### Week 6

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Testing III</td>
<td>• Group</td>
<td>• Testing IV</td>
<td>• Section</td>
<td>• ZFR demos</td>
</tr>
<tr>
<td>• Reading due</td>
<td>meetings</td>
<td></td>
<td>• ZFR due</td>
<td>• Progress report</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>due</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>• Readings out</td>
</tr>
</tbody>
</table>

- Concolic testing – combine symbolic and concrete testing
- Back to the basics of testing
Concolic

- Symbolic execution (or evaluation or testing) counts on a constraint solver (a kind of automated theorem prover) to solve for path conditions that will exercise specific branches in the CFG – we saw this last lecture, and we’ll see it again today
- The technology for constraint solvers is impressive, but there are still some constraints that cannot be automatically solved
- Concolic approaches combine concrete and symbolic execution to increase code coverage and, ideally, find bugs that would be otherwise hard to find
- KLEE, Cute, DART, etc. are examples of tools supporting concolic testing
To another’s slide deck for examples

- From Pınar Sağlam – elided to examples
- Two examples, swapped in our slide deck
  - The (now) second example (starting at slide 12) is really only symbolic execution, but shows how it works on data structures with some complexity
Back to partitioning

• Ideal test suite
  – Identify sets with same behavior
  – Try one input from each set

• Two problems
  1. Notion of the same behavior is subtle
     Naive approach: execution equivalence
     Better approach: revealing subdomains
  2. Discovering the sets requires perfect knowledge
     – Use heuristics to approximate cheaply
Naive Approach: Execution Equivalence

```c
// returns: x < 0 => returns -x
// otherwise => returns x
int abs(int x) {
    if (x < 0) return -x;
    else return x;
}
```

- All $x < 0$ are execution equivalent – that is, the program takes same sequence of steps for any $x < 0$
- All $x \geq 0$ are also execution equivalent
- Suggests that $\{-3, 3\}$, for example, is a good test suite
So, what’s the problem?

There are two execution paths, but combined with the specification there are three separate behaviors:

- $x < -2$
- $x = -2 \lor x = -1$
- $x \geq 0$

{-3, 3} does not reveal the error behaviors!
Heuristic: Revealing Subdomains

- A subdomain is a subset of possible inputs
- A subdomain is *revealing* for error E if either
  - Every input in that subdomain triggers error E, or
  - No input in that subdomain triggers error E
- Need test only one input from a given subdomain
  - If subdomains cover the entire input space, then we are guaranteed to detect the error if it is present
- The trick is to guess these revealing subdomains
Ex: buggy \texttt{abs}, revealing subdomains?

```c
int abs(int x) {
    if (x < -2) return -x;
    else return x;
}
```

- Possible subdomains
  - \{-1\}
  - \{-2\}
  - \{-2,-1\}
  - \{-3,-2,-1\}

- Which of these is not a revealing subdomain for this bug?
- Which of these is the best revealing subdomain for this bug?
Heuristics for Designing Test Suites

• A good heuristic gives
  – few subdomains
  – ∀ errors E in some class of errors,
    – high probability that some subdomain is revealing for E
• Different heuristics target different classes of errors
  – In practice, combine multiple heuristics
Black Box Testing

• Heuristic: Explore alternate specification paths
  – Procedure is a black box: interface visible, internals hidden
• Example
  – \( \text{int max(int } a, \text{ int } b) \)
    \( \text{// effects: } a > b \Rightarrow \text{ returns } a \)
    \( \text{// } a < b \Rightarrow \text{ returns } b \)
    \( \text{// } a = b \Rightarrow \text{ returns } a \)
• Three paths, so three test cases
  – \((4, 3) \Rightarrow 4\) (i.e. any input in the subdomain \(a > b\))
  – \((3, 4) \Rightarrow 4\) (i.e. any input in the subdomain \(a < b\))
  – \((3, 3) \Rightarrow 3\) (i.e. any input in the subdomain \(a = b\))
More Complex Example

```java
int find(int[] a, int value) throws Missing
// returns: the smallest i such
//          that a[i] == value
// throws: Missing if value is not in a
```

- Two obvious tests:
  - ( [4, 5, 6], 5 ) => 1
  - ( [4, 5, 6], 7 ) => throw Missing

- Must hunt for multiple cases in the specification
  - ( [4, 5, 5], 5 ) => 1
Heuristic: Boundary Testing

- Create tests at the edges of subdomains
  - off-by-one bugs
  - forgot to handle empty container
  - overflow errors in arithmetic
  - aliasing
- Small subdomains at the edges of the “main” subdomains have a high probability of revealing these common errors
- Also, you might have misdrawn the boundaries
Boundary Testing

• To define the boundary, need a distance metric
  – Define adjacent points

• One approach
  – Identify basic operations on input points
  – Two points are adjacent if one basic operation apart

• Point is on a boundary if either
  – There exists an adjacent point in a different subdomain
  – Some basic operation cannot be applied to the point

• Example: list of integers
  – Basic operations: create, append, remove
  – Adjacent points: <[2,3],[2,3,3]>, <[2,3],[2]>
  – Boundary point: [] (can’t apply remove integer)
Boundary Cases: Aliases

\begin{verbatim}
<E> void appendList(List<E> src, List<E> dest) {
    // modifies: src, dest
    // effects: removes all elements of src and appends them in reverse order to the end of dest

    while (src.size()>0) {
        E elt = src.remove(src.size()-1);
        dest.add(elt)
    }
}

• What happens if \texttt{src} and \texttt{dest} refer to the same thing? This is aliasing, and it’s easy to forget! Watch out for shared references in inputs
\end{verbatim}
Regression Testing

• Whenever you find a bug
  – Store the input that elicited that bug, plus the correct output
  – Add these to the test suite
  – Verify that the test suite fails
  – Fix the bug
  – Verify the fix
• Ensures that your fix solves the problem
• Helps to populate test suite with good tests
• Protects against reversions that reintroduce bug
  – It happened at least once, and it might happen again
Rules of Testing

• First rule of testing: **Do it early and do it often**
  – Best to catch bugs soon, before they have a chance to hide.
  – Automate the process if you can
  – Regression testing will save time
• Second rule of testing: **Be systematic**
  – If you randomly thrash, bugs will hide in the corner until you're gone
  – Writing tests is a good way to understand the spec
    • Think about revealing domains and boundary cases
    • If the spec is confusing \( \rightarrow \) write more tests
  – Spec can be buggy too
    • Incorrect, incomplete, ambiguous, and missing corner cases
  – When you find a bug \( \rightarrow \) write a test for it first and then fix it
<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>Testing III</td>
<td>Reading due</td>
<td>Group meetings</td>
<td>Testing IV</td>
<td>Section</td>
<td>ZFR demos</td>
</tr>
<tr>
<td>Reading due</td>
<td></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>