint: \((l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5)\)
- Diagnostic Constraint: \(l_{2} = 3\)
- Total Constraint: \(l_{1} \geq 3 \land l_{2} > 0 \land (l_{1} - l_{2}) \geq 5 \land l_{2} = 3\)

Results

Division by zero is now possible—if the second input value is 3 and the first input value drives execution down this path
Fault Detection (continued)

• Checking user-defined assertions
  – Determine if the complement of the assertion is consistent with the PC
  – example
    • Assert \((A > B)\)
    • PC and \((PV.A \leq PV.B)\)
    • if consistent then assertion not valid

Recall the Example program

```
Stmt  PV      PC
1  X⟵ln1  true
   Y⟵ln2
2,3 Z⟵ln1+ln2  true ∧ ln1≥3 = ln2≥3
5,7 Y⟵ln1+5  ln1≥3 ∧ ln1≤0
7,9 (ln1≥3 ∧ ln1>0 ∧ ln1-(ln1+5)≤0) = (ln1≥3 ∧ ln1>0 ∧ (ln1-ln2)≤25)
```

Change the Example program program Path

```
Stmt  PV      PC
1  X⟵ln1  true
   Y⟵ln2
2,3 Z⟵ln1+ln2  true ∧ ln1≥3 = ln2≥3
5,7 Y⟵ln1+5  ln1≥3 ∧ ln1≤0
7,9 (ln1≥3 ∧ ln1>0 ∧ ln1-(ln1+5)≤0) = (ln1≥3 ∧ ln1>0 ∧ (ln1-ln2)≤25)
```

Example program With Assertion

```
Stmt  PV      PC
1  X⟵ln1  true
   Y⟵ln2
2,3 Z⟵ln1+ln2  true ∧ ln1≥3 = ln2≥3
5,7 Y⟵ln1+5  ln1≥3 ∧ ln1≤0
7,9 (ln1≥3 ∧ ln1>0 ∧ ln1-(ln1+5)≤0) = (ln1≥3 ∧ ln1>0 ∧ (ln1-ln2)≤25)
```

Combine Path Condition with Assertion

• Path Condition
  \(ln1≥3 ∧ ln1≤0\)

• Assertion
  \(ln1 < ln2\)

• Yields an inconsistency
• Assertion will always fail on this path
• A different assertion may fail sometimes or never
• The same assertion may behave differently on different paths

Comparing Dynamic and Symbolic Assertion Checking Approaches

• With run-time assertion checking, assertions are inserted as executable instructions and checked during execution
  – dependent on test data selected
    (dynamic testing)
• With symbolic evaluation, assertions checked when they occur on a path being evaluated
  – dependent on path, but not on the test data
  – look for violating data in the path domain
  – Assumes theoretical model of numeric computation
    • Risky assumption, especially for real numbers
### Additional Features:
- Simplification
- Path Condition Consistency
- Fault Detection
- Path Selection
- Test Data Generation

### Path Selection
- User selected
- Automated selection to satisfy some criteria
  - e.g., exercise all statements at least once
- Because of infeasible paths, best if path selection done incrementally

### Incremental Path Selection
- PC and PV maintained for partial path
- Inconsistent partial path can often be salvaged

### Path Selection (continued)
- Can be used in conjunction with other static analysis techniques to determine path feasibility
  - Testing criterion generates a path that needs to be tested
  - Symbolic evaluation determines if the path is feasible
    - Can eliminate some paths from consideration

### Additional Features:
- Simplification
- Path Condition Consistency
- Fault Detection
- Path Selection
- Test Data Generation
Test Data Generation

- Simple test data selection: Select test cases that satisfy the path condition \( p_c \)
- Error based test data selection
  - Try to select test cases that will help reveal faults
  - Use information about the path domain and path values to select test data
    - e.g., \( P.V. X = a \times (b + 2) \):
      - \( a = 1 \) combined with min and max values of \( b \)
      - \( b = -1 \) combined with min and max values for \( a \)
      - values that would force \( X \) to take on "special" values
        e.g., \( a = 0 \) or \( b = -2 \), if \( X = 0 \) is a special value

Some Problems

- Information explosion
- Impracticality of all paths
- Path condition consistency
- Aliasing
  - elements of a compound type
    e.g., arrays and records
  - pointers

Alias Problems

```
Do for I = 1 to 100; A(I) := I

A(2) := 5
read I; A(I)

X := A(2)
I > 2

Y := A(I)
Z := A(I)
```

What is \( X \)?
Indeterminate subscript

What is \( Y \)?
What is \( Z \)?

This is an essential capability for supporting Formal Verification

- Formal Verification can be used to prove that all possible executions of a program will always produce the correct computational results
  - Even if there is an infinite number of paths
  - "some restrictions apply"