CS 465 — Software Testing, Quality Assurance, and Maintenance

Kevin James

Winter 2015

Contents

1 Things that Break

There are many things which we must be worried about when deisgning stable software. In particular, we will concern ourselves primarily with

- race conditions
- segmentation faults, crashes, infinite loops
- wrong output, unwanted side-effects
- wrong API, incompatible submodules
- bad system behaviour, bad security, wrong specs, wrong output
- non-functional properties, memory leaks, bad performance
- regressions

There are many steps we can take to fix these issues:

- in-house testing
- validatoin suires for plugins
- code review
- better design
- fewer features
- defensive programming, especially for plugins

Software never completely works, so our goal should be to make it "good enough". How do we do this (ie. how do we cope)? Generally, we use disclaimers, patches, defensive programming, and backups of user data.

1.1 Testing Methods

We can test our systems by

1. compiling it
2. running it on single input
3. running it on many inputs
4. running it on representative inputs
5. running it on all inputs (statically)
6. integration testing
7. regression testing
8. non-function testing

1.2 Terms

Validation is the act of evaluating software prior to release to ensure compliance. We do **verification** by ensuring that products of phase x fill the requirements of phase $x + 1$. We consider **faults** to be static defects, **errors** to be incorrect internal states, and **failures** to be internally incorrect behaviour. All failures must be caused by errors, but not all errors generate failures. **Testing** is the act of evaluating an application by observing execution and **debugging** is the act of finding (fixing) a fault given a failure.

Automation makes testing better and less boring. This also allows you to run tests faster and generate reports, eg. coverage.

List of tools:

- unit test libraries
- clang, llvm
- valgrind
- daikon
- randoop
- Java Path Finder
- cppcheck
- FindBugs
- splint
- coverity
- CS SAL
- iComment

2 Faults, Errors, and Failures

Faults are static defects in the software (ie. lines of code that are wrong). **Errors** occur when the system has the incorrect state, though this does not necessarily have to be observed. **Failures** are externally incorrect behaviour which has been observed by a user.

Within the category of faults, there exist both **design faults** and **mechanical faults**. Design faults occur when the system was designed in such as way as to cause wrong behaviour, do the wrong thing, etc. Mechanical faults occur when unexpected input causes a change in behaviour (rendering it incorrect).

The **state** of a system is all the variables in the program; this includes input, output, return values, temporary variables, iterators/other temporary variables, and the program counter.

2.1 RIP Model

There are three necessary conditions for a failure:

- Reachability: the PC must reach the part of the program with the fault
- Infection: there must exist a state problem after executing the fault
- Propogation: the state issue must propogate to a visiable location, ie. cause incorrect output

How do we deal with an imperfect world? Fault avoidance (better design, better languages), fault detection (testing), and fault tolerance (redundancy, isolation).

3 Metrics

Code coverage does not provide and accurate representation of test quality, ie. a line which is run but has a bug given some input. Since we do not necessarily have this/these input(s) in our test suire, this code would be marked as covered despite being incorrect.

Outputs of a function may be a good metric if we run a function for all inputs.

Input space coverage is effective if done well; random testing is ineffective, but can prove effective when directed. This involves a test case for each subset of imput space.

Logic coverage can also be effective when done well; if we logically ensure we test all cases, we can prove correctness.

3.1 Static

Static testing involves not actually running the code. You can compile it, use semantic verification (to determine unreachable code or always-thrown unhandled exceptions), do full functional verification, and do code reviews/inspections.

3.2 Dynamic

Dynamic testing is the usual means of testing: you simply run and observe the code.

Test sets are groups of **test cases**. A test case is a single-input, single-output test with respect to some code. We can have some difficulties in implementing test cases; slow systems, difficulty in setting tests up, hard-to-parse results (hardware, non-observable, etc).

We avoid "bad words" with respect to coverage such as "complete testing", "exhaustive testing", and "full coverage". It is must better to find a reduced space and cover *that* space entirely.

A **test requirement** is a specific element of a software artifact that a test case must satisfy or cover. A test set may satisfy a test requirement / set of test requirements. Infeasible test requirement (ie. "cover all statements" when there exist unreachable statements) may make full requirement coverage impossible.

Criteria subsumption: coverage criterion c_1 subsumes c_2 if and only if every test set that satisfies c_1 also satisfies c_2.

How do we evaluate coverage criteria? We base them on

- the difficulty of generating test requirements,
- the difficulty of generating tests, and
- their effectiveness in determining faults

3.3 Test Paths

A **path** through a directed program graph must have length greater than or equal to zero. **Subpaths** are legal subsequences of a path. A **test path** is a path that starts at n_0 and ends at n_f. The execution of a test case creates a test path. Non-deterministic behaviour could cause each test case to generate multiple (infinite?) test paths.

Causes of non-determinism: scheduling, user input, memory allocation.

To find test cases for a given test path, there are some complicated tools which only work in simple cases; otherwise, this must be done manually.

4 Syntax and Semantics

A statement is **syntactically** reachable if there exists a path from n_0 to that node. It is **semantically** reachable if some set of program inputs can lead us there.

We denote the portion of graph G which is syntactically reachable from χ as $\text{reach}_G(\chi)$. One of the most useful of these graphs is $\text{reach}_G(n_0)$, the set of reachable blocks in that program.

A single-entry, single-exit (SESE) graph has one input and one output. We will be mostly discussing these in this class.

Black-box testing is any test cases that are written without looking at the implementation. We, in effect, do not know how the code works behind the scenes. **White-box testing** involves test based on the system design / implementation.

Given a set of code such as

```
void foo(int x) {
  if (x == 5) {
    print "5";
  } else {
    print "not 5";
  }

  if (x == 5) {
    print "5";
  } else {
```

```
    print "not 5";
  }
}
```

an **infeasible test path** is one that prints "5" and then "not 5".

4.1 Graphs and Methods

Each method induces a graph. Nodes correspond to statements and edges correspond to the successor relation. Note that graphs created in this way must be **directed**, ie. edges must include a directionalit / a direction upon which we must travel to follow this edge.

Node coverage is one of many types of coverage. This is also commonly refered to as statement coverage, since this covering this form of graph corresponds exactly with covering the related nodes. To cosider a program as covered under this methodology, there must exist a set of test cases which causes the execution of every statement.

Within a test set, for each test case we execute the program on that test case's input. After doing so, we get the test case's output and validate it as well as a trace of the executed statements (ie. a test path).

Benefits of node coverage

- easy-to-understand metric
- easy-to-compute metric
- shows unused code
- first glance at "testedness" of system

Disadvantages of node coverage

- false sense of security when "100%" coverage is reached.
- promote useless tests
- does not help detect duplicate code

We can also have **edge coverage**, which is similar to node coverage but for branches instead of statements. If our program has a loop, we can satisfy node coverage by running the loop a single time. To satisfy edge coverage, we must actually run the loop multiple times. These cases tend to be somewhat contrived, so in practice we consider both forms of coverage to be equivalent.

Beyond node and edge coverage, we have **complete path coverage**, which covers paths of all lengths. Note that if these is at least one loop in the program, this creates an infinite number of test requirements.

A path is **simple** if no node appears more than once in the path, except that the first and last nodes may (optionally) be the same. Simple paths have the following properties:

- no internal loops
- can bound their length

- can create any path by composing simple paths
- (too) many simple paths exist

A **prime path** is a simple path that is not a proper subpath of any other path. **Prime path coverage**, then, requires we simply test all prime paths.

A **round trip path** is a simple path of non-zero length that starts and ends at the same nodes. **Simple Round-Trip coverage** contains at least on round-trip path for each reachable node that begains and ends a round-trip path.

A graph G is **connected** if each node is reachable from an initial node. An edge e is a **bridge** for a graph G if G is connected but removing e results in a disconnected graph. Note that since graphs are directed in this class, a connected graph precludes the existance of multiple starting nodes.

4.1.1 Control Flow Graphs

Basic Blocks can be formed within graphs if we group together statements which always execute together. These blocks must have a single entry point and a single exit point.

5 Race Conditions

A race condition occurs when some shared access is written to by two different processes with no guarantee of order. This often occurs when there is no lock on a resource when it is to be written to and then read to.

There are several tools which can help us detect race conditions:

- Helgrind
- lockdep
- Oracle Solaris Studio thread analyzer
- Coverity thread analyzer
- Intel Inspector XE 2011

Sometimes, eliminating race conditions does not eliminate bugs. See below, where we lock resources when we use them, but can still have a race condition due to the stale state between locks.

```
#include <iostream>
#include <mutex>
#include <thread>

int counter = 0;
std::mutex m;
```

```
void func() {
  int tmp;

  m.lock();
  tmp = counter;
  m.unlock();
  tmp++;
  m.lock();
  count = tmp;
  m.unlock();
}

int main() {
  std::thread t1(func);
  std::thread t2(func);

  t1.join();
  t2.join();
  std::cout << counter << "\n";
  return 0;
}
```

As we've seen, removing race conditions does not solve all problems. Try to:

- run your code multiple times
- add noise (sleep, increase system load, etc.)
- use tools
- force scheduling (Java PathFinder)
- static approaches (lock-set, happens-before, state-of-the-art lock techniques)

5.1 Bad Lock Usage

Locks and unlocks must be paired, otherwise Bad Things™ could happen.

```
lock();
if (something):
  do_something()
  return
unlock()
```

6 Finding Bugs

Since our goal is to find as many bugs as possible (so we can fix them, of course...), we need to determine how to deduce whether something is a bug.

- fund ocntradicitons and/or errors
- find deviance from normal behaviour

7 Grammars

Grammars allow us to generate inputs directly: correct inputs which fulfil the grammar and incorrect inputs which do not. We can also use mutation testing, by modifying the grammar or the program and finding inputs which exploit differences. We refer to these modified programs as mutants and search for ways to "kill the mutant".

Ground strings are strings within our grammar. **Mutation operators** are modificatios we can utilize to change a string. A **mutant** is a ground string upon which we've applied a mutation operator to generate something which appears close to being within our grammar.

For a mutant m of ground string m_0, we define test case t as **killing** m if running t on m produces different output than running t on m_0. Potential criteria: mutation coverage, mutation operator coverage, and mutation production coverage.

Mutants are programs, not tests.

The goals of mutation testing: mimic typical mistakes and encode knowledge about effective test practices. The goal is to kill mutants, but the desired side effect is the test cases which kill them. For weak mutation coverage, the broken output is not required.