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**
  - 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 $P$

- **Tests not to rerun**
Fig. 1. Procedure avg and its CFG.

70. return(avg)
79. avg = calculate(numarray, count)
9. endwhile
8. fead(footer)
endif
7. count++
6. numarray[count] = \( u \)
else
5. return(error)
4. if \( (u > 0) \)
3. while (not EOF) do
2. fead(footer)
0. count = 0

Procedure avg
<table>
<thead>
<tr>
<th>Entity, D</th>
<th>Entity, D</th>
<th>D</th>
<th>Excl</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3, S9, S10</td>
<td>p4, S9, S10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p4, S6, S7</td>
<td>p4, S7, S8</td>
<td>1 2 3</td>
<td></td>
</tr>
<tr>
<td>p4, S6, S7</td>
<td>p4, S7, S8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>p3, S2, S3</td>
<td>p3, S3, S2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>p3, S2, S3</td>
<td>p3, S3, S2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Edges Traversed</th>
<th>Output</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty File</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1.** Test Information and Test History for Procedure avg.
Figure 3: Procedure avg2 and its CGF:

```plaintext
S10. return(avg)
S9. avg = calclavg(numarray, count)
endwhile
S8. if (iftrue, n)
end

-------------------------------
S6. numarray[count] = n
else
S5. return(error)

S4. print("bad input")
if (n>0)
P3. while (not EOF) do
S2. ifread(iftrue, n)
S1. count = 0
Procedure avg2
```
procedure avg

210. return(avg)
29. avg = calcavg(numerary, count)
endwhile

88. fread(fichier, n)
endf

------------------------------
6. numerary[count] = n
else
55. return(error)
print("bad input")

44. if (n>0)
3. while (not eof) do
22. fread(fichier, n)
11. count = 0
procedure avg

210. return(avg)
29. avg = calcavg(numerary, count)
endwhile

88. fread(fichier, n)
endf

------------------------------
6. numerary[count] = n
else
55. return(error)
print("bad input")

44. if (n>0)
3. while (not eof) do
22. fread(fichier, n)
11. count = 0
procedure avg
\( T' = \{t2, t3\} \)
Cost of Regression Testing

Cost = $C_x$

\begin{align*}
\text{Analysis} & \\
\text{Selective Retest} & \\
\end{align*}

Cost = $C_y$

We want $C_x < C_y$

Key is the test selection algorithm/technique

We want to maintain the same “quality of testing”
Selective-retest Approaches

- **Coverage-based approaches**
  - Rerun tests that could produce different output than the original program. Use some coverage criterion as a guide

- **Minimization approaches**
  - Minimal set of tests that must be run to meet some structural coverage criterion
    - E.g., every program statement added to or modified for $P'$ be executed (if possible) by at least one test in $T$
Selective-retest Approaches

- **Safe approaches**
  - Select every test that may cause the modified program to produce different output than the original program
    - E.g., every test that when executed on P, executed at least one statement that has been deleted from P, at least one statement that is new in or modified for P'

- **Data-flow coverage-based approaches**
  - Select tests that exercise data interactions that have been affected by modifications
    - E.g., select every test in T, that when executed on P, executed at least one def-use pair that has been deleted from P', or at least one def-use pair that has been modified for P'
Selective-retest Approaches

- Ad-hoc/random approaches
  - Time constraints
  - No test selection tool available
    - E.g., randomly select n test cases from T
Factors to consider

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

• Hypothesis
  - Non-random techniques are more effective than random techniques but are much more expensive
  - The composition of the original test suite greatly affects the cost and benefits of test selection techniques
  - Safe techniques are more effective and more expensive than minimization techniques
  - Data-flow coverage based techniques are as effective as safe techniques, but can be more expensive
  - Data-flow coverage based techniques are more effective than minimization techniques but are more expensive
Measure

- Costs and benefit of several test selection algorithms
- Developed two models
  - Calculating the cost of using the technique w.r.t. the retest-all technique
  - Calculate the fault detection effectiveness of the resulting test case
Modeling Cost

- Did not have implementations of all techniques
  - Had to simulate them
- Experiment was run on several machines (185,000 test cases) – results not comparable
- Simplifying assumptions
  - All test cases have uniform costs
  - All sub-costs can be expressed in equivalent units
    - Human effort, equipment cost
Modeling Cost

- **Cost of regression test selection**
  - Cost = A + E(T')
  - Where A is the cost of analysis
  - And E(T') is the cost of executing and validating tests in T'
  - Note that E(T) is the cost of executing and validating all tests, i.e., the retest-all approach
  - Relative cost of executing and validating = |T'|/|T|
Modeling Fault-detection

- Per-test basis
  - Given a program $P$ and
  - Its modified version $P'$
  - Identify those tests that are in $T$ and reveal a fault in $P'$, but that are not in $T'$
  - Normalize above quantity by the number of fault-revealing tests in $T$

- Problem
  - Multiple tests may reveal a given fault
Modeling Fault-detection

• **Per-test-suite basis**
  
  - **Three options**
    
    • The test suite is inadequate
      
      - No test in $T$ is fault revealing, and thus, no test in $T'$ is fault revealing
    
    • Same fault detection ability
      
      - Some test in both $T$ and $T'$ is fault revealing
    
    • Test selection compromises fault-detection
      
      - Some test in $T$ is fault revealing, but no test in $T'$ is fault revealing
  
  - **Percentage of cases in which $T'$ contains no fault-revealing tests**
Experimental Design

- 6 C programs
- Test suites for the programs
- Several modified versions

<table>
<thead>
<tr>
<th>Program</th>
<th>Functions</th>
<th>Lines</th>
<th>Versions</th>
<th>Avg T Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>21</td>
<td>516</td>
<td>32</td>
<td>398</td>
</tr>
<tr>
<td>printtokens2</td>
<td>19</td>
<td>483</td>
<td>10</td>
<td>389</td>
</tr>
<tr>
<td>schedule2</td>
<td>16</td>
<td>297</td>
<td>10</td>
<td>234</td>
</tr>
<tr>
<td>schedule</td>
<td>18</td>
<td>299</td>
<td>9</td>
<td>225</td>
</tr>
<tr>
<td>totinfo</td>
<td>7</td>
<td>346</td>
<td>23</td>
<td>199</td>
</tr>
<tr>
<td>tcas</td>
<td>9</td>
<td>138</td>
<td>41</td>
<td>83</td>
</tr>
</tbody>
</table>

Table 1: Experimental Subjects.
Test Suites and Versions

• Given a test pool for each program
  - Black-box test cases
    • Category-partition method
  - Additional white-box test cases
    • Created by hand
    • Each (executable) statement, edge, and def-use pair in the base program was exercised by at least 30 test cases

• Nature of modifications
  - Most cases single modification
  - Some cases, 2-5 modifications
Versions and Test Suites

- Two sets of test suites for each program
  - **Edge-coverage based**
    - 1000 edge-coverage adequate test suites
    - To obtain test suite $T$, for program $P$ (from its test pool): for each edge in $P$'s CFG, choose (randomly) from those tests of pool that exercise the edge (no repeats)
  - **Non-coverage based**
    - 1000 non-coverage-based test suites
    - To obtain the $k^{th}$ non-coverage based test suite, for program $P$: determine $n$, the size of the $k^{th}$ coverage-based test suite, and then choose tests randomly from the test pool for $P$ and add them to $T$, until $T$ contains $n$ test cases
Another look at the subjects

For each program
- 1000 edge-coverage based test suites:
- 1000 non-coverage based test suites:

<table>
<thead>
<tr>
<th>Program</th>
<th>Functions</th>
<th>Lines</th>
<th>Versions</th>
<th>Avg T Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>21</td>
<td>516</td>
<td>32</td>
<td>398</td>
</tr>
<tr>
<td>printtokens2</td>
<td>19</td>
<td>483</td>
<td>10</td>
<td>389</td>
</tr>
<tr>
<td>schedule2</td>
<td>16</td>
<td>297</td>
<td>10</td>
<td>234</td>
</tr>
<tr>
<td>schedule</td>
<td>18</td>
<td>299</td>
<td>9</td>
<td>225</td>
</tr>
<tr>
<td>totinfo</td>
<td>7</td>
<td>346</td>
<td>23</td>
<td>199</td>
</tr>
<tr>
<td>tcas</td>
<td>9</td>
<td>138</td>
<td>41</td>
<td>83</td>
</tr>
</tbody>
</table>

Table 1: Experimental Subjects.
Test Selection Tools

• **Minimization technique**
  - Select a minimal set of tests that cover modified edges

• **Safe technique**
  - DejaVu
    • we discussed the details earlier in this lecture

• **Data-flow coverage based technique**
  - Select tests that cover modified def-use pairs

• **Random technique**
  - Random(n) randomly selects n% of the test cases

• **Retest-all**
Variables

- **The subject program**
  - 6 programs, each with a variety of modifications

- **The test selection technique**
  - Safe, data-flow, minimization, random(25), random(50), random(75), retest-all

- **Test suite composition**
  - Edge-coverage adequate
  - random
Measured Quantities

- Each run
  - Program P
  - Version P'
  - Selection technique M
  - Test suite T

- Measured
  - The ratio of tests in the selected test suite T' to the tests in the original test suite
  - Whether one or more tests in T' reveals the fault in P'
Dependent variables

- Average reduction in test suite size
- Fault detection effectiveness
  - Percentage of test suites in which $T'$ does not reveal a fault in $P'$
**Number of runs**

- For each subject program, from the test suite universe
  - Selected 100 edge-coverage adequate
  - And 100 random test suites
- For each test suite
  - Applied each test selection method
  - Evaluated the fault detection capability of the resulting test suite
Fault-detection Effectiveness

Percentage of test suites in which $T'$ does not reveal a fault in $P'$
How to read the graphs
How to read the graphs

Entire structure represents a data distribution
How to read the graphs

Entire structure represents a data distribution

Box’s height spans the central 50% of the data
How to read the graphs

Entire structure represents a data distribution

Box's height spans the central 50% of the data

Upper quartile
How to read the graphs

Entire structure represents a data distribution

Box’s height spans the central 50% of the data

Upper quartile

Lower quartile
How to read the graphs

Entire structure represents a data distribution

Box’s height spans the central 50% of the data

Upper quartile
Median
Lower quartile
How to read the graphs
Fault-detection Effectiveness
Figure 2: Test suite reduction by method, conditioned on program.
Figure 3: Fault-detection effectiveness by selection method, conditioned on program.
Figure 4: Fault-detection effectiveness and test suite size, irrespective of analysis costs.
Conclusions

- Minimization produces the smallest and the least effective test suites
- Random selection of slightly larger test suites yielded equally good test suites as far as fault-detection is concerned
- Safe and data-flow nearly equivalent average behavior and analysis costs
  - Data-flow may be useful for other aspects of regression testing
- Safe methods found all faults (for which they has fault-revealing tests) while selecting (average) 74% of the test cases
Conclusions

• In certain cases, safe method could not reduce test suite size at all
  - On the average, slightly larger random test suites could be nearly as effective

• Results were sensitive to
  - Selection methods used
  - Programs
  - Characteristics of the changes
  - Composition of the test suites