Symbolic Evaluation/Execution

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
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**

\[ X_i \text{ is the domain of path } P_i \]

Denoted \( D[P_i] \)

\[ X = D[P_1] \cup \ldots \cup D[P_r] = D[P] \]

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

*The D[P_i] partition the input space*

**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, Z : integer;
  1 read X, Y;
  2 if X ≥ 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    write Y;
 14  endif;
 15 end Contrived;
```

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
</table>
| 1    | X← In₁ | true
|      | Y ← In₂ | |
| 2,3  | Z ← In₁+In₂ | true ∧ In₁≥3 = In₁ ≥3
| 5,6  | Y ← In₂+5 | In₁ ≥3 ∧ In₂>0
| 7,9  | (In₁≥3 ∧ In₂≥0 ∧ In₁ - (In₂+5 ≥0) = (In₁≥3 ∧ In₂≥0 ∧ (In₁-In₂)≥5)

Note: “Inᵢ” means the ith 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>procedure Contrived is</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X, Y, Z : integer;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read X, Y;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if X ≥ 3 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z := XY; else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X := 0; endif;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if Y &gt; 0 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y := Y + 5; endif;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if X - Y &lt; 0 then</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Z; else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write Y; endif</td>
<td></td>
<td></td>
</tr>
<tr>
<td>end Contrived</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
P = 1, 2, 3, 5, 6, 7, 9\]
\[
D[P] = \{(In_1, In_2) | In_1 ≥ 3 ∧ In_2 > 0 ∧ (In_1 - In_2) ≥ 5\}
\[
C[P] = PV.Y = In_2 + 5
\]
\[
(PV.Z = In_1 +, In_2)
\]

Results (feasible path)

\[
P = 1, 2, 3, 5, 6, 7, 9\]
\[
D[P] = \{(In_1, In_2) | 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;
  read X, Y;
  if X ≥ 3 then
    Z := X+Y;
  else
    Z := 0;
  endif;
  if Y > 0 then
    Y := Y + 5;
  endif;
  if X - Y < 0 then
    write Z;
  else
    write Y;
  endif;
end Contrived;

Stmts  PV  PC
1  X← ln₁  true
    Y← ln₂
2,3 Z ← ln₁+ ln₂  true ∧ ln₁≥3 = ln₁≥3
5,7 ln₁≥3 ∧ ln₂≤0
7,8 ln₁≥3 ∧ ln₂≤0 ∧ ln₁-ln₂<0

P = 1, 2, 3, 5, 7, 8
D[P] = { (ln₁, ln₂) | ln₁≥3 ∧ ln₂≤0 ∧ ln₁-ln₂<0}  (infeasible path!)
Results (infeasible path)

\[(\ln_1 - \ln_2) < 0\]

\[\ln_1 \geq 3\]

\[\ln_2 \leq 0\]

---

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

• Simplifier often determines consistency

\[ PC = (\ ln_1 >= 5) \land (\ ln_1 < 0) \]

• May want to document path computation in simplified and unsimplified forms

\[ PV.X = \ ln_1 + (\ ln_1 + 1) + (\ ln_1 + 2) + (\ ln_1 + 3) = 4 * \ ln_1 + 6 \]
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
    • $X \cdot Y = 10$
    • $X^3 - X^2 + 3X + Y^3 = 0$
  – 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., Q:=N/D

– Create assertion to represent the fault and conjoin with the pc
  • Division by zero assert(divisor ≠ 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

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;

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X←ln₁</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y ← ln₂</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>Z ← ln₁+ln₂</td>
<td>true ∧ ln₁≥3 = ln₁≥3</td>
</tr>
<tr>
<td>5,6</td>
<td>Y ← ln₂+5</td>
<td>ln₁≥3 ∧ ln₂&gt;0</td>
</tr>
<tr>
<td>7,9</td>
<td>(ln₁≥3 ∧ ln₂&gt;0 ∧ ln₁-(ln₂+5)≥0) = (ln₁≥3 ∧ ln₂&gt;0 ∧ (ln₁-ln₂)≥5)</td>
<td></td>
</tr>
</tbody>
</table>
### Modified Example program

```plaintext
procedure Contrived is
X, Y, Z : integer;
read X, Y;
if X ≥ 3 then
  Z := X+Y;
else
  Z := 0;
endif;
if Y > 0 then
  Y := Y + 5;
endif;
if X - Y < 0 then
  write Z;
else
  Z := X / (Y-2);
  write Z;
endif;
end Contrived;
```

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X ← \text{\textit{ln}$_1$}</td>
<td>\text{true}</td>
</tr>
<tr>
<td></td>
<td>Y ← \text{\textit{ln}$_2$}</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>Z ← \text{\textit{ln}$_1$}+\text{\textit{ln}$_2$}</td>
<td>\text{true} ∧ \text{\textit{ln}$_1\geq3 = ln$_1\geq3$}</td>
</tr>
<tr>
<td>5,6</td>
<td>Y ← \text{\textit{ln}$_2$}+5</td>
<td>\text{\textit{ln}$_2\geq3 ∧ ln$_2&gt;0$}</td>
</tr>
<tr>
<td>7,9,10</td>
<td>Z ← \text{\textit{ln}$_1$}/(\text{\textit{ln}$_2$}+5 - 2)</td>
<td>\text{\textit{ln}$_1$}/(\text{\textit{ln}$_2$}+3)</td>
</tr>
</tbody>
</table>

\[ PC: \text{\textit{ln}$_2\geq3 ∧ ln$_2>0 ∧ ln$_1-(ln$_2$+5)≥0} \]
\[ = (\text{\textit{ln}$_2\geq3 ∧ ln$_2>0 ∧ (ln$_1-ln$_2)≥5}) \]

Diagnos-c constraint: \text{\textit{ln}$_2$+5-2)=0 or \text{\textit{ln}$_2$=-3}
Is Division by Zero possible?

• Path Constraint $= (ln_1 \geq 3 \land ln_2 > 0 \land (ln_1 - ln_2) \geq 5)$
• Diagnostic Constraint $ln_2 = -3$
• Total Constraint:
  $ln_1 \geq 3 \land ln_2 > 0 \land (ln_1 - ln_2) \geq 5 \land ln_2 = -3$

Results

Division by zero is impossible
### Modified Example program

```plaintext
procedure Contrived is
  X, Y, Z : integer;
  read X, Y;
  if X ≥ 3 then
    Z := X+Y;
  else
    Z := 0;
  endif;
  if Y > 0 then
    Y := Y + 5;
  endif;
  if X - Y < 0 then
    write Z;
  else
    Z := X / (Y-8);
    write Z;
  endif;
end Contrived;
```

### Add in Different Constraint

```plaintext
procedure Contrived is
  X, Y, Z : integer;
  read X, Y;
  if X ≥ 3 then
    Z := X+Y;
  else
    Z := 0;
  endif;
  if Y > 0 then
    Y := Y + 5;
  endif;
  if X - Y < 0 then
    write Z;
  else
    Z := X / (Y-8);
    write Z;
  endif;
end Contrived;
```

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X←ln₁</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y ← ln₂</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>Z ← ln₁+ln₂</td>
<td>true ∧ ln₁≥3 = ln₁≥3</td>
</tr>
<tr>
<td>5,6</td>
<td>Y ← ln₂+5</td>
<td>ln₁≥3 ∧ ln₂&gt;0</td>
</tr>
<tr>
<td>7,9,10</td>
<td>Z ← ln₁ / (ln₂+5 - 8) = ln₁ / (ln₂-3)</td>
<td>PC: (ln₁≥3 ∧ ln₂&gt;0 ∧ ln₁-(ln₂+5)≥0) = (ln₁≥3 ∧ ln₂&gt;0 ∧ (ln₁-ln₂)≥5)</td>
</tr>
</tbody>
</table>

**Diagnostic constraint:** (ln₂+5-8)=0 or ln₂=3
Is Division by Zero possible now?

- Path Constraint = \((I_{n_1} \geq 3 \land I_{n_2} > 0 \land (I_{n_1} - I_{n_2}) \geq 5)\)
- Diagnostic Constraint \(I_{n_2} = 3\)
- Total Constraint:
  \(I_{n_1} \geq 3 \land I_{n_2} > 0 \land (I_{n_1} - I_{n_2}) \geq 5 \land I_{n_2} = 3\)

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

```
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    write Y;
  14  endif;
  15  end Contrived;
```

<table>
<thead>
<tr>
<th>Stmt</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X \leftarrow \text{ln}_1</td>
<td>true</td>
</tr>
<tr>
<td></td>
<td>Y \leftarrow \text{ln}_2</td>
<td></td>
</tr>
<tr>
<td>2,3</td>
<td>Z \leftarrow \text{ln}_1 + \text{ln}_2</td>
<td>true \land \text{ln}_1 \geq 3 = \text{ln}_1 \geq 3</td>
</tr>
<tr>
<td>5,6</td>
<td>Y \leftarrow \text{ln}_2 + 5</td>
<td>\text{ln}_2 \geq 3 \land \text{ln}_2 &gt; 0</td>
</tr>
<tr>
<td>7,9</td>
<td></td>
<td>\begin{align*} \text{ln}_1 \geq 3 \land \text{ln}_2 &gt; 0 \land \text{ln}_1 - \text{ln}_2 + 5 \geq 0 \ = \text{ln}_1 \geq 3 \land \text{ln}_2 &gt; 0 \land (\text{ln}_1 - \text{ln}_2) \geq 5 \end{align*}</td>
</tr>
</tbody>
</table>
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;

assert X < Y

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

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;

assert ln1 < ln2

Stmt  PV                  PC
1    X ← ln1              true
     Y ← ln2
2,3  Z ← ln1 + ln2        true ∧ ln1 ≥ 3 = ln1 ≥ 3
5,7  Y ← ln2 + 5          ln2 ≥ 3 ∧ ln2 ≤ 0
7,9  (ln1 ≥ 3 ∧ ln2 > 0 ∧ (ln1 - (ln2 + 5) ≥ 0)
     = (ln1 ≥ 3 ∧ ln2 > 0 ∧ (ln1 - ln2) ≥ 5)
Combine Path Condition with Assertion

- Path Condition
  \[ In_1 \geq 3 \land In_2 \leq 0 \]
- Assertion
  \[ In_1 < In_2 \]
- 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

\[ P_c'' = p_c' \text{ and } (x > 3) \]
\[ = p_c \text{ and } (x \leq 0) \text{ and } (x > 3) \]

INCONSISTENT!
infeasible path

\[ P_c''' = p_c' \text{ and } (x \leq 3) \]
\[ = p_c \text{ and } (x \leq 0) \text{ and } (x \leq 3) \]

CONSISTENT [if \( p_c' \) is consistent]
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 \( pc \)
• 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., \( PV.X = a \times (b + 2) \);
      – \( a = 1 \) 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?  What is Y?  What is Z?

Indeterminate subscript

constraints on subscript value due to path condition add more uncertainty

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”