Chapter 8 Outline

1. Instrumentation for Graph and Logical Expression Criteria
2. Building Mutation Testing Tools
Categories of Test Tools

- Research
- Private non-commercial
- Commercial

Test management
Simple coverage
Metrics
Code independent

Moderate techniques
Acceptors
Mostly complete
Some code-based
Often special purpose

Advanced techniques
Generators
Very powerful
Often code-based
Usually incomplete
General purpose

Types of Test Tools

1. Coverage analyzers (Acceptors)
   - What coverage? (most common is node)
   - What level? (sometimes object code)
2. Flow graph generators
3. Metrics
4. Instrumentation support
5. Test execution automation
   - Runs scripts and reports results
   - JUnit, HttpUnit, ...
6. Capture / replay
   - Useful for GUIs
7. Stubs and Drivers

8. Test data generators
   - Graph path – based
   - Data flow
   - Logic based
   - Functional
   - Mutation
   - Random
   - Only random test data generators exist
The key technology behind many test tools is “instrumentation”

- Coverage analysis is measured with instrumentation
- **Instrument**: One or more statements inserted into the program to monitor some aspect of the program
  - Must not affect the behavior
  - May affect timing
  - Source level or object code level

```java
public int min (int A, B)
{
    int m = A;
    if (A > B)
    {
        m = B;
    }
    return (m);
}
```

Mark: “if body is reached”
Instrumenting for Statement Coverage

1. Each node is given a unique id #
   - Node # or statement #

2. Create an array indexed by id #s – nodeCover [ ]

3. Insert an instrument at each node
   - nodeCover [ i ] ++;

4. Save nodeCover [ ] after each execution
   - Must accumulate results across multiple test cases

---

Statement Coverage Example

```
int nodeCover[] = {0,0,0,0,0,0,0,0,0,0}
```

- Read (nodeCover [i])
  - nodeCover [1] ++

After running a sequence of tests, any node for which nodeCover[node]==0 has not been covered.
**Edge Coverage Instrumentation**

```
 int edgeCover[] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0}

 for each edge e, put edgeCover[e]++ on the edge.

 If edgeCover[e] == 0, e has not been covered.
```

Note that the arrays could be boolean

**CACC Coverage Instrumentation**

```
CACC_Mark_1 (A, B)
{
 if (!A)
   // Don't allow short circuit
   if (B)
     CACCover[1]++;

 else if (A)
   {
     if (B)
       CACCover[2]++;
     else
       CACCover[3]++;
   
}

CACC_Mark_2 (C, D, E)
{

    if (C && (D || E))

        {  

            // Add code here to mark edge

        }

}
```

<table>
<thead>
<tr>
<th>A &amp; B</th>
<th>CACCover[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A &amp; B</th>
<th>CACCover[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

CACC_Mark_2 (C, D, E)
```
CACC Coverage Instrumentation (2)

def (x, y)
defCover[y] = 2
defCover[x] = 1
defCover[y] = 1

def (y)

def (x, y)

defCover[y] = 2

use (x)
use (x)

useCover[5, x, defCover[x]]++

useCover[4, x, defCover[x]]++

use (y)

useCover[6, y, defCover[y]]++

useCover[5, x, defCover[x]]++

For each variable, keep track of its current def location.

At each use, increment a counter for the def that reached it.

All-Uses Coverage Instrumentation
Instrumentation Summary

- Instrumentation can be added in multiple copies of the program
  - Source code
  - Java byte code (or other intermediate code)
  - Executable
- Instrumentation must not change or delete functionality
  - Only add new functionality
- Instrumentation may affect timing behavior
- Requires the program to be parsed
  - Once parsed, inserting instruments is straightforward
- Most challenging part is the user interface
  - Present reports as dumps of instrument arrays?
  - Present tables?
  - Program source with colored annotations?
  - Graph of program with colored annotations?