Classification according to underlying testing approach

- **Structural testing**
  - Coverage of a particular set of elements in the structure of the program

- **Fault-based testing**
  - Some measurement of the fault detecting ability of test sets

- **Error-based testing**
  - Check on some error-prone points

---

**Structural Testing**

- **Program-based structural testing**
  - Control-flow based adequacy criteria
    - Statement coverage
    - Branch coverage
    - Path coverage
      - Length-i path coverage
    - Multiple condition coverage
      - All possible combinations of truth values of predicates
  - Data-flow based adequacy criteria
Structural Testing

- Data-flow based adequacy criteria
  - All definitions criterion
    - Each definition to some reachable use
  - All uses criterion
    - Definition to each reachable use
  - All def-use criterion
    - Each definition to each reachable use

Fault-based Adequacy

- Error seeding
  - Introducing artificial faults to estimate the actual number of faults
- Program mutation testing
  - Distinguishing between original and mutants
    - Competent programmer assumption
      - Mutants are close to the program
    - Coupling effect assumption
      - Simple and complex errors are coupled
Test Oracles

- Discussion
  - Automation of oracle necessary
  - Expected behavior given
  - Necessary parts of an oracle

Test Oracle

- A test oracle determines whether a system behaves correctly for test execution

- Webster Dictionary - Oracle
  - a person giving wise or authoritative decisions or opinions
  - an authoritative or wise expression or answer
Purpose of Test Oracle

- Sequential Systems
  - Check functionality
- Reactive (event-driven) Systems
  - Check functionality
  - Timing
  - Safety

Reactive Systems

- Complete specification requires use of multiple computational paradigms
- Oracles must judge all behavioral aspects in comparison with all system specifications and requirements
- Hence oracles may be developed directly from formal specifications
**Parts of an Oracle**

- **Oracle information**
  - Specifies what constitutes correct behavior
    - Examples: input/output pairs, embedded assertions
- **Oracle procedure**
  - Verifies the test execution results with respect to the oracle information
    - Examples: equality
- **Test monitor**
  - Captures the execution information from the run-time environment
    - Examples
      - Simple systems: directly from output
      - Reactive systems: events, timing information, stimuli, and responses

**Regression Testing**

- Developed first version of software
- Adequately tested the first version
- Modified the software; version 2 now needs to be tested
- How to test version 2?
- Approaches
  - Retest entire software from scratch
  - Only test the changed parts, ignoring unchanged parts since they have already been tested
  - Could modifications have adversely affected unchanged parts of the software?
Regression Testing

- “Software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program.”

Regression Testing vs. Development Testing

- During regression testing, an established test set may be available for reuse

  - Approaches
    - Retest all
    - Selective retest (selective regression testing) ← Main focus of research
Formal Definition

- Given a program $P$,
- its modified version $P'$, and
- a test set $T$
  - used previously to test $P$
- find a way, making use of $T$ to gain sufficient confidence in the correctness of $P'$

Regression Testing Steps

1. Identify the modifications that were made to $P$
   - Either assume availability of a list of modifications, or
   - Mapping of code segments of $P$ to their corresponding segments in $P'$

2. Select $T' \subseteq T$, the set of tests to re-execute on $P'$
   - May need results of step 1 above
   - May need test history information, i.e., the input, output, and execution history for each test
Regression Testing Steps

3. Retest P' with T'
   - Use expected output of P, if same

4. Create new tests for P', if needed
   - Examine whether coverage criterion is achieved

5. Create T''
   - The new test suite, consisting of tests from steps 2 and 4, and old tests that were not selected

Selective Retesting

Tests to rerun Tests not to rerun

- Tests to rerun
  - Select those tests that will produce different output when run on P'
    - Modification-revealing test cases
    - It is impossible to always find the set of modification-revealing test cases - (we cannot predict when P will halt for a test)
  - Select modification-traversing test cases
    - If it executes a new or modified statement in P' or misses a statement in P' that it executed in P
Procedure avg

S1.  count = 0
S2.  fread(fileptr,n)
P3.  while (not EOF) do
P4.    if (n<0)
S5.      return(error)
else
S6.        numarray[count] = n
S7.        count++
endif
S8.    fread(fileptr,n)
endwhile
S9.  avg = calcavg(numarray,count)
S10. return(avg)

Fig. 1. Procedure avg and its CFG.

<table>
<thead>
<tr>
<th>Test</th>
<th>Type</th>
<th>Output</th>
<th>Edges Traversed</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>Empty File</td>
<td>0</td>
<td>(entry, D), (D, S1), (S1, S2) (S2, P3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(P3, S9), (S9, S10), (S10, exit)</td>
</tr>
<tr>
<td>t2</td>
<td>-1</td>
<td>Error</td>
<td>(entry, D) (D, S1), (S1, S2), (S2, P3),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(P3, P4), (P4, S5), (S5, exit)</td>
</tr>
<tr>
<td>t3</td>
<td>1 2 3</td>
<td>2</td>
<td>(entry, D) (D, S1), (S1, S2), (S2, P3), (P3, P4),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(P4, S6), (S6, S7), (S7, S8), (S8, P3),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(P3, S9), (S9, S10), (S10, exit)</td>
</tr>
</tbody>
</table>
### Test History

<table>
<thead>
<tr>
<th>Edge</th>
<th>TestsOnEdge(edge)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(entry, D)</td>
<td>111</td>
</tr>
<tr>
<td>(D, S1)</td>
<td>111</td>
</tr>
<tr>
<td>(S1, S2)</td>
<td>111</td>
</tr>
<tr>
<td>(S2, P3)</td>
<td>111</td>
</tr>
<tr>
<td>(P3, P4)</td>
<td>011</td>
</tr>
<tr>
<td>(P3, S9)</td>
<td>101</td>
</tr>
<tr>
<td>(P4, S5)</td>
<td>010</td>
</tr>
<tr>
<td>(P4, S6)</td>
<td>001</td>
</tr>
<tr>
<td>(S5, exit)</td>
<td>010</td>
</tr>
<tr>
<td>(S6, S7)</td>
<td>001</td>
</tr>
<tr>
<td>(S7, S8)</td>
<td>001</td>
</tr>
<tr>
<td>(S8, P3)</td>
<td>001</td>
</tr>
<tr>
<td>(S9, S10)</td>
<td>101</td>
</tr>
<tr>
<td>(S10, exit)</td>
<td>101</td>
</tr>
</tbody>
</table>

---

**Procedure avg2**

S1'. count = 0

S2'. fread(fileptr,n)

P3'. while (not EOF) do

P4'. if (n<0)

* S5'.      print("bad input")

S5'.      return(error)

else

* S6'.      numarray[count] = n

endif

S8'. fread(fileptr,n)

endwhile

S9'. avg = calavg(numarray,count)

S10'. return(avg)

---

Fig. 3. Procedure avg2 and its CFG.
**Procedure avg**

S1. count = 0
S2. fread(fileptr,n)
P3. while (not EOF) do
P4.   if (n<0)
S5.     return(error)
else
S6.     numarray[count] = n
S7.     count++
endif
S8.   fread(fileptr,n)
endwhile
S9. avg = calcavg(numarray,count)
S10. return(avg)

**Procedure avg2**

S1’. count = 0
S2’. fread(fileptr,n)
P3’. while (not EOF) do
P4’.   if (n<0)
S5a.   print("bad input")
S5’.   return(error)
else
S6’.   numarray[count] = n
S7’.   count++
endif
S8’.   fread(fileptr,n)
endwhile
S9’. avg = calcavg(numarray,count)
S10’. return(avg)

\[ T' = \{ t2, t3 \} \]
Cost of Regression Testing

Cost = $C_x$

\[ \text{Analysis} \]
\[ \text{Selective Retest} \]

Cost = $C_y$

Retest All

We want $C_x < C_y$

Key is the test selection algorithm/technique

We want to maintain the same "quality of testing"

Factors to consider

- Testing costs
- Fault-detection ability
- Test suite size vs. fault-detection ability
- Specific situations where one technique is superior to another
Data-flow Testing

1. read(x, y)
2. x := x + 2;
3. y := 2;
4. y := y + 2;
5. x := x + 2;
6. x := y + 2;
7. x := x + y + 2;
8. x := y + 2;

All Definitions Criterion

- A set P of execution paths satisfies the all-definitions criterion iff
  - for all definition occurrences of a variable x such that
    - there is a use of x, which is feasibly reachable from that definition,
    - there is at least one path p in P such that
      - p includes a subpath through which the definition of x reaches some use occurrence of x

1. read(x, y)
2. x := x + 2;
3. y := 2;
4. y := y + 2;
5. x := x + 2;
6. x := y + 2;
7. x := y + 2;
8. x := x + y + 2;
A set $P$ of execution paths satisfies the all-uses criterion iff
- for all definition occurrences of a variable $x$ and all use occurrences of $x$,
  - that the definition feasibly reaches,
- there is at least one path $p$ in $P$ such that
  - $p$ includes a subpath through which that definition reaches the use.
All DU-paths criterion

- A set $P$ of execution paths satisfies the all-DU paths criterion iff
  - for all definitions of a variable $x$ and all paths $q$ through which that definition reaches a use of $x$,
  - there is at least one path $p$ in $P$ such that
    - $q$ is a subpath of $p$ and $q$ is cycle-free