Test Oracles and Mutation Testing

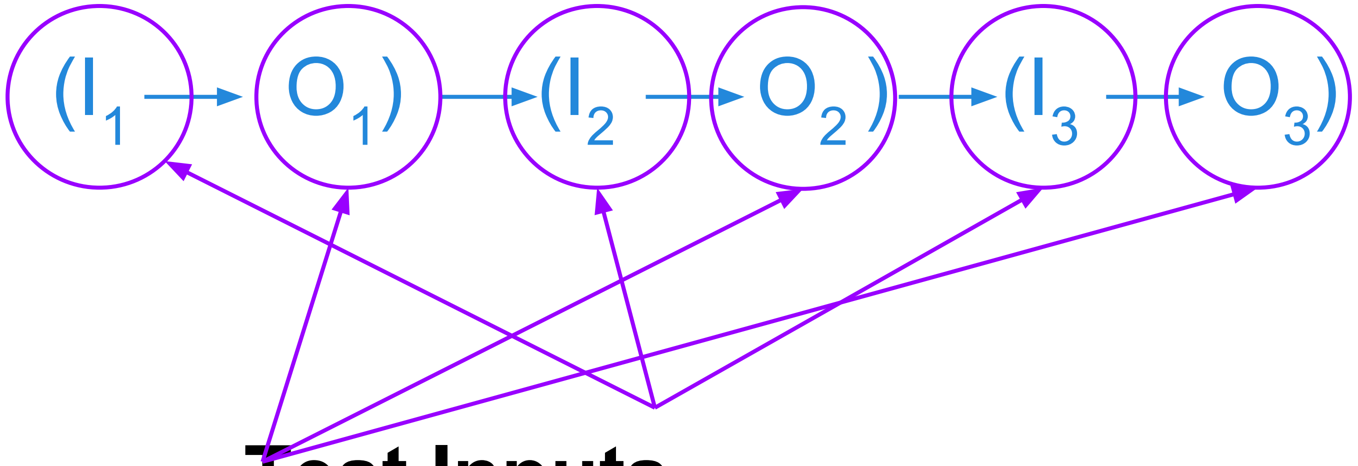
CSCE 740 - Lecture 23 - 11/18/2015

Software Testing - Back to the Basics

Tests are sequences of **stimuli** and **observations**. We care about input and output.

$$(I_1 \rightarrow O_1) \rightarrow (I_2 \rightarrow O_2) \rightarrow (I_3 \rightarrow O_3)$$

What Do We Need For Testing?


$(I_1 \rightarrow O_1) \rightarrow (I_2 \rightarrow O_2) \rightarrow (I_3 \rightarrow O_3)$

Test Inputs

How we “stimulate” the system.

if O_n = Expected(O_n)

then… Pass

else… Fail

Test Oracle

How we check the correctness of the resulting observation.

We Will Cover

Software Test Oracles

- Where do they come from?
- How do we create them?
- Why are they important for testing?

Test Oracle - Definition

If a software test is a sequence of activities (*stimuli and observations*), an **oracle** is a predicate that determines whether a given sequence is acceptable or not.

An oracle will respond with a *pass* or a *fail* verdict on the acceptability of any test sequence for which it is defined.

Test Oracle - Definition

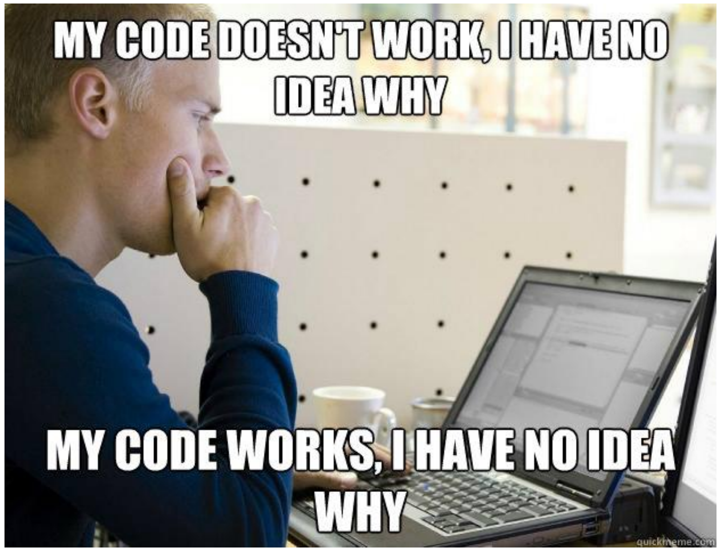
An oracle, as an artifact, can be broken into:

- **Oracle Information**
  - The information used by the oracle to judge the correctness of the observed behavior, given the set of inputs.
- **Oracle Procedure**
  - How that information is used to arrive at a verdict.
  - Commonly...

```
(value(system output) == value(expected output))
```

Where Do We Get Test Oracles?

Most commonly:
Developers write inputs and oracles by hand.


MY CODE DOESN'T WORK, I HAVE NO IDEA WHY
MY CODE WORKS, I HAVE NO IDEA WHY
quickmeme.com

- Large amount of manual effort and time required to create tests
- Will not be able to run many tests. Did you choose the right inputs?

The “Test Oracle Problem”

We are good at coming up with new input...

$$\begin{tikzcd}
 & f(\textbf{int}) \arrow[dl] \arrow[d] \arrow[dr] & \\
f(0) & f(6) & f(1123)
\end{tikzcd}$$

But, it is **much harder** to automatically check results.

f(0) = **?**

f(6) = **?**

f(1123) = **?**

Can We Design Better Test Oracles?

Judging the Judge

What properties do we want out of an oracle?

- Cost to Build (Per Test and Overall)
    - How much effort goes into building it?
    - Want **low cost**.
- Accuracy of Verdicts
    - Can it give the wrong answer?
    - Want **high accuracy**.
- Completeness
    - Can it offer verdicts for multiple test cases?
    - What kind of situations is it useful for?
    - Want **high completeness**.

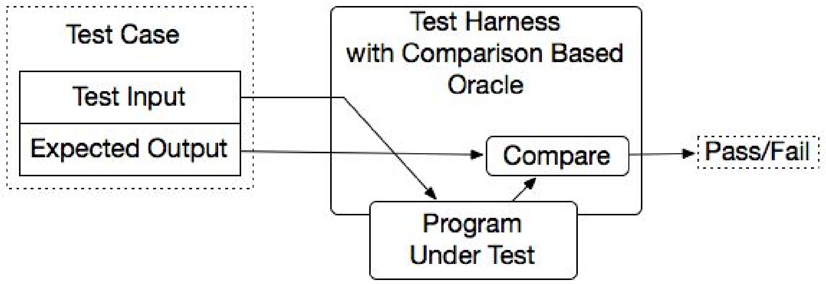
Types of Oracles

- **Specified Oracles**
  - Developers, using the requirements, formally specify properties that correct behavior should follow.
- **Derived Oracles**
  - An oracle is derived from development artifacts or system executions.
- **Implicit Oracles**
  - An oracle judges correctness using properties expected of many programs.
- **Human Oracles**
  - How do you handle the lack of an oracle?

Specified Oracles

Specified Oracles

Specified Oracles judge behavior using a human-created specification of correctness.

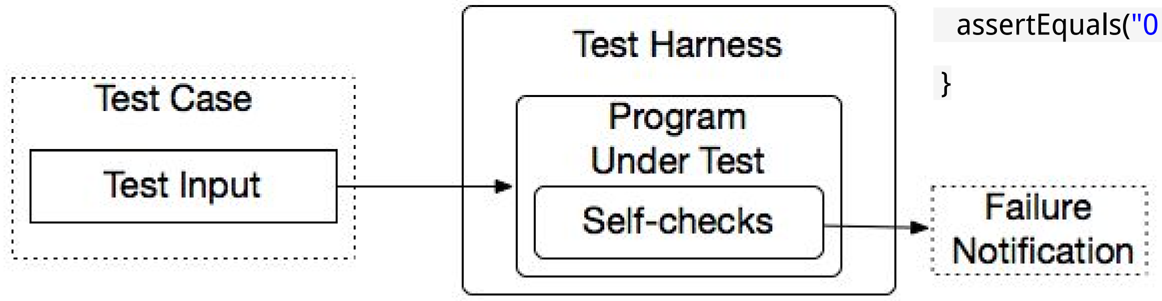


Any manually-created test case has a specified oracle (expected-value oracle: what is the expected value given this input?)

How can we extend this to multiple tests?

Self-Checks as Oracles

Rather than comparing actual values, use properties about results to judge sequences.



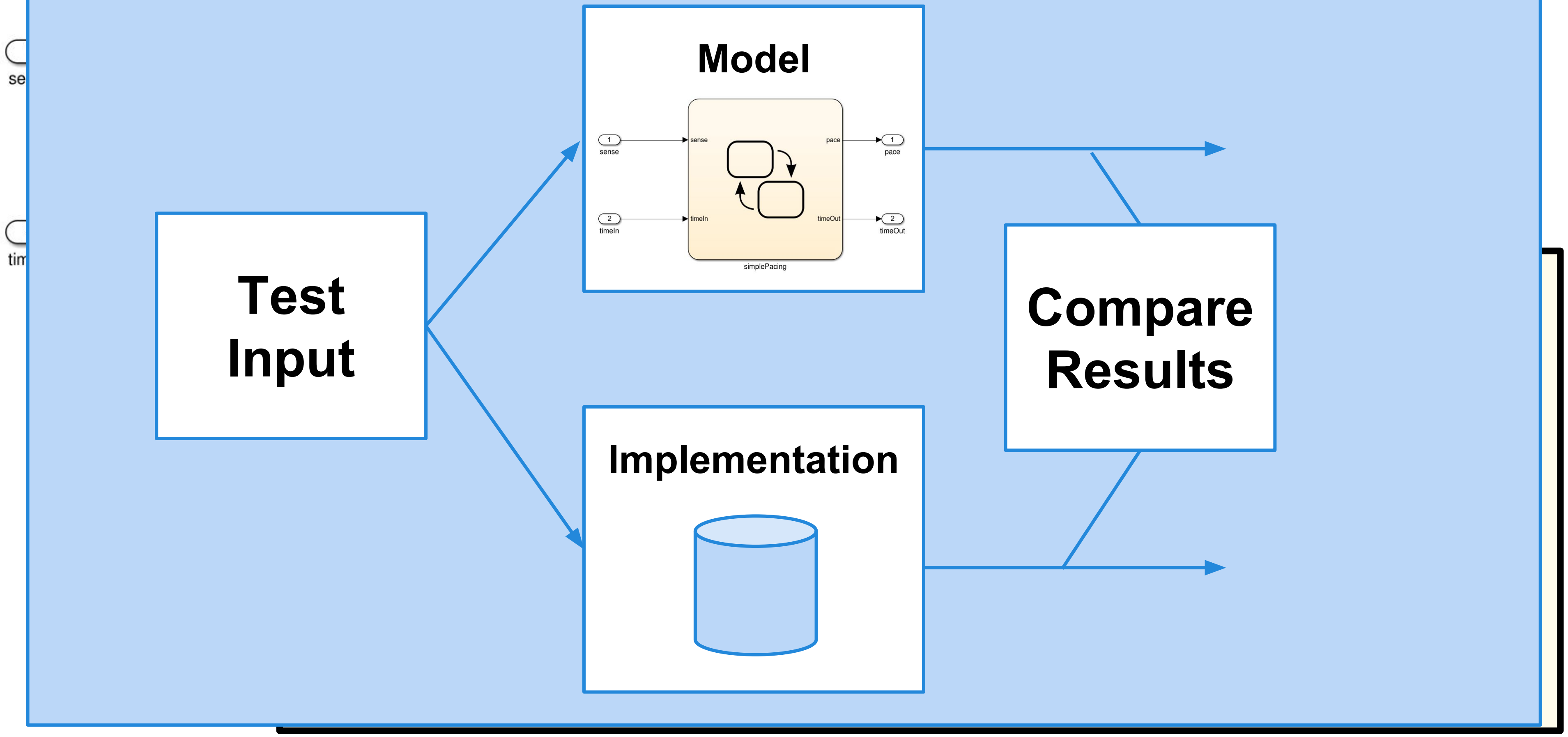
Take the form of assertions, contracts, and other logical properties.

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {
  // Tests
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

```
@Test
public void propertiesOfSort (String[] input) {
  // Tests
       String[] sorted = quickSort(input);
       assert(sorted.size >= 1, "This array can't be empty.")
}
```

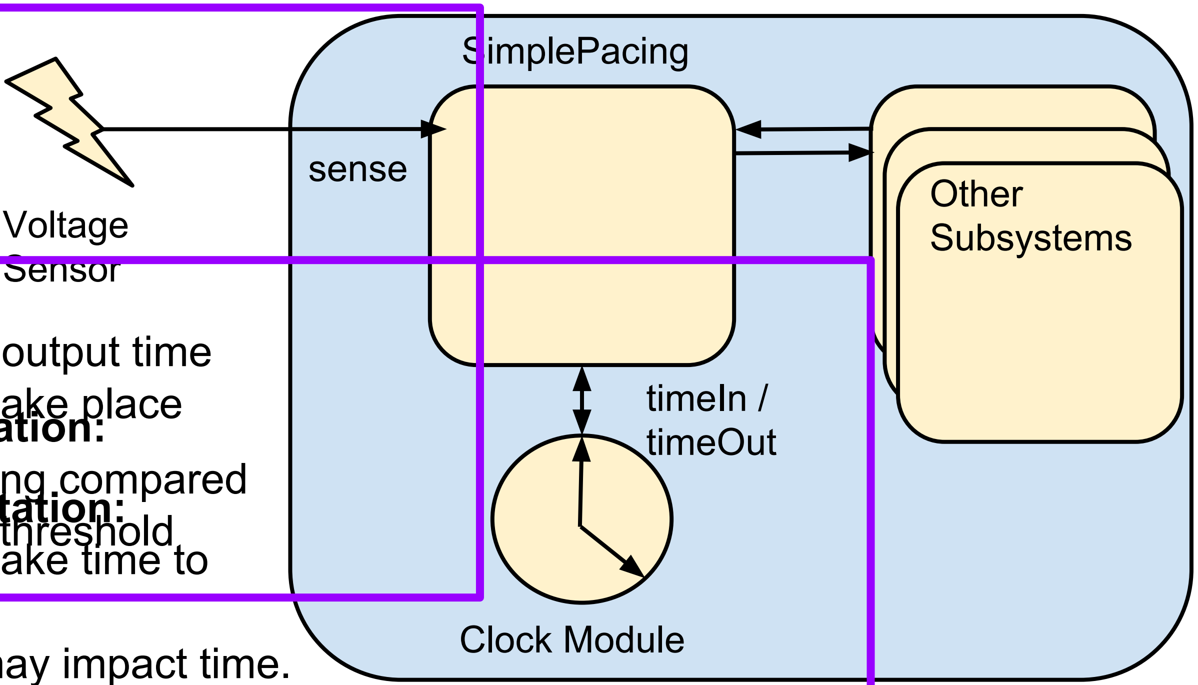
System Models as Oracles

Models could potentially serve as a "universal" test oracle



Problem: Abstraction

Models require abstraction. Useful for requirements analysis, but may not reflect operating conditions.



In the model:
- Binary input

In the implementation:
- Voltage reading compared to calculated threshold

In the model:
- input time = output time
- Operations take place instantly.

In the implementation:
- Operations take time to compute.
- Clock drift may impact time.

Derived Oracles

Derived Oracles

In the absence of a specified oracle, **oracles can be derived** from existing sources of information:

- Project Artifacts
  - Documentation
  - Existing tests
  - Other versions of the system
- Program Executions
  - Invariant detection
  - Specification mining

Often lowers costs significantly, but may suffer from accuracy issues.

Pseudo Oracles (N-Version Programming)

An alternate version of the program as an oracle.

Does `output(V1) = output(V2)`?

- Pseudo Oracle because we know the two don't agree, but don't know which is wrong or why.

Also called N-Version programming, where multiple designs are implemented, or same design is implemented by independent teams.

Genetic programming can use search (through genetic algorithms) to automatically produce multiple implementations.

Regression Testing

When changes are made to a system, rerun your tests. Any existing tests that passed previously should still pass.

An older version of your program can be the oracle.

- Do new features break working features?
- Do bug fixes break working features?
- If requirements have changed, you do NOT want the output to match for features related to the requirement.

Metamorphic Relations

If you have a set of test cases, you can generate partial oracles for follow-up tests by deriving **metamorphic relations** between tests.

- A metamorphic relation is a necessary property of a function:
  - A property of a sin function is that sin(x) = sin (pi - x).
  - Thus, sin(x) and sin(pi - x) have the same expected output.
- If these relationships are violated, then there is a bug.
- Can be an equation or more general properties specified between inputs.

Invariant Detection

Invariants (pre/post-conditions) can be specified as a form of test oracle. If they are not known in advance, there are algorithms that can detect them from program executions.

- Testers take a set of tests that the program is known to produce correct behavior for.
- Properties true of all observed executions are extracted for methods, loops, and conditional statements.

Model Inference

From system executions, we can derive a state machine model of system execution. As we observe more executions, we refine the model.

Major problem of both invariant and model detection: **Accuracy!**

- The more executions observed, the more accurate, but requires much more effort.
- What do we do about this?

Interlude:
Mutation Testing

Fault-Based Testing

Many testing techniques based on what we *think should happen*. We can also test based on *what we think could go wrong.*

Deliberately seed faults into a system, and see if you can distinguish the faulty system from the existing system.

Mutation Testing

1. Select *mutation operators* - code transformations that represent classes of faults that we are interested in.
2. Generate *mutants* by applying mutation operators to the program.
3. Execute tests against the program and mutants to *kill* mutants.

Mutation testing can judge adequacy of a test suite and can help tune metamorphic relations, invariants, and inferred models.

Mutation Operators

Operand Modifications

- Replace scalar with constant, alter constants in expressions (+1 changes to +X), negate booleans, etc.

Expression Modifications

- Replace arithmetic (+,-,*,/,%) operator, logical (&&, ||, !=, ==) operator, remove operator and an operand.

Statement Modifications

- Delete statement, switch case label, shift end of block.

Does Mutation Testing Make Sense?

Mutation testing is valid if seeded faults are *representative* of real faults.

Competent Programmer Hypothesis

- System under test is *close to* correct.

Coupling Effect

- Complex changes are a combination of smaller changes.

Mutant Quality

To be used in testing, mutants must be:

- Syntactically correct (*valid*)
- Plausible (*useful*)

Can a mutant be valid, but not useful?

Mutant Quality

Mutants that are valid, but not useful might remain live if:

- They are *equivalent* to the original program.
  - for(i=0; i < 10; i++)
  - for(i=0; i != 10; i++)
  - Identifying equivalency is NP-hard.
- Test suite is *inadequate* for that mutation.
  - (a <= b) and (a >= b) cannot be differentiated if a==b in the test case.

Mutation Coverage

Adequacy of the suite can be measured as:

$$\frac{(\text{\# mutants killed})}{(\text{total mutants})}$$

(Note - mutants can be equivalent when both the original and the mutant are wrong.)

Mutation can subsume other coverage:

- Statement: apply statement deletion to all statements.
- Branch: Replace all predicates with constants T/F.

Practical Considerations

Mutation testing is expensive.

- Must run *all* tests against *all* mutants.
- Many mutants typically generated.

If cost is an issue, can use “weak” mutation testing:

- Apply multiple mutation operators per mutant.
- Trade lower inference quality for fewer mutants.

Implicit Oracles

Implicit Oracles

Implicit oracles require no domain knowledge or specification, but instead can be applied to check properties that are expected of any runnable program.

Implicit oracles often detect particular anomalies, such as network irregularities or deadlock. These are faults that do not require expected output to detect.

Uses of Implicit Oracles

Implicit oracles can be built to detect:

- Concurrency Issues
  - Deadlock, livelock, and race detection
- Violations of properties related to non-functional attributes of the system
  - Performance properties
  - Robustness
  - Memory access and leaks

Fuzzing

Fuzzing is a way to find implicit anomalies.

- Generate random (or fuzz inputs).
- Attack the system with these inputs.
    - Generation and attacks guided by “attack profiles” that reflect certain malicious use scenarios.
- Report anomalies with the test sequence that caused them.

Commonly used to detect security vulnerabilities, such as buffer overruns, memory leaks, unhandled exceptions, and denial of service.

Human Oracles

The Human Oracle

- If no automation is possible or no specification exists, a human can always judge output manually.
- Not ideal, but surprisingly common in practice.

Handling the Lack of Oracles

Even if there is no oracle, there are techniques that can reduce the *human oracle cost* through:

- Quantitative reduction in the amount of work the tester has to do for the same amount of fault-detection potential.
- Qualitative increase in the ease of evaluating testing results.

Quantitative Cost Reduction

Test suites can be unnecessarily large:

- Tests that cover too few testing goals or scenarios.
- Tests that are unnecessarily long, with redundant method calls.

Human oracle cost can be reduced by cutting out either of these.

- Test suite reduction techniques cut tests that cover redundant code structure or do not penetrate deeply into the code.
- Test case reduction techniques attempt to remove unnecessary test steps.

Qualitative Cost Reduction

Not all test cases are equally understandable by human testers. Automated test generation often produces test inputs that do not match the expected usage of a program, and humans have trouble judging the results of such tests.

Some test generation approaches allow the seeding of human knowledge or use usage profiles to help generate input.

Crowdsourcing the Oracle

Recent development - outsource the oracle problem to many different human oracles.

Several services exist for this now - Amazon Mechanical Turk, Mob4Hire, MobTest, uTest.

Users cannot be expected to have much domain knowledge, so understandability of test inputs and documentation are very important.

Placing the Oracles

	Cost(T/O)	Accuracy	Completeness
Manual Specification	L/H	H	L
Behavioral Model	H/L	M-H	H
Self-Checks	L/M	H	M
N-Version Programming	L-H/L	L-H	M-H
Metamorphic Testing	L/M	H	M
Invariant Detection	L/L	L-H	M
Implicit Oracles	L/L	H	L

We Have Learned

- Test Oracles judge the correctness of sequences of stimuli and observations.
- Oracles can be:
  - specified (expected values, models, assertions)
  - derived from correct executions or project artifacts
  - built to detect implicit properties
  - humans asked to check results
- Mutation testing can help improve testing efforts by using fake faults to learn about how our system works.

Next Time

- When to stop testing.
  - Statistical testing and some wrap-up testing topics.
  - Reading:
    - Sommerville, ch. 11
- Any homework questions?
- No office hours on 11/24
  - Makeup hours 11/23, 3:30 - 4:30 PM