Introduction to Dynamic Analysis
Reference material

• Introduction to dynamic analysis
Common Definitions

- **Failure** -- result that deviates from the expected or specified intent
- **Fault/defect** -- a flaw that could cause a failure
- **Error** -- erroneous belief that might have led to a flaw that could result in a failure
- **Static Analysis** -- the static examination of a product or a representation of the product for the purpose of inferring properties or characteristics
- **Dynamic Analysis** -- the execution of a product or representation of a product for the purpose of inferring properties or characteristics
- **Testing** -- the (systematic) selection and subsequent "execution" of sample inputs from a product's input space in order to infer information about the product's behavior.
  - usually trying to uncover failures
  - the most common form of dynamic analysis
- **Debugging** -- the search for the cause of a failure and subsequent repair
Validation and Verification: V&V

- **Validation** -- techniques for assessing the quality of a software product
- **Verification** -- the use of analytic inference to (formally) prove that a product is consistent with a specification of its intent
  - the specification could be a selected property of interest or it could be a specification of all expected behaviors and qualities
    - e.g., all deposit transactions for an individual will be completed before any withdrawal transaction will be initiated
  - a form of validation
  - usually achieved via some form of static analysis
Correctness

- A product is correct if it satisfies all the requirement specifications
  - Correctness is a mathematical property
  - Requires a specification of intent
  - Specifications are rarely complete
  - Difficult to prove poorly-quantified qualities such as user-friendly

- A product is behaviorally or functionally correct if it satisfies all the specified behavioral requirements
Reliability

• measures the dependability of a product
  • the probability that a product will perform as expected
  • sometimes stated as a property of time e.g., mean time to failure

• Reliability vs. Correctness
  • reliability is relative, while correctness is absolute (but only wrt a specification)
  • given a "correct" specification, a correct product is reliable, but not necessarily vice versa
Robustness

• behaves "reasonably" even in circumstances that were not expected
  • making a system robust more than doubles development costs
  • a system that is correct may not be robust, and vice versa
Approaches

• Dynamic Analysis
  • Assertions
  • Error seeding, mutation testing
  • Coverage criteria
  • Fault-based testing
  • Specification-based testing
  • Object oriented testing
  • Regression testing

• Static Analysis
  • Inspections
  • Software metrics
  • Symbolic execution
  • Dependence Analysis
  • Data flow analysis
  • Software Verification
Types of Testing--what is tested

- **Unit testing** - exercise a single simple component
  - Procedure
  - Class
- **Integration testing** - exercise a collection of inter-dependent components
  - Focus on interfaces between components
- **System testing** - exercise a complete, stand-alone system
- **Acceptance testing** - customer's evaluation of a system
  - Usually a form of system testing
- **Regression testing** - exercise a changed system
  - Focus on modifications or their impact
Test planning
Approaches to testing

- Black Box/Functional/Requirements based

- White Box/Structural/Implementation based
White box testing process

- **test data selection criteria**
  - test cases
  - executable component (obj code)
  - execution results

- **evaluation**
  - test data selection criteria

- **executable component (textual rep)**

- **Requirements or specifications**

- **oracle**
  - testing report
Black box testing process

- Test data selection
  - Criteria
    - Evaluation
      - Evaluation criteria
      - Test cases

- Executable component (textual rep)
  - Executable component (obj code)
    - Execution results
      - Oracle
        - Testing report

- Requirements or specifications
Why black AND white box?

- **Black box**
  - May not have access to the source code
  - Often do not care how s/w is implemented, only how it performs

- **White box**
  - Want to take advantage of all the information
  - Looking inside indicates structure => helps determine weaknesses
Paths

• Paths:
  -1, 2, 4, 5, 7
  -1, 2, 4, 6, 7
  -1, 3, 4, 5, 7
  -1, 3, 4, 6, 7
Paths can be identified by predicate outcomes

- $X > 0$
- $Z := 1$
- $Z := 5$
- $X * Y > 0$
- $Z := Z + 10$
- $Z := Z + 20$
- $X := Y + Z$

Outcomes:
- $-t, t$
- $-t, f$
- $-f, t$
- $-f, f$
Paths can be identified by domains

- \( \{ X, Y \mid X > 0 \text{ and } X \cdot Y > 0 \} \)
- \( \{ X, Y \mid X > 0 \text{ and } X \cdot Y \leq 0 \} \)
- \( \{ X, Y \mid X \leq 0 \text{ and } X \cdot Y > 0 \} \)
- \( \{ X, Y \mid X \leq 0 \text{ and } X \cdot Y \leq 0 \} \)

\[
X := Y + Z \\
Z := 1 \\
Z := 5 \\
X \cdot Y > 0 \\
Z := Z + 10 \\
Z := Z + 20
\]
Example with an infeasible path

1. X > 0

2. Y := X / 2

3. Y := 5

4. X * Y > 0

5. Z := 10

6. Z := 20

7. X := Y + Z
Example with an infeasible path

1. \( X > 0 \)
2. \( X > 0 \):
   - \( Y := X / 2 \)
   - \( X > 0, Y > 0 \)
3. \( X \leq 0 \):
   - \( Y := 5 \)
   - \( X \leq 0, Y = 5 \)
4. \( X * Y > 0 \)
5. \( Z := 10 \)
6. \( Z := 20 \)
7. \( X := Y + Z \)
Example Paths

- Feasible path: 1, 2, 4, 5, 7
- Infeasible path: 1, 3, 4, 5, 7
- Determining if a path is feasible or not requires additional semantic information
  - In general, unsolveable
  - In practice, intractable
Another example of an infeasible path

For $i := 1$ to 5 do
  $x(i) := x(i+1) + 1$;
end for:

Note, implicit instructions are explicitly represented.
Infeasible paths vs. unreachable code and dead code

unreachable code
X := X + 1;
Goto loop;
Y := Y + 5;

dead code
X := X + 1;
X := 7;
X := X + Y;

Never executed

‘Executed’, but irrelevant
Test Selection Criteria

• How do we determine what are good test cases?
• How do we know when to stop testing?

Test Adequacy
Test Selection Criteria

• A test set $T$ is a finite set of inputs (test cases) to an executable component.
• Let $D(S)$ be the domain of execution for program/component/system $S$.
• Let $S(T)$ be the results of executing $S$ on $T$.
• A test selection criterion $C(T,S)$ is a predicate that specifies whether a test set $T$ satisfies some selection criterion for an executable component $S$.
• Thus, the test set $T$ that satisfies the Criterion $C$ is defined as:

$\{ t \in T \mid T \subseteq D(S) \text{ and } C(T, S) \}$
Ideal Test Criterion

• A test criterion is ideal if for any executable system $S$ and every $T \subseteq D(S)$ such that $C(T, S)$, if $S(T)$ is correct, then $S$ is correct.

  • of course we want $T << D(S)$

  • In general, $T = D(S)$ is the only test criterion that satisfies ideal
In general, there is no ideal test criterion

“Testing shows the presence, not the absence of bugs”
E. Dijkstra

• Dijkstra was arguing that verification was better than testing
• But verification has similar problems
  • can't prove an arbitrary program is correct
    • can't solve the halting problem
  • can't determine if the specification is complete
• Need to use dynamic and static techniques that compliment each another
Effectiveness a more reasonable goal

- A test criterion $C$ is *effective* if for any executable system $S$ and every $T \subseteq D(S)$ such that $C(T, S)$,
  - if $S(T)$ is correct, then $S$ is highly reliable
  OR
  - if $S(T)$ is correct, then $S$ is guaranteed (or is highly likely) not to contain any faults of a particular type

- Currently can not do either of these very well
  - Some techniques (static and dynamic) can provide some guarantees
Two Uses for Testing Criteria

• **Stopping rule**—when has a system been tested enough

• **Test data evaluation rule**—evaluates the quality of the selected test data
  
  • May use more than one criterion
  • May use different criteria for different types of testing
    • regression testing versus acceptance testing
Black Box/Functional Test Data Selection

• Typical cases
• Boundary conditions/values
• Exceptional conditions
• Illegal conditions (if robust)
• Fault-revealing cases
  • based on intuition about what is likely to break the system
• Other special cases
Functional Test Data Selection

• Stress testing
  • large amounts of data
  • worse case operating conditions

• Performance testing

• Combinations of events
  • select those cases that appear to be more error-prone
  • Select 1 way, 2 way, … n way combinations
Sequences of events

• Common representations for selecting sequences of events
  • Decision tables
  • Usage scenarios
## Decision Table

<table>
<thead>
<tr>
<th>events</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>-</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e2</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>e3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>e4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
Usage Scenarios

Graphical Usage Model of a Simple System

Invocation → Main Menu → Termination

Display
Overview of Dynamic Analysis Techniques

• Testing Processes
  • Unit, Integration, System, Acceptance, Regression, Stress

• Testing Approaches
  • Black Box versus White Box

• Black Box Strategies
  • Test case selection criteria
  • Representations for considering combinations of events/states
White Box/Structural Test Data Selection

• Coverage based
• Fault-based
  • e.g., mutation testing, RELAY
• Failure-based
  • domain and computation based
  • use representations created by symbolic execution
Coverage Criteria

• control-flow adequacy criteria

• $G = (N, E, s, f)$ where
  • the nodes $N$ represent executable instructions (statement or statement fragment)
  • the edges $E$ represent the potential transfer of control
  • $s \in N$ is a designated start node
  • $f \in N$ is a designated final node
  • $E = \{ (n_i, n_j) |$ syntactically, the execution of $n_j$ follows the execution of $n_i\}$
Control-Flow-Graph-Based Coverage Criteria

- Statement Coverage
- Branch Coverage
- Path Coverage
- Hidden Paths
- Loop Guidelines
  - General
  - Boundary – Interior
Statement Coverage

- requires that each statement in a program be executed at least once
- formally:
  - a set $P$ of paths in the CFG satisfies the statement coverage criterion iff for each $n_i \in N$, $\exists p \in P$ such that $n_i$ is on path $p$
  - defined in terms of paths
Statement Coverage

- only about 1/3 of NASA statements were executed before software was released (Stucki 1973)
- usually can achieve 85% coverage easily, but why not 100%?
  - unreachable code
  - complex sequence (should be tested!)
- Microsoft reports 80-90% code coverage
How does OO affect coverage?

• Often only parts of a reused component are actually executed by a system
  • Would expect good coverage for unit testing
  • More restricted coverage for integration testing
**Coincidental Correctness**

- Executing a statement does not guarantee that a fault on that path will be revealed

- **Example:**
  
  \[
  Y := X \times 2
  \]

  \[
  Y := X \times \times 2
  \]

  If \( x = 2 \) then the fault is not exposed
Branch Coverage

- Requires that each branch in a program (each edge in a control flow graph) be executed at least once
  - e.g., Each predicate must evaluate to each of its possible outcomes
- Branch coverage is stronger than statement coverage
Branch Coverage

Statement Coverage: PATH 1, 2, 3

Branch Coverage: PATH 1, 2, 1, 2, 3
Hidden Path (branch) Coverage

- Requires that each condition in a compound predicate be tested

Example:

\(( X > 1 ) \lor ( Y < 2 )\)

Test Data:

- \(X = 2, Y = 5 \rightarrow T\)
- \(X = 1, Y = 5 \rightarrow F\)

but, true branch is never tested for data where \(Y < 2\).

\[
\begin{array}{c|c|c}
X & Y < 2 & \text{Result} \\
\hline
T & F & T \\
T & T & T \\
F & T & F \\
F & F & F \\
\end{array}
\]
Path Coverage

• Requires that every executable path in the program be executed at least once

• In most programs, path coverage is impossible
  • Example:
    
    read N;
    SUM := 0;
    for I = 1 to N do
      read X;
      SUM := SUM + X;
    endfor

• How do we choose a set of paths?
Loop Coverage

• Path 1, 2, 1, 2, 3 executes all branches (and all statements) but does not execute the loop well.
Typical Guidelines for loop coverage

- fall through case
- minimum number of iterations
- minimum +1 number of iterations
- maximum number of iterations

1, 3
1,2,3
1,2,1,2,3
$(1, 2,)^n$ 3
Boundary - Interior Criteria

• **boundary test** of a loop causes the loop to be entered but not iterated

• **interior test** of a loop causes a loop to be entered and then iterated at least once

• both boundary and interior tests are to be selected for each unique path through the loop
Example
Paths for Example

Boundary paths
1,2,3,5,7          a
1,2,3,6,7          b
1,2,4,5,7          c
1,2,4,6,7          d

Interior paths
(for 2 executions of the loop)
    a,a
    a,b
    a,c
    a,d
    b,a
    b,b
    ...
    x,y for x,y = a, b, c, d
Selecting paths that satisfy these criteria

• static selection
  • some of the associated paths may be infeasible

• dynamic selection
  • monitors coverage and displays areas that have not been satisfactorily covered
Problem with coverage criteria:

• Fault detection may depend upon
  • Specific combinations of statements, not just coverage of those statements
  • Astutely selected test data that reveals the fault, not just test data that executes the statement/branch/path
• Will look at semantically richer models
• First look at some axioms about testing criteria
Example program (symbolic evaluation)

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;
end Contrived;

Stmt    PV        PC
  1      X← x     true
  2,3    Z ← x+y  true ∧ x≥3 = x≥3
  5,6    Y ← y+5  x≥3 ∧ y>0
  7,9      x≥3 ∧ y>0 ∧ x-(y+5)≥0
            = x≥3 ∧ y>0 ∧ (x-y)≥5
Presenting the results

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

P = 1, 2, 3, 5, 6, 7, 9
D[P] = \{ (x,y) | x≥3 ∧ y>0 ∧ x-y≥5\}
C[P] = PV.Y = y +5
Results (feasible path)

\[ x \geq 3 \quad \land \quad y > 0 \quad \land \quad x - y \geq 5 \]

\[ (x-y) \geq 5 \]

\[ P = 1, 2, 3, 5, 6, 7, 9 \]

\[ D[P] = \{ (x, y) | x \geq 3 \land y > 0 \land x - y \geq 5 \} \]

\[ C[P] = PV.Y = y + 5 \]
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;

<table>
<thead>
<tr>
<th>Stmts</th>
<th>PV</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X← x</td>
<td>true</td>
</tr>
<tr>
<td>2,3</td>
<td>Z ←  x+y</td>
<td>true ∧ x≥3 = x≥3</td>
</tr>
<tr>
<td>5,7</td>
<td>x≥3 ∧ y≤0</td>
<td></td>
</tr>
<tr>
<td>7,8</td>
<td>x≥3 ∧ y≤0 ∧ x-y &lt; 0</td>
<td></td>
</tr>
</tbody>
</table>
procedure EXAMPLE 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 EXAMPLE

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

P = 1, 2, 3, 5, 7, 8
D[P] = { (x, y) | x ≥ 3 ∧ y ≤ 0 ∧ x - y < 0}

infeasible path!
Results (infeasible path)

\[ y \leq 0 \]
\[ x \geq 3 \]
\[ (x-y) < 0 \]