<table>
<thead>
<tr>
<th></th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Testing III</td>
<td>• Group meetings</td>
<td>• Testing IV</td>
<td>• Section</td>
<td>• ZFR demos</td>
</tr>
<tr>
<td></td>
<td>• No reading</td>
<td></td>
<td></td>
<td>• ZFR due</td>
<td>• Progress report due</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Readings out</td>
</tr>
</tbody>
</table>
Today: symbolic & mutation testing

• Symbolic example from Michael Beder
  – Basic idea of symbolic testing is to consider inputs as symbols, not values
  – Track predicates and constraints over those symbols through the control flow graph (CFG)
  – Can help in determining inputs that will cause the execution of particular paths
• Mutation testing – an approach to assessing test suites
  – Systematically change (mutate) the program being tested
  – If the test suite cannot distinguish the original program from the mutated program, it has a weakness
Example
all variables are \textbf{ints}

\begin{verbatim}
I   a = read(b)  
II  c = 0  
III while (a > 1) {  
   IV    if (a^2 > c)  
   V       c = c + a  
   VI    a = a - 2  
}
VII write(c)
\end{verbatim}

What input(s) will take path:
(I,II) → III → IV → V → VI → III → IV → V → VI → III → VII
I  a = read(b)
II c = 0
III while (a > 1) {
    IV if (a^2 > c)
    V   c = c + a
    VI a = a - 2
}
VII write(c)

What input(s) will take path:
(I,II)→III → IV → V → VI→ III → IV → V → VI → III→ VII

### After-node [A,B,C] Condition

<table>
<thead>
<tr>
<th>After-node</th>
<th>[A,B,C]</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I,III)</td>
<td>(B,B,0)</td>
<td>true</td>
</tr>
<tr>
<td>III</td>
<td>(B,B,0)</td>
<td>B&gt;1</td>
</tr>
<tr>
<td>IV</td>
<td>(B,B,0)</td>
<td>B&gt;1∧B^2&gt;0 ⇔ B&gt;1</td>
</tr>
<tr>
<td>V</td>
<td>(B,B,B)</td>
<td>B&gt;1</td>
</tr>
<tr>
<td>VI</td>
<td>(B-2,B,B)</td>
<td>B&gt;1</td>
</tr>
<tr>
<td>III</td>
<td>(B-2,B,B)</td>
<td>B&gt;1∧B-2&gt;0 ⇔ B&gt;3</td>
</tr>
<tr>
<td>IV</td>
<td>(B-2,B,B)</td>
<td>B&gt;3∧(B-2)^2&gt;B ⇔ B&gt;4</td>
</tr>
<tr>
<td>V</td>
<td>(B-2,B,2B-2)</td>
<td>B&gt;4</td>
</tr>
<tr>
<td>VI</td>
<td>(B-4,B,2B-2)</td>
<td>B&gt;4</td>
</tr>
<tr>
<td>III</td>
<td>(B-4,B,2B-2)</td>
<td>B&gt;4∧(B-4)≤1 ⇔ B=5</td>
</tr>
</tbody>
</table>

Expected result for input B=5
What happens when solving …

- \( B > 3 \land (B-2)^2 > B \) (or such) is hard?
- Remember, we have to automate all these steps if they are going to be genuinely useful
- Come on Wednesday…
Mutation testing

- Mutation testing is an approach to evaluate – and to improve – test suites
- Basic idea
  - Create small variants of the program under test
  - If the tests don’t exhibit different behavior on the variants then the test suite is not sufficient
- The material on the following slides is due heavily to Pezzè and Young on fault-based testing
Estimation

- Given a big bowl of marbles, how can we estimate how many?
- Can’t count every marble individually
What if I also…

- … have a bag of 100 other marbles of the same size, but a different color (say, black) and mix them in?
- Draw out 100 marbles at random and find 20 of them are black
- How many marbles did we start with?
Estimating test suite quality

• Now take a program with bugs and create 100 variations each with a new and distinct bug
  – Assume the new bugs are exactly like real bugs in every way
• Run the test suite on all 100 new variants
  – ... and the tests reveal 20 of the bugs
  – ... and the other 80 program copies do not fail
• What does this tell us about the test suite?
Basic Assumptions

• The idea is to judge effectiveness of a test suite in finding real faults by measuring how well it finds seeded fake faults

• Valid to the extent that the seeded bugs are representative of real bugs: not necessarily identical but the differences should not affect the selection
Mutation testing

- A mutant is a copy of a program with a mutation: a syntactic change that represents a seeded bug
  - Ex: change \( i < 0 \) to \( i \leq 0 \)
- Run the test suite on all the mutant programs
- A mutant is killed if it fails on at least one test case
  - That is, the mutant is distinguishable from the original program by the test suite, which adds confidence about the quality of the test suite
- If many mutants are killed, infer that the test suite is also effective at finding real bugs
Mutation testing assumptions

• Competent programmer hypothesis: programs are nearly correct
  – Real faults are small variations from the correct program and thus mutants are reasonable models of real buggy programs

• Coupling effect hypothesis: tests that find simple faults also find more complex faults
  – Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults, too
Mutation Operators

- Syntactic change from legal program to legal program and are thus specific to each programming language

- Ex: constant for constant replacement
  - from $(x < 5)$ to $(x < 12)$
  - Maybe select from constants found elsewhere in program text

- Ex: relational operator replacement
  - from $(x \leq 5)$ to $(x < 5)$

- Ex: variable initialization elimination
  - from `int x = 5;` to `int x;`
Live mutants scenario

• Create 100 mutants from a program
  – Run the test suite on all 100 mutants, plus the original program
  – The original program passes all tests
  – 94 mutant programs are killed (fail at least one test)
  – 6 mutants remain alive
• What can we learn from the living mutants?
How mutants survive

• A mutant may be equivalent to the original program
  – Maybe changing $x < 0$ to $x \leq 0$ didn’t change the output at all!
  – The seeded “fault” is not really a “fault” — determining this may be easy or hard or in the worst case undecidable

• Or the test suite could be inadequate
  – If the mutant could have been killed, but was not, it indicates a weakness in the test suite
  – But adding a test case for just this mutant is a bad idea — why?
Weak mutation: a variation

- There are lots of mutants – the number of mutants grows with the square of program size
- Running each test case to completion on every mutant is expensive
- Instead execute a “meta-mutant” that has many of the seeded faults in addition to executing the original program
  - Mark a seeded fault as “killed” as soon as a difference in an intermediate state is found – don’t wait for program completion
  - Restart with new mutant selection after each “kill”
Statistical Mutation: another variation

• Running each test case on every mutant is expensive, even if we don’t run each test case separately to completion
• Approach: Create a random sample of mutants
  – May be just as good for assessing a test suite
  – Doesn’t work if test cases are designed to kill particular mutants
In real life ...

- Fault-based testing is a widely used in semiconductor manufacturing
  - With good fault models of typical manufacturing faults, e.g., “stuck-at-one” for a transistor
  - But fault-based testing for design errors – as in software – is more challenging
- Mutation testing is not widely used in industry
  - But plays a role in software testing research, to compare effectiveness of testing techniques
- Some use of fault models to design test cases is important and widely practiced
Summary

• If bugs were marbles ...
  – We could get some nice black marbles to judge the quality of test suites

• Since bugs aren’t marbles ...
  – Mutation testing rests on some troubling assumptions about seeded faults, which may not be statistically representative of real faults

• Nonetheless ...
  – A model of typical or important faults is invaluable information for designing and assessing test suites
<table>
<thead>
<tr>
<th>Week 5-6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monday</strong></td>
</tr>
<tr>
<td>• Testing I</td>
</tr>
<tr>
<td>• No reading</td>
</tr>
<tr>
<td><strong>Monday</strong></td>
</tr>
<tr>
<td>• Testing III</td>
</tr>
<tr>
<td>• Readings due</td>
</tr>
</tbody>
</table>