Test Inputs, Oracles and Generation
One-Slide Summary

- Formally, a **test case** consists of an **input (data)**, an **oracle (output)**, and a **comparator**.

- Test inputs determine the behavior of the program. High-coverage inputs can be **generated automatically** through path enumeration, path predicates and mathematical constraint solving.

- Test oracles correspond to what the program should do. Generating them is an expensive **problem**; it can be done automatically through **invariants** and **mutation**.

- **Test suite minimization** finds the smallest subset of tests that meet a coverage goal.
The Story So Far ...

- Testing is **very expensive** (e.g., 35% of total IT spending).
- Test suite **quality metrics** support informed comparisons between tests.
- But where do we get one test, much less many to compare?
Outline

• Test inputs
• Test input generation
• Test oracles
• Test oracle generation
• Test minimization

• “Kill it with Math” vs. “Humans Are Central”
What is a test?

- Formally, a **test case** has three components: the **test input** (or **data**), the **test oracle** (or expected output), and the **comparator**.
- Sometimes called the Oracle-Comparator model.

```bash
./prog < input > output ; diff -b output oracle
```
Comparator

- Many test cases use “must match exactly” as the comparator
- But officially it could be more general
  - Known random output, precision limits, embedded dates, etc.
Non-Trivial Comparator Example

- jsoup/internal/ConstrainableInputStreamTest.java
- (from Homework 2)

```java
@Test
public void noLimitAfterFirstRead() throws IOException {
    int bufferSize = 5 * 1024;

    String url = "http://direct.infohound.net/tools/large.html"; // 280 K
    BufferedInputStream inputStream = Jsoup.connect(url).execute().bodyStream();

    assertTrue(inputStream instanceof ConstrainableInputStream);
    ConstrainableInputStream stream = (ConstrainableInputStream) inputStream;

    // simulates parse which does a limited read first
    stream.mark(bufferSize);
    ByteBuffer firstBytes = stream.readToByteBuffer(bufferSize);
    byte[] array = firstBytes.array();
    String firstText = new String(array, "UTF-8");
    assertTrue(firstText.startsWith("<html><head><title>Large"));
    assertEquals(bufferSize, array.length);
```
Test Data

• What are all the inputs to a test?

  • Many programs (especially student programs) read from a file or stdin ...

  • But what else is “read in” by a program and may influence its behavior?
Test Inputs

- User Input (e.g., GUI)
- Environment Variables, Command-Line Args
- Scheduler Interleavings
- Data from the Filesystem
  - User configuration, data files
- Data from the Network
  - Server and service responses
Operating System Philosophy

- “Everything is a file.”
- After a few libraries and levels of indirection, reading from the user's keyboard boils down to opening a special device file (e.g., /dev/ttyS0) and reading from it
  - Similarly with mouse clicks, GUI commands, etc.
- Ultimately programs can only interact with the outside world through system calls
  - open, read, write, socket, fork, gettimeofday
  - Those (plus OS scheduling, etc.) are the full inputs
Test Input Generation

• We want to generate high quality tests
  • **Automatically**!

• From test suite metrics we prefer some tests

• Statement Coverage: visit every line
• Branch Coverage: visit every →true, →false
• Path Coverage: visit every path
Path Coverage

```python
foo(a,b,c,d,e,f):
    if a < b: this
    else: that
    if c < d: foo
    else: bar
    if e < f: baz
    else: quoz
```

- How many *paths*?
Path Coverage

foo(a,b,c,d,e,f):
  if a < b: this
  else: that
  if c < d: foo
  else: bar
  if e < f: baz
  else: quoz

• There are 8 paths, but only 6 branch coverage edges
Branch vs. Path

• If you have N sequential (or serial) if-statements ...

• There are $2N$ branch edges
  • Which you could cover in 2 tests!
    • One always goes left, one always right

• But there are $2^N$ paths
  • You need $2^N$ tests to cover them

• Path coverage subsumes branch coverage
Path Test Input Generation

- Consider generating test inputs to cover a path
  - If we can do that, branch, stmt, etc., are easy!

- Solve this problem with math

- A **path predicate** (or **path condition**, or **path constraint**) is a boolean formula over program variables that is true when the program executes the given path
Path Predicate Example

• Consider the highlighted path
  • a.k.a. “False, False, True”
• Its path predicate is
  • a >= b && c >= d && e < f
• When the path predicate is true, control flow follows the given path
• So what should we do to make a test input that covers this path?
Solving Systems of Equations

- A **satisfying assignment** is a mapping from variables to values that makes a predicate true.

- One satisfying assignment for $a \geq b \land c \geq d \land e < f$
  
  - \text{Is}
  
  $a=5, \ b=4, \ c=3, \ d=2, \ e=1, \ f=2$

- Another \text{Is}
  
  $a=0, \ b=0, \ c=0, \ d=0, \ e=0, \ f=1$
Producing Satisfying Assignments

• Ask Humans
  • Labor-intensive, expensive, etc.
• Repeatedly guess randomly
  • Works surprisingly well (when answers are not sparse)
• Use an automated theorem prover
  • cf. Wolfram, MatLab, Mathematica, etc.
  • Works very well on restricted types of equations (e.g., linear but not arbitrary polynomial, etc.)
Test Input Generation Plan

- Consider generating high-branch-coverage tests for a method ...

- Enumerate “all” paths in the method
- For each path, collect the path predicate
- For each path predicate, solve it
  - A solution is a satisfying assignment of values to input variables → those are your test input
  - None found? Dead code, tough predicate, etc.
Enumerating Paths

• What could go wrong with enumerating paths in a method?
Enumerating Paths

• What could go wrong with enumerating paths in a method?

• There could be arbitrarily many!

    while a<b:
        a = a + 1
    return a

• One path corresponds to executing the loop once, another to twice, another to three times, etc.
Path Enumeration Approximations

• Typical Approximations
  
  • Consider only acyclic paths (corresponds to taking each loop zero times or one time)
  • Consider only taking each loop at most $k$ times
  • Enumerate paths breadth-first or depth-first and stop after $k$ paths have been enumerated

• For more information, take a *Programming Languages, Compilers or Theory* class
Collecting Path Predicates

- Now we have a path through the program
- What could go wrong with collecting the path predicate?

\[
\begin{align*}
\sqrt{\heartsuit} &= \text{?} & \cos \heartsuit &= \text{?} \\
\frac{d}{dx} \heartsuit &= \text{?} & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \heartsuit &= \text{?} \\
F\{\heartsuit\} &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{it \heartsuit} dt = \text{?}
\end{align*}
\]

My normal approach is useless here.
Path Predicate

- The path predicate may not be expressible in terms of the inputs we control

```python
foo(a,b):
    str1 = read_from_url("abc.com")
    str2 = read_from_url("xyz.com")
    if (str1 == str2):
        bar()
```

- Suppose we want to exercise the path that calls bar. One predicate is `str1==str2`. What do you assign to `a` and `b`?
Path Predicate Woes

• Typical solutions:
  • “We don't care.”
  • Collect up the path predicate as best you can
  • Ask the solver to solve it in terms of the input variables
  • If it can't
    • … either because the math is too hard
    • … or because the variables are out of our control
  • Then we don't generate a test input exercising that path. Best effort.
Test Data Generation

- One of the earliest approaches was DART (Directed Automatic Random Testing)
- Their example program has three paths:
  - False, True-False, True-True
- Predicates:
  - $z = y \land x \neq z$
  - $z = y \land x = z \land y \neq x + 10$
  - $x = y \land x = z \land y = x + 10$
- Give me three solutions in terms of $x$ and $y$.

```c
int f(int x, int y) {
    int z;
    z = y;
    if (x == z)
        if (y == x + 10)
            abort();
    return 0;
}
```
Microsoft's Pex Tool

• Pex is a test input generation tool integrated into Visual Studio
  • It has special handling for pointers, is language-independent, etc., but otherwise works just like what we covered here
  • Other tools (e.g., jCUTE for Java) exist
### Does it Work?

<table>
<thead>
<tr>
<th>Class</th>
<th>Blocks</th>
<th>Block Coverage</th>
<th>Arcs</th>
<th>Arc Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (mostly stateless methods)</td>
<td>&gt;300</td>
<td>95%</td>
<td>&gt;400</td>
<td>90%</td>
</tr>
<tr>
<td>B (mostly stateless methods)</td>
<td>&gt;100</td>
<td>97%</td>
<td>&gt;200</td>
<td>94%</td>
</tr>
<tr>
<td>C (stateful)</td>
<td>&gt;200</td>
<td>76%</td>
<td>&gt;300</td>
<td>65%</td>
</tr>
<tr>
<td>D (parsing code)</td>
<td>&gt;500</td>
<td>81%</td>
<td>&gt;800</td>
<td>73%</td>
</tr>
<tr>
<td>E (numerical algorithms)</td>
<td>&gt;400</td>
<td>71%</td>
<td>&gt;600</td>
<td>67%</td>
</tr>
<tr>
<td>F (numerical algorithms)</td>
<td>&gt;100</td>
<td>82%</td>
<td>&gt;200</td>
<td>79%</td>
</tr>
<tr>
<td>G (numerical algorithms)</td>
<td>&gt;100</td>
<td>98%</td>
<td>&gt;100</td>
<td>97%</td>
</tr>
<tr>
<td>H (numerical algorithms)</td>
<td>&gt;200</td>
<td>71%</td>
<td>&gt;200</td>
<td>61%</td>
</tr>
<tr>
<td>I (numerical algorithms)</td>
<td>&gt;200</td>
<td>97%</td>
<td>&gt;300</td>
<td>96%</td>
</tr>
</tbody>
</table>

- Why are these MS Dot.Net classes anonymous?
- What are block and arc coverage?
So, did we win?

- We want to automatically generate test cases
- We have an approach that works well in practice:
  - Enumerate some paths
  - Extract their path constraints
  - Solve those path constraints
- What are we missing?
We Forgot Oracles!

- We know to generate test inputs
  - e.g., “for high coverage, run f(1,0) and f(-5,-7)”
- But we don't know what the answer is supposed to be when you do that!
- So we cannot tell if a program is passing or failing.

- Well ... maybe we can still salvage something. Thoughts?
Test Generation $\rightarrow$ Bug Finding

- If your program crashes on that input → bad
- “This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code ... EXE works well on real code, finding bugs along with inputs that trigger them in: the BSD and Linux packet filter implementations, the udhcpd DHCP server, the pcre regular expression library, and three Linux file systems.”

[Cadar et al. EXE: Automatically Generating Inputs of Death. CCS 2006.]
Big Problem

• In general, though, we're going to need both the question and the answer!

• But don't panic yet ...

• No need to throw in the towel ...
Oracles

• “If Croesus goes to war he will destroy a great empire.”
  - Barbara Gordon The Oracle at Delphi, on whether Croesus should go to war against the Persians

• Oracles are tricky.

• Many believe that formally writing down what a program should do is as hard as coding it.
  • (We return to this topic later.)
The Oracle Problem

- The **Oracle Problem** is the difficulty and cost of determining the correct test oracle for a given input.
  - “What *should* the program do?”
- It is expensive both for humans and for machines.
- An **implicit oracle** is one associated with the language or architecture, rather than program-specific semantics (e.g., “don't segfault”, “don't loop forever”).
Aside: Philosophy

• The difficulty here should not be surprising.
• Recall from Ethics that it is easier to make negative moral edicts (“Do not steal”) than it is to elaborate positive ones (“Here is what it means to be a generous person …”)
• Similarly, it is much easier to make negative program edicts (“Do not crash”) than it is to elaborate positive ones (“Here is what it means to be a good webserver …”)

35
Idea: Use The Program

- In this setting we do *have* the program
  - We're trying to generate tests for it ...
- Perhaps the *program itself* can somehow tell us what its correct behavior should be
  - But how?
Insight: Competent Programmers

- We return to the assumption that the program is mostly correct (where was this from?)
- If I run the program ten different times and every time we have index == array_len - 1
  - ... perhaps that is the test oracle we want:  
    ```java
    assertEquals(index, array_len-1);
    ```
  - That is, “it should be true every time”
- An **invariant** is a predicate over program expressions that is true on every execution.
  - High-quality invariants can serve as test oracles
Learning Invariants

• We can **learn** (or **infer**) program invariants by running the program many times and noting what is always true of the output
  
  • e.g., if we run sqrt() many times, we may learn retval>=0
Learning Invariants

- We can **learn** (or **infer**) program invariants by running the program many times and noting what is always true of the output
  - e.g., if we run \( \text{sqrt()} \) many times, we may learn \( \text{retval} \geq 0 \)
- Simple implementation: start with a big list of possible invariants (e.g., \( \text{retval} = 0 \), \( \text{retval} = 5 \), \( \text{retval} \geq 0 \), etc.) and, on every run, cross off those that are falsified
  - Recall: by definition an invariant is true on all runs
Learning Invariants

- We can **learn** (or **infer**) program invariants by running the program many times and noting what is always true of the output
  - e.g., if we run sqrt() many times, we may learn retval>=0

- Simple implementation: start with a big list of possible invariants (e.g., retval=0, retval=5, retval>=0, etc.) and, on every run, cross off those that are falsified
  - Recall: by definition an invariant is true on all runs
Common vs. Correct

• In some sense, we are assuming that common behavior (or behavior we can observe) is correct behavior.

• This is like learning the rules of English by reading high school essays. What could go wrong?

ME FAIL ENGLISH?

THAT'S UNPOSSIBLE
Bad Invariants

- Consider the following situations

- We test sqrt once, on sqrt(9), and learn the invariant: retval==3

- We test findNode thousands of times, and learn the invariant: pointer%4==0
Fixing This Mess

• The “sqrt == 3” issue can be partially addressed with more random inputs
• The “ptr % 4 == 0” issue is more troubling
  • It is only coincidentally correct here
  • (Why do we care? Hint: cost!)
• Competent Programmers: in general, every line of code matters to correctness
The Chain of Reasoning

- Competent Programmers: in general, every line of human-written code matters to human-intended correctness
- So if an invariant or oracle captures human-intended correctness, there must be at least one line of code that ensures it
- So if I poke and mutate your programs, I should be able to falsify the invariant!
  - If I can't, it was coincidental and not a product of the code you actually wrote!
Example

- Suppose we have tested this on 1, 9, 16, 30
- Candidate Invariants:
  - retval < reval+1
  - retval <= 6
  - x >= retval*retval
- What do we do?

```c
int floorSqrt(int x) {
    // Base cases
    if (x == 0 || x == 1)
        return x;

    // Staring from 1, try all numbers until // i*i is greater than or equal to x.
    int i = 1, result = 1;
    while (result < x) {
        if (result == x)
            return result;
        i++;
        result = i*i;
    }
    return i-1;
}
```
Example

- Suppose we have tested this on 1, 9, 16, 30

- Candidate Invariants:
  - $\text{retval} < \text{reval} + 1$
  - $\text{retval} \leq 6$
  - $x \geq \text{retval} \times \text{retval}$

- What do we do?

```c
// Returns floor of square root of x
int floorSqrt(int x)
{
    // Base cases
    if (x == 0 || x == 1)
        return x;

    // Starting from 1, try all numbers until
    // $i \times i$ is greater than or equal to $x$.
    int i = 1, result = 1;
    while (result < x)
    {
        if (result == x)
            return result;
        i++;
        result = i*i;
    }
    return i-1;
}
```

- Never ruled out by any mutation, dropped!
- Ruled out by trying more inputs (e.g., 81), dropped!
- Falsified by some mutations (which?), retained!
EvoSuite

• This oracle-generation approach is implemented in the **EvoSuite** tool
  • It generates high-coverage unit tests for Java
  • It is award-winning, takes first place in competitions as recently as 2017, etc.
  • You will get a chance to try it in Homework 2!

• EvoSuite is an instance of **search-based software engineering**, a topic we'll return to on March 14th
An Embarrassment of Riches

• At this point, we may actually have *too many* test cases
  • Surprisingly, this is normal in industry: you almost always have far too few or far too many!
  • Recall Google optional reading from last week
• This is especially true when using automated test generation tools
  • Which many produce many tests but lower-quality ones than humans would produce
  • A big cost problem!
Test Suite Minimization

- Given a set of test cases and coverage information for each one, the **test suite minimization** problem is to find the minimal number of test cases that still have the maximum coverage.

Example

- T1 covers lines 1,2,3
- T2 covers lines 2,3,4,5
- T3 covers lines 1,2
- T4 covers lines 1,6
Reveng of CS Theory

- You can add in details like the tests have different costs to run, but ignore that for now.
- How **hard is it** to solve the test suite minimization problem?
- What is a **correct** algorithm for it? Can we do better?
Questions?

- Homework 1b, 1c, 1d all due Monday!
  - They are *much* harder than 1a