Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 2 — January 7/9, 2017

Patrick Lam *version 1*

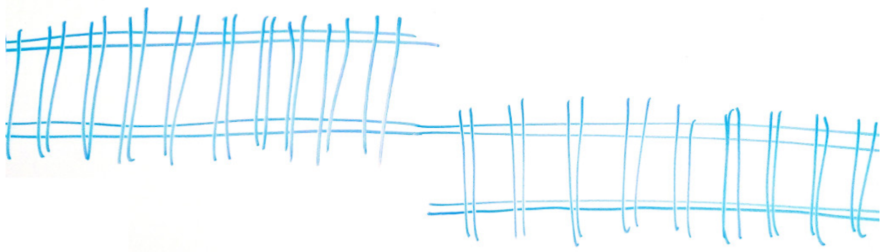
Faults, Errors, and Failures

For this course, we are going to define the following terminology.

- **Fault** (also known as a bug): A static defect in software—incorrect lines of code.
- **Error**: An incorrect internal state—not necessarily observed yet.
- **Failure**: External, incorrect behaviour with respect to the expected behaviour—must be visible (e.g. EPIC FAIL).

These terms are not used consistently in the literature. Don't get stuck on memorizing them.

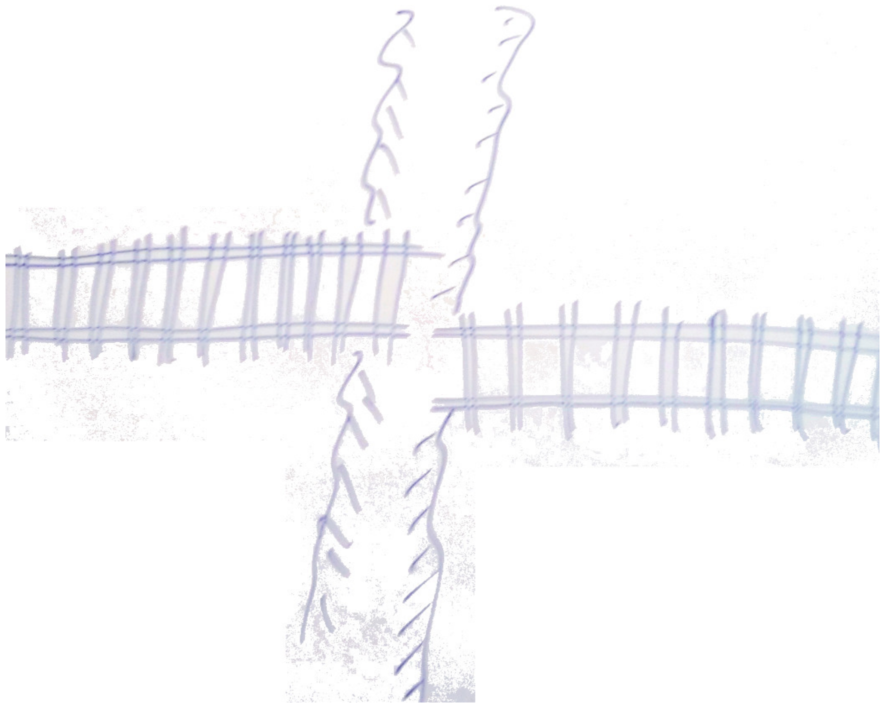
Motivating Example. Here's a train-tracks analogy.



(all railroad pictures inspired by: Bernd Bruegge & Allen H. Dutoit, *Object Oriented Software Engineering: Using UML, Patterns and Java.*)

Is it a failure? An error? A fault? Clearly, it's not right. But no failure has occurred yet; there is no behaviour. I'd also say that nothing analogous to execution has occurred yet either. If there was a train on the tracks, pre-derailment, then there would be an error. That picture most closely corresponds to a fault.

Perhaps it was caused by mechanical stresses.



Or maybe it was caused by poor design.

Software-related Example. Let's get back to software and consider this code:

```
public static numZero(int[] x) {
  // effects: if x is null, throw NullPointerException
  //          otherwise, return number of occurrences of 0 in x.
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    // program point (*)
    if (x[i] == 0) count++;
  }
  return count;
}
```

As we saw, it has a fault (independent of whether it is executed or not): it's supposed to return the number of 0s, but it doesn't always do so. We define the state for this method to be the variables `x`, `i`, `count`, and the Program Counter (PC).

Feeding this `numZero` the input {`2, 7, 0`} shows a wrong state.

The **wrong state** is as follows: `x = `{`2, 7, 0`}`, i = `**`1`**`, count = 0, PC = (*)`, on the first time around the loop.

The **expected state** is: `x = `{`2, 7, 0`}`, i = `**`0`**`, count = 0, PC = (*)`

However, running `numZero` on {`2, 7, 0`} executes the fault and causes a (transient) error state, but doesn't result in a failure, as the output value `count` is 1 as expected.

On the other hand, running `numZero` on {`0, 2, 7`} causes an error state with `count = 0` on return, hence leading to a failure.

RIP Fault Model

To get from a fault to a failure:

1. Fault must be *reachable*;
2. Program state subsequent to reaching fault must be incorrect: *infection*; and
3. Infected state must *propagate* to output to cause a visible failure.

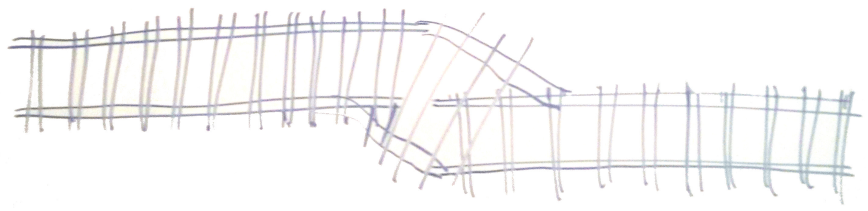
Applications of the RIP model: automatic generation of test data, mutation testing.

Dealing with Faults, Errors and Failures

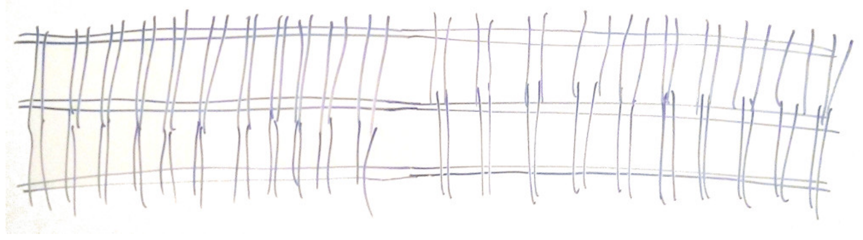
Three strategies for dealing with faults are avoidance, detection and tolerance. Or, you can just try to declare that the fault is not a bug, if the specification is ambiguous.

Fault Avoidance. Certain faults can just be avoided by not programming in vulnerable languages; buffer overflows, for instance, are impossible in Java. Better system design can also help avoid faults, for instance by making an error state unreachable.

Fault Detection. Testing (construed broadly) is the primary means of fault detection. Software verification also qualifies. Once you have detected a fault, if it is economically viable, you might repair it:



Fault Tolerance. You are never going to remove all of the bugs, and some errors arise from conditions beyond your control (such as hardware faults). It's worthwhile to tolerate faults too. Strategies include redundancy and isolation. An example of redundancy is provisioning extra hardware in case a server goes down. Isolation includes things as simple as checking preconditions.



Testing vs Debugging

Recall from last time:

Testing: evaluating software by observing its execution.
Debugging: finding (and fixing) a fault given a failure.

I said that you really need to automate your tests. But even so, testing is still hard: only certain inputs expose the fault in the form of a failure. As you've experienced, debugging is hard too: you have the failure, but you have to find the fault.

Contrived example. Consider the following code:

```
if (x - 100 <= 0)
  if (y - 100 <= 0)
    if (x + y - 200 == 0)
      crash();
```

Only one input, `x = 100` and `y = 100`, will trigger the crash. If you're just going to do a random brute-force search over all 32-bit integers, you are never going to find the crash.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 3 — January 7/9, 2019

Patrick Lam *version 0*

Exercise: findLast. Here's a faulty program.

```
1     static public int findLast(int[] x, int y) {
2         /* bug: loop test should be i >= 0 */
3         for (int i = x.length - 1; i > 0; i--) {
4             if (x[i] == y) {
5                 return i;
6             }
7         }
8         return -1;
9     }
```

You might expect to be asked a question like this:

(a) Identify the fault, and fix it.

(b) If possible, identify a test case that does not execute the fault.

(c) If possible, identify a test case that executes the fault, but does not result in an error state.

(d) If possible, identify a test case that results in an error, but not a failure. (Hint: program counter)

(e) For the given test case, identify the first error state. Be sure to describe the complete state.

I asked you to work on that question in class. Here are some answers:

(a) The loop condition must include index 0: `i >= 0`.

(b) This is a bit of a trick question. To avoid the loop condition, you must not enter the loop body. The only way to do that is to pass in `x = null`. You should also state, for instance, `y = 3`. The expected output is a `NullPointerException`, which is also the actual output.

(c) Inputs where `y` appears in the second or later position execute the fault but do not result in the error state; nor do inputs where `x` is an empty array. (There may be other inputs as well.) So, a concrete input: `x = {2, 3, 5}; y = 3`. Expected output = actual output = 1.

(d) One error input that does not lead to a failure is where `y` does not occur in `x`. That results in an incorrect PC after the final executed iteration of the loop.

(e) After running line 6 with `i = 1`, the decrement occurs, followed by the evaluation of `i > 0`, causing the PC to exit the loop (statement 8) instead of returning to statement 4. The faulty state is `x = {2, 3, 5}; y = 3; i = 0; PC = 8`, while correct state would be `PC = 4`.

Someone asked about distinguishing errors from failures. These questions were about failures at the method level, and so a wrong return value would be a failure when we're asking about methods. In the context of a bigger program, that return value might not be visible to the user, and so it might not constitute a failure, just an error.

Line Intersections

We then talked about different ways of validating a test suite. Consider the following Python code, found by Michael Thiessen on stackoverflow (`http://stackoverflow.com/questions/306316/determine-if-two-rectangles-overlap-each-other`).

```
class LineSegment:
    def __init__(self, x1, x2):
        self.x1 = x1; self.x2 = x2;

def intersect(a, b):
    return (a.x1 < b.x2) & (a.x2 > b.x1);
```

We could construct test suites that:

- execute every statement in `intersect` (statement coverage). Well, that's not very useful; any old test case will do that. There are no branches, so what we'll call edge coverage doesn't help either.
- feed random inputs to `intersect`; unfortunately, interesting behaviours are not random, so it won't help much in general.
- check all outputs of `intersect` (i.e. a test case with lines that intersect and one with lines that don't intersect): we're getting somewhere—that will certify that the method works in some cases, but it's easy to think of situations that we missed.
- check different values of clauses `a.x1 < b.x2` and `a.x2 > b.x1` (logic coverage)—better than the above coverage criteria, but still misses interesting behaviours;
- analyze possible inputs and cover all interesting combinations of inputs (input space coverage)—can create an exhaustive test suite, if your analysis is sound.

Let's try to prove correctness of `intersect`. There's an old saying about testing—supposedly, it can only find the presence of bugs, not their absence. This is not completely true, especially if you have reliable software that automatically constructs an exhaustive test suite. But that is beyond the state of the practice in 2015, for the most part.

Inputs to `intersect`. There are essentially four inputs to this function. Rename them $aAbB$, for `a.x1`, `a.x2`, etc.

- Let's first assume that all points are distinct. We should make a note to ourselves to check violations of this, as well: we may have $a = A, a = b, a = B$, and symmetrically for B: $b = a, b = A, b = B$.
- For the purpose of the analysis, let's assume that $a < b$; when constructing test cases, we can swap a and b around. That's why there are duplicate assert statements below.
- Without loss of generality, we can assume that $a < A$ and $b < B$. (We ought to update the constructor if we want to make that assumption.)

With these assumptions, we have to test the three possible permutations $aAbB$, $abAB$, and $abBA$. It is simple to construct test cases for these permutations, using Python's unittest framework:

```
  def test_aAbB(self):
    a = LineSegment(0,2)
    b = LineSegment(3,7)
    self.assertFalse(intersect(a,b))
    self.assertFalse(intersect(b,a))

def test_abAB(self):
    a = LineSegment(0,4)
    b = LineSegment(3,7)
    self.assertTrue(intersect(a,b))
    self.assertTrue(intersect(b,a))

def test_abBA(self):
    a = LineSegment(0,4)
    b = LineSegment(1,2)
    self.assertTrue(intersect(a,b))
    self.assertTrue(intersect(b,a))
```

Those test cases pass. However, if you construct test cases for equality (as I've committed to the repository), you see that the given `intersect` function fails on line segments that intersect only at a point. Replacing `<` with `<=` and `>` with `>=` fixes the code.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 4 — January 14, 2019

Patrick Lam *version 1*

Last time. We saw two examples: `findLast`, where we did an example about identifying faults, errors, and failures, and finding distinguishing test cases; and the line intersection example, where we informally reasoned about the correctness of that algorithm and demonstrated it using testing. The reasoning is a lot more ad-hoc than what we've seen in algorithms courses.

About Testing

We can look at testing statically or dynamically.

Static Testing (ahead-of-time): this includes static analysis, which is typically automated and runs at compile time (or, say, nightly), as well human-driven static testing—typically code review.

Dynamic Testing (at run-time): observe program behaviour by executing it; includes black-box testing (not looking at code) and white-box testing (looking at code to develop tests).

Usually the word "testing" means *dynamic testing.*

Naughty words. People like to talk about "complete testing", "exhaustive testing", and "full coverage". However, for many systems, the number of potential inputs is infinite. It's therefore impossible to completely test a nontrivial system, i.e. run it on all possible inputs. There are both practical limitations (time and cost) and theoretical limitations (i.e. the halting problem).

The first part of the course is about defining test suites and aims to give you tools to answer the question "when should I stop testing?" Reiterating the syllabus, I might stop:

- *When I run out of time.* Open-ended exploratory testing; automatic input generation.
- *When I am close enough to being exhaustive.* Coverage of: (enough) statements, branches, program states, use cases.

I'll again put in a shout-out to mutation as being another way to validate whether one's test suite is good enough.

Test cases

As we've seen in the last two lectures, a *test case* contains:

- what you feed to software; and
- what the software should output in response.

Our test cases have been easy to generate so far, but that's not always the case.

Definition 1 Observability *is how easy it is to observe the system's behaviour, e.g. its outputs, effects on the environment, hardware and software.*

Definition 2 Controlability *is how easy it is to provide the system with needed inputs and to get the system into the right state.*

Anatomy of a Test Case

Consider testing a cellphone from the "off" state:

$$\underset{\text{prefix values}}{\langle \text{ on } \rangle} \quad \underset{\text{test case values}}{1\ 519\ 888\ 4567} \quad \underbrace{\underset{\text{verification values}}{\langle \text{ talk } \rangle} \quad \underset{\text{exit codes}}{\langle \text{ end } \rangle}}_{\text{postfix values}}$$

Definition 3

- Test Case Values*: input values necessary to complete some execution of the software. (often called the test case itself)*
- Expected Results*: result to be produced iff program satisfies intended behaviour on a test case.*
- Prefix Values*: inputs to prepare software for test case values.*
- Postfix Values*: inputs for software after test case values;*
  - verification values*: inputs to show results of test case values;*
  - exit commands*: inputs to terminate program or to return it to initial state.*

Definition 4

- Test Case*: test case values, expected results, prefix values, and postfix values necessary to evaluate software under test.*
- Test Set*: set of test cases.*
- Executable Test Script*: test case prepared in a form to be executable automatically and which generates a report.*

On Coverage

Ideally, we'd run the program on the whole input space and find bugs. Unfortunately, such a plan is usually infeasible: there are too many potential inputs.

Key Idea: Coverage. Find a reduced space and cover that space.

We hope that covering the reduced space is going to be more exhaustive than arbitrarily creating test cases. It at least tells us when we can plausibly stop testing.

The following definition helps us evaluate coverage.

Definition 5 *A* test requirement *is a specific element of a (software) artifact that a test case must satisfy or cover.*

We write TR for a set of test requirements; a test set may cover a set of TRs.

Two software examples:

- cover all decisions in a program (branch coverage); each decision gives two test requirements: branch is true; branch is false.
- each method must be called at least once; each method gives one test requirement.

Infeasible Test Requirements. Sometimes, no test case will satisfy a test requirement. For instance, dead code can make statement coverage infeasible, e.g.:

```
if (false)
  unreachableCall();
```

or, a real example from the Linux kernel:

```
while (0)
  {local_irq_disable();}
```

Hence, a criterion which says "test every statement" is going to be infeasible for many programs.

Quantifying Coverage. How good is a test set? It's great if it covers everything, but sometimes that's impossible. We can instead assign a number.

Definition 6 *(Coverage Level). Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.*

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 5 — January 14/16, 2019

Patrick Lam *version 2*

Last time. We spent most of the time talking about the distinction between static and dynamic analysis of code. Static analysis finds all the things (perhaps too many), while dynamic analysis tells you exactly what happens on a particular execution (but you have to have the right inputs). We also looked at a JUnit test case (setup/teardown/establishing state/checking system under test actions); the test case we looked at was at `https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/io/MoreFilesTest.java`. Finally, we introduced the notion of coverage and test requirements a bit more formally.

Exploratory Testing

Exploratory testing is usually (but not always) carried out by dedicated testers. In that sense, it's somewhat different from the other testing activities in this course, which are more developer-focussed—our usual goal is learning, as developers, how to deploy better automated test suites for our software. Hallway usability testing, though, is an application of exploratory testing. Furthermore, the dedicated QA function is important, and we should learn about how it works.

Resources. James Bach has a shorter and a longer introduction to exploratory testing:
- `http://www.satisfice.com/articles/what_is_et.shtml`
- `http://www.satisfice.com/articles/et-article.pdf`

There is an exhaustive set of notes on exploratory testing by Cem Kaner:
- `http://www.kaner.com/pdfs/QAIExploring.pdf`

> "Exploratory testing is simultaneous learning, test design, and test execution."

Contrast this to scripted testing: test design happens ahead of time and then test execution happens (repeatedly) throughout the product's development cycle. When we think of dedicated QA teams, we think they are manually executing scripted tests. In 2019, that is not an effective use of staff.

There is a continuum between scripted testing and exploratory testing. Good exploratory testing may use prepared scripts for certain tasks.

Scenarios where Exploratory Testing Excels. (from Bach's article)

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process. Exploratory testing should not be randomly bumbling around (we can call that "ad hoc testing")—the random approach finds bugs but isn't the most efficient at giving you an idea of how well the software works.

- Start with a charter for your testing activity, e.g. "Explore and analyze the product elements of the software." These charters should be somewhat ambiguous.
- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Exploratory testing shouldn't produce an exhaustive set of notes. Good testers will be able to reproduce the bugs that they encounter during their testing from brief notes. Taking full notes takes too long.

The output from exploratory testing is at least a set of bug reports. It may also include test notes, which include overall impressions and a summary of the test strategy/thought process. Artifacts such as test data or test materials are also both inputs and outputs from exploratory testing.

Primary vs contributing tasks. One way to classify tasks that software can do (or, in other words, its features) is *primary* vs *contributing.* A *primary* task is core functionality of the system; it's something that you would say "You Had One Job!" about. As examples, text editors must be able to load text files, add text, and save the text files. On the other hand, *contributing* tasks are secondary. A macro system for a text editor would be a contributing task. Being able to read email in your editor is definitely a contributing task. Sometimes it's not black-and-white. Spell-check can go either way.

Example. I recommend reading the example by James Bach in "et-article.pdf" about the photo editing program ("ET in Action"). He describes how he used ET to evaluate software for Windows compatibility and found critical defects.

In-class exercise: Exploratory testing of WaterlooWorks.

We will try out exploratory testing with WaterlooWorks. I believe that everyone should have access to the system, although for some of you there may be no jobs visible right now.

The charter will be "Explore the overall functionality of WaterlooWorks". Summarize in one or two sentences what the purpose of WaterlooWorks is. Identify the tasks that WaterlooWorks should be able to do and classify them as primary or contributing. Identify areas of potential instability. Test each function and record results (bugs).

Of course, don't do things that have actual effects. Usually, testers would have access to a development server and could test those areas more aggressively. But we are working with production systems here.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 6 — January 16, 2019

Patrick Lam *version 1*

Last time. We discussed benefits of exploratory testing and a sketch of how to do it. We also started to do a case study of WaterlooWorks.

Source Code Coverage Criteria

We alluded to statement coverage and branch coverage. It is possible to evaluate these criteria directly on the source code, but better to use a sensible intermediate representation. The fundamental graph for source code is the *Control-Flow Graph* (CFG), which originates from compilers.

- CFG nodes: a node represents zero or more statements;
- CFG edges: an edge (s_1, s_2) indicates that s_1 may be followed by s_2 in an execution.

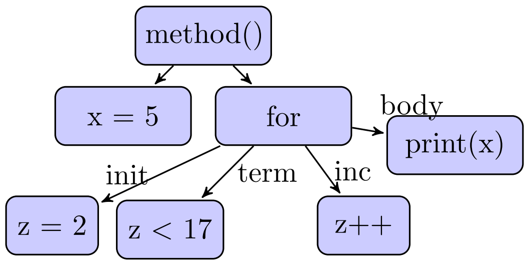
Example. Consider the following code.

```
x = 5;
for (z = 2; z < 17; z++)
  print(x);
```

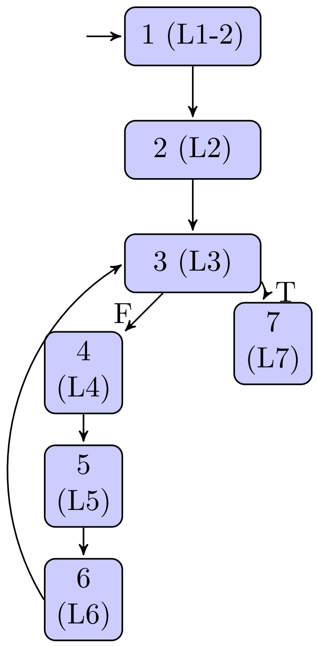
Recall the steps in compilation:

- lexing: input = stream of characters, output = stream of tokens (if, while, strings)
- parsing: input = stream of tokens, output = concrete syntax tree
- construction of Abstract Syntax Tree (AST): cleans up the concrete syntax tree
- conversion to Control Flow Graph: input = AST, output = CFG
- optimizations: input = CFG, output = CFG
- convert to bytecode/machine code: input = CFG, output = bytecode/machine code

The Abstract Syntax Tree corresponding to the example code might look like this:



From ASTs to CFGs. We can convert the Abstract Syntax Tree into the following Control Flow Graph (and we'll see how to do so in Lecture 7).



From CFG to low-level code. And we can convert the CFG into the following low-level code:

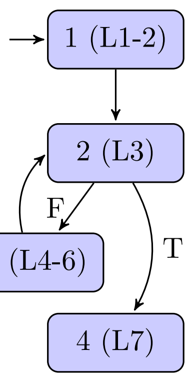
```
1       x = 5
2       z = 2
3   q0: if (z < 17) goto q1
4       z = z + 1
5       print (x)
6       goto q0
7   q1: nop
```

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 7 — January 21, 2019

Patrick Lam *version 1*

Basic Blocks. We can simplify a CFG by grouping together statements which always execute together (in sequential programs):



We use the following definition:

Definition 1 *A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.*

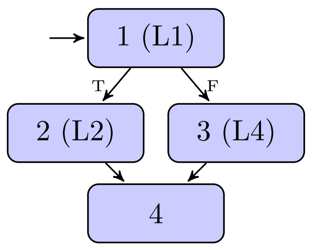
We are usually interested in forming maximal basic blocks. Note that a basic block may have multiple successors. However, there may not be any jumps into the middle of a basic block (which is why statement `l0` has its own basic block.)

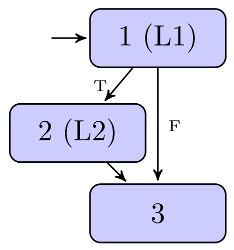
Some Examples

We'll now see how to construct control-flow graph fragments for various program constructs.

if statements: One can put the conditions (and hence uses) on the control-flow edges, rather than in the `if` node. I prefer putting the condition in the node.

```
1  if (z < 17)
2    print (x);
3  else
4    print (y);
```



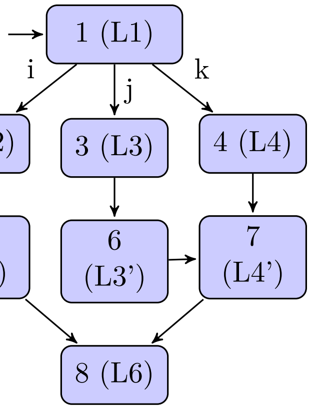


```
1  if (z < 17)
2    print(x);
```

Short-circuit `if` evaluation is more complicated; I recommend working it out yourself.

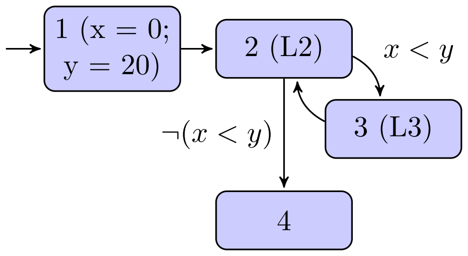
case / switch statements:

```
switch (n) {
  case `I': ...; break;
  case `J': ...; // fall thru
  case `K': ...; break;
}
// ...
```



while statements:

```
x = 0; y = 20;
while (x < y) {
  x ++; y --;
}
```



Note that arbitrarily complicated structures may occur inside the loop body.

for statements:

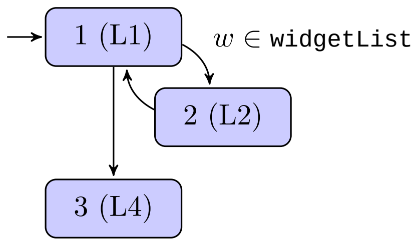
```
for (int i = 0; i < 57; i++) {
  if (i % 3 == 0) {
    print (i);
  }
}
```

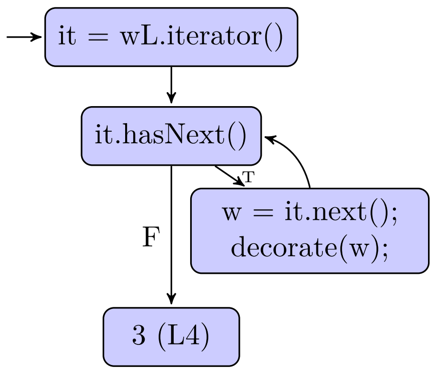
(an exercise for the reader;
we saw one earlier!)

This example uses Java's enhanced for loops, which iterates over all of the elements in the `widgetList`:

```
for (Widget w : widgetList) {
  decorate(w);
}
```

I will accept the simplified CFG or the more useful one on the right:





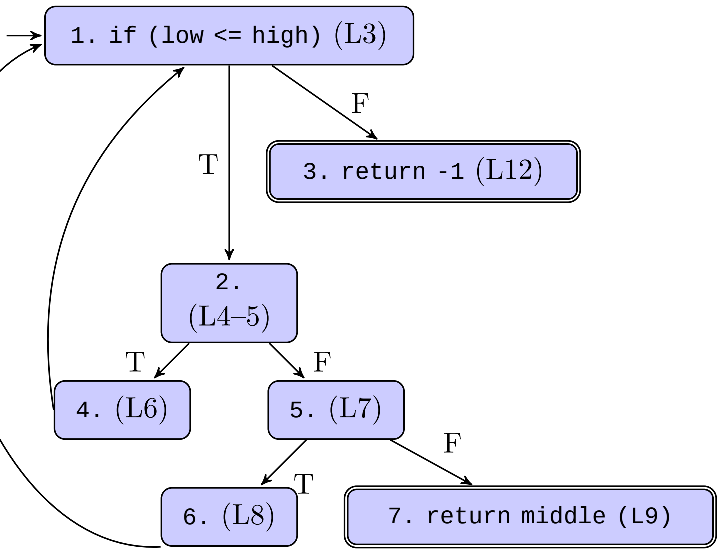
Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 8 — January 23, 2019

Patrick Lam *version 1*

Larger CFG example. You can draw a 7-node CFG for this program:

```
1    /** Binary search for target in sorted subarray a[low..high] */
2    int binary_search(int[] a, int low, int high, int target) {
3      while (low <= high) {
4        int middle = low + (high-low)/2;
5        if (target < a[middle)
6          high = middle - 1;
7        else if (target > a[middle])
8          low = middle + 1;
9        else
10         return middle;
11     }
12     return -1; /* not found in a[low..high] */
13   }
```



Here are more exercise programs that you can draw CFGs for.

```
/* effects: if x==null, throw NullPointerException
            otherwise, return number of elements in x that are odd, positive or both. */
int oddOrPos(int[] x) {
  int count = 0;
  for (int i = 0; i < x.length; i++) {
    if (x[i]%2 == 1 || x[i] > 0) {
      count++;
    }
  }
  return count;
}

// example test case: input: x=[-3, -2, 0, 1, 4]; output: 3
```

Finally, we have a really poorly-designed API (I'd give it a D at most, maybe an F) because it's impossible to succinctly describe what it does. **Do not design functions with interfaces like this.** But we can still draw a CFG, no matter how bad the code is.

```
/** Returns the mean of the first maxSize numbers in the array,
    if they are between min and max. Otherwise, skip the numbers. */
double computeMean(int[] value, int maxSize, int min, int max) {
  int i, ti, tv, sum;

  i = 0; ti = 0; tv = 0; sum = 0;
  while (ti < maxSize) {
    ti++;
    if (value[i] >= min && value[i] <= max) {
      tv++;
      sum += value[i];
    }
    i++;
  }
  if (tv > 0)
    return (double)sum/tv;
  else
    throw new IllegalArgumentException();
}
```

Statement and Branch Coverage

We defined Control-Flow Graphs so that we can give principled definitions of statement and branch coverage. We can start with the definition of a test path:

Definition 1 *A* test path *is a path p (possibly of length 0) that starts at some initial node (i.e. in N_0) and ends at some final node (i.e. in N_f).*

Here's a definition of coverage for graphs:

Definition 2 *Given a set of test requirements* TR *for a graph criterion C, a test set T satisfies C on graph G iff for every test requirement* tr *in* TR*, at least one test path p in path(T) exists such that p satisfies* tr*.*

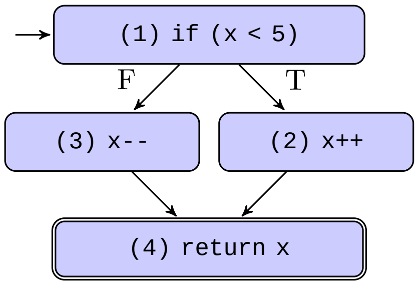
We'll use this notion to define a number of standard testing coverage criteria. But first, what are test paths?

Test cases and test paths. We connect test cases and test paths with a mapping path_G from test cases to test paths; e.g. $\text{path}_G(t)$ is the set of test paths corresponding to test case t.

- usually we just write path since G is obvious from the context.
- we can lift the definition of path to test sets T by defining $\text{path}(T) = \{\text{path}(t) | t \in T\}$.
- each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

Example. Here is a short method, the associated control-flow graph, and some test cases and test paths.

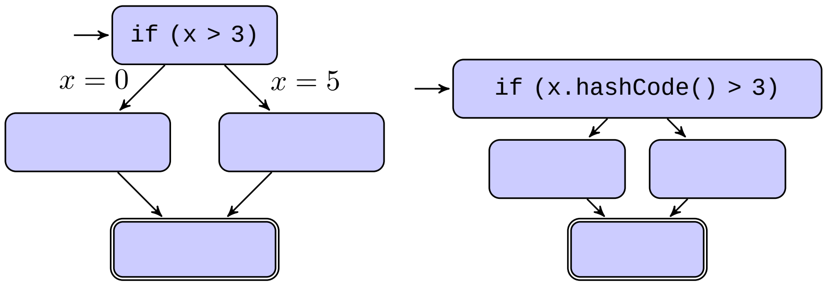
```
int foo(int x) {
  if (x < 5) {
    x ++;
  } else {
    x --;
  }
  return x;
}
```



- Test case: $x = 5$; test path: $[(1), (3), (4)]$.
- Test case: $x = 2$; test path: $[(1), (2), (4)]$.

Note that (1) we can deduce properties of the test case from the test path; and (2) in this example, since our method is deterministic, the test case determines the test path.

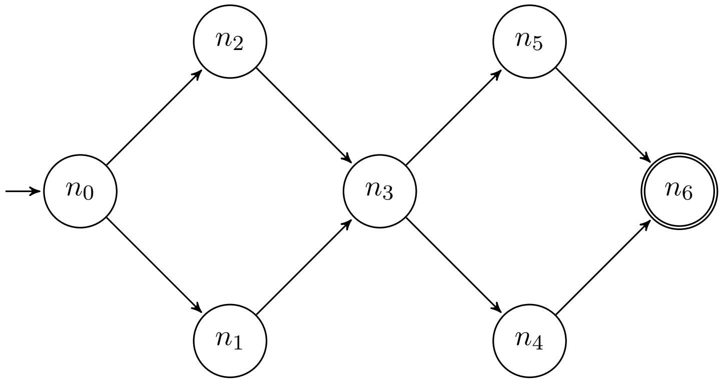
Nondeterminism. I mentioned the mapping between test cases and test paths above. The mapping is not one-to-one for nondeterminstic code. Here's an example of deterministic and non-deterministic control-flow graphs:



Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

As another (more abstract) example, consider the double-diamond graph D.



Here are the four test paths in D:

$$[n_0, n_1, n_3, n_4, n_6]$$
$$[n_0, n_1, n_3, n_5, n_6]$$
$$[n_0, n_2, n_3, n_4, n_6]$$
$$[n_0, n_2, n_3, n_5, n_6]$$

For the *statement coverage* criterion, we get the following test requirements:

$$\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

That is, any test set T which satisfies statement coverage on D must include test cases t; the cases t give rise to test paths $\text{path}(t)$, and some path must include each node from n_0 to n_6. (No single path must include all of these nodes; the requirement applies to the set of test paths.)

Let's formally define statement coverage.

Definition 3 *Statement coverage: For each node $n \in reach_G(N_0)$,* TR *contains a requirement to visit node n.*

For our example,

$$TR = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}.$$

Let's consider an example of a test set which satisfies statement coverage on D.

Start with a test case t_1; assume that executing t_1 gives the test path

$$\text{path}(t_1) = p_1 = [n_0, n_1, n_3, n_4, n_6].$$

Then test set $\{t_1\}$ does not give statement coverage on D, because no test case covers node n_2 or n_5. If we can find a test case t_2 with test path

$$\text{path}(t_2) = p_2 = [n_0, n_2, n_3, n_5, n_6],$$

then the test set $T = \{t_1, t_2\}$ satisfies statement coverage on D.

What is another test set which satisfies statement coverage on D?

Here is a more verbose definition of statement coverage.

Definition 4 *Test set T satisfies* statement coverage *on graph G if and only if for every syntactically reachable node $n \in N$, there is some path p in path(T) such that p visits n.*

A second standard criterion is that of branch coverage.

Criterion 1 Branch Coverage. *TR contains each reachable path of length up to 1, inclusive, in G.*

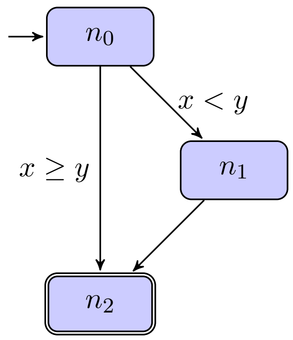
Here are some examples of paths of length ≤ 1:

Note that since we're not talking about *test paths*, these reachable paths need not start in N_0.

In general, paths of length ≤ 1 consist of nodes and edges. (Why not just say edges?)

Saying "edges" on the above graph would not be the same as saying "paths of length ≤ 1".

Another example. Here is a more involved example:



Let's define

$$\begin{aligned} \text{path}(t_1) &= [n_0, n_1, n_2] \\ \text{path}(t_2) &= [n_0, n_2] \end{aligned}$$

Then

T_1	$=$	$\langle ? \rangle$	satisfies statement coverage
T_2	$=$	$\langle ? \rangle$	satisfies branch coverage

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 9 — January 23, 2019

Patrick Lam *version 1*

Selenium[1]

Selenium is a tool for (among other things) testing web applications. In its full generality, "Selenium automates browsers" and lets you programmatically drive Web browsers. For the purposes of this class, you can use Selenium to automate web application tests, testing multiple browsers and multiple platforms.

In Assignment 1, I asked you to use Selenium to test a web application. I've simplified the setup by not actually launching a browser; Selenium contains a "headless" (GUI-less) browser simulator called `HtmlUnit`. Selenium can also drive real browsers, and it's interesting to watch it do that. Furthermore, you can set up a bunch of different computers and drive them from a single Selenium script.

Selenium also includes record-replay functionality (the Selenium IDE, a Firefox add-on), but I won't discuss it.

Anatomy of a Selenium Test. There are many bindings for Selenium, but we'll talk about using Selenium from JUnit.

Like any other test, there is setup and teardown; for Selenium, you ask for the browser to be started and terminated.

```
@Before
public void openBrowser() {
  baseUrl = System.getProperty("webdriver.base.url");
  driver = new FirefoxDriver(); // could also be Chrome, IE, HtmlUnit, etc.
  driver.get(baseUrl);
}

@After
public void saveScreenshotAndCloseBrowser() throws IOException {
  driver.quit();
}
```

Note that we create a browser and ask it to load a page.

[1] http://seleniumhq.org

Now let's look at a particular test. Like any other test, it sets up the test state, calls upon the system under test to do the action, and checks the result.

```java
@Test
public void testPageTitleAfterSearchShouldBeginWithDrupal() throws IOException {
  assertEquals("The page title should equal Google at the start of the test.",
               "Google", driver.getTitle());  // (1)
  WebElement searchField = driver.findElement(By.name("q")); // (2)
  searchField.sendKeys("Drupal!"); // (3)
  searchField.submit();            // (3)
  assertTrue("The page title should start with the search string after the search.",
      // (4)
             (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
                     public Boolean apply(Object d) {
                         return ((WebDriver)d).getTitle().toLowerCase().startsWith("
                             drupal!");
                     }
                 })
             );
}
```

Here's what's going on.

1. The initial `assertEquals` is based on an assumption that the test is called on `www.google.com`.
2. Next, the test searches for the appropriate input on the webpage. (For Java tests, we don't need to search for the method that we're calling. Here, we do.) Selenium includes various ways to search the Document Object Model.
3. Having found the appropriate field, the test sends input and submits the form. This is analogous to calling the methods on the System Under Test.
4. Finally, the test waits for the action to complete (with `WebDriverWait`). Once complete, it checks that the outputs are correct; in this case, it checks that the title of the resulting page starts with "drupal!".

Waiting. In the above example, the test simply waits until the page loads (or 10 seconds). Selenium tests can also wait for more sophisticated conditions, for instance until a certain element appears on the page. (This is relevant when the JavaScript is adding content to the page dynamically).

Example:

```java
WebDriverWait wait = new WebDriverWait(driver, 10);
WebElement element =
  wait.until(ExpectedConditions.elementToBeClickable(By.id("someid")));
```

Page Objects

The above example hard-coded the “q” field on Google’s page. But things change. We can fix that by introducing a level of indirection.

References:

`http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#chapter06-reference`
`https://martinfowler.com/bliki/PageObject.html`

A page object should abstractly represent the actions that a user can take on a page and allow the caller to query the state of the page (if necessary). The Selenium documentation suggests a page object for a sign-in page:

```java
public class SignInPage {
  private WebDriver driver;

  public SignInPage(WebDriver driver) {
    this.driver = driver;
    if(!driver.getTitle().equals("Sign in page")) {
      throw new IllegalStateException("This is not sign in page, current page is: "
      +driver.getLocation());
    }
  }

  public HomePage loginValidUser(String userName, String password) {
    driver.type("usernamefield", userName);
    driver.type("passwordfield", password);
    driver.click("sign-in");
    driver.waitForPageToLoad("waitPeriod");

    return new HomePage(driver);
  }
}
```

The documentation suggests that the only verification that should occur on a page object is testing that the driver points to the right page. Everything else should be done in the tests.

The sign-in page object also exposes the sole functionality of the page, which is to sign in. It then returns the page that gets loaded after sign-in.

Note that the caller does not need to know about the identity of the username and password fields. In fact, let’s say that a new version of the app included Facebook OAuth login. It would suffice to change the `SignInPage` to use the new login functionality; no other parts of the test suite would need to change.

In assignment 2, I will ask you to implement a fairly low-level Page Object, which simply passes the field contents directly to the caller. The sign-in example demonstrates a higher level of abstraction.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 10 — January 23, 2019

Patrick Lam *version 1*

REST APIs

REST (Representational State Transfer) is an architectural style widely used in web applications. Architectural styles are SE 464 material, but relevant to Assignment 1, so we'll talk about them briefly here. [I wouldn't ask questions about REST itself on exams, but one could expect questions about testing REST systems.]

Many applications in the world these days are web apps, and as you know, these apps usually have a frontend and a backend. Assignment 1 has you testing both the frontend and the backend of my app. We'll talk about the backend here. For our purposes, it provides an API which allows the client to store data.

Key features of REST:

- client-server;
- stateless; and
- provides uniform interfaces.

Client-server is easy. The client (in our case, the front-end JavaScript code) sends requests to a server. The server is responsible for storing the data. This enables the client to change somewhat independently of the server. For instance, the server might decide to change what database it uses behind the scenes. The client doesn't need to know about that—separation of concerns.

Statelessness helps systems scale better. What this means is that the server isn't responsible for remembering anything about client identities. The server receives requests and processes them independently. The usual concern is about authentication; requests need to carry authentication data with them (usually provided by a separate service). An analogy: because SE is a small program, I try to remember state in my head about each of you between interactions—at least your name. Larger programs need students to provide the necessary state (student ID numbers) with each request; the advisors would then look you up in a database. You can see how my approach doesn't scale to larger sets of students.

Uniform interfaces include resources and verbs. Resources are what the REST service is modifying, e.g. cat pictures. Each resource has a Uniform Resource Identifier. In our se465-flashcards app, it's just an ID number. But it could be more complicated, e.g. `/store/1/employee/2`. Verbs are what to do with the resources. HTTP defines standard operations, including `POST`, `GET`, and `DELETE`.

Further reading:

- Brian Mosigisi. "A Quick Understanding of REST".
  `https://scotch.io/bar-talk/a-quick-understanding-of-rest`

- Wikipedia is an OK reference here: `https://en.wikipedia.org/wiki/Representational_state_transfer`

Testing the REST API

Sending REST requests. At the lowest level, we need to be able to send REST requests. It's easy to send GET requests in the browser by just navigating to a page. You can't quite send other requests with your bare hands. But you can use the browser developer tools in Firefox and Chrome. The syntax can be a bit unwieldy; stackoverflow provides the following example[1]:

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({
    title: 'foo', body: 'bar', userId: 1
  }),
  headers: {
    'Content-type': 'application/json; charset=UTF-8'
  }
})
.then(res => res.json())
.then(console.log)
```

Or, you can use command-line tools like `cURL` and `wget`. The Firefox dev tools will allow you to copy a request as cURL and run at the command line, possibly after editing.

But really, you want to script this. REST Assured lets you write Java code like this[2]:

```
    @Test public void makeSureThatGoogleIsUp() {
        given().when().get("http://www.google.com").then().statusCode(200);
    }
```

and run it as a JUnit test.

To check the result, one can compare the status code to what's expected. 200 OK is usually a good HTTP code, which APIs should return when everything went well. And there's also a result, which one can reason about:

```
    @Test public void verifyNameStructured() {
        given().when().get("/garage").then().body("name",equalTo("Acme garage"));
    }
```

You can also construct a POST request using the REST Assured API.

Testing Strategy. We discussed testing at the tactical level above, which is all you have to do for Assignment 1. The broader view from this course also includes test suite construction strategies, and we can talk about various kinds of coverage. You could, for instance, require coverage of the complete API. Better yet, you could require API coverage and also require that the tests cover the specified behaviour of the API.

[1] `https://stackoverflow.com/questions/14248296/making-http-requests-using-chrome-developer-tools`

[2] `https://semaphoreci.com/community/tutorials/testing-rest-endpoints-using-rest-assured`

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 11 — January 30, 2019

Patrick Lam *version 0*

Review: Statements, branches and beyond

We talked about statement (length-0 paths) and branch coverage (length-1 paths) last time. In lecture, we reviewed the elements needed to satisfy these coverage criteria: given a graph and a criterion, we get a set of Test Requirements TR. We execute each test t in the test set T on the system under test, giving a set of test paths $p \in P$. The criterion is satisfied if there exists at least one test path $p \in P$ that satisfies each of the test requirements tr $\in$ TR.

I'll say this again later, but for real programs, 80% coverage is usually good enough; but also consider what is not tested. Also, Assignment 1 Question 1 should point out to you that it's possible to have 100% statement coverage but not actually test anything, if you write test cases that don't have asserts.

We could extend to paths of length 2 and beyond, but soon that gets us to Complete Path Coverage (CPC), which requires an infinite number of test requirements.

Criterion 1 Complete Path Coverage. *(CPC)* TR *contains all paths in G.*

Note that CPC is impossible to achieve for graphs with loops.

[note: conceptually, the L14 notes should be here]

Testing State Behaviour of Software via FSMs

We can also model the behaviour of software using a finite-state machine. Such models are higher-level than the control-flow graphs that we've seen to date. They instead capture the design of the software. There is generally no obvious mapping between a design-level FSM and the code.

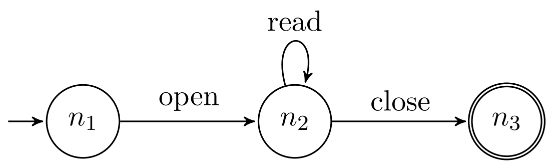
We propose the use of graph coverage criteria to test with FSMs.

- nodes: software states (e.g. sets of values for key variables);
- edges: transitions between software states, i.e. something changes in the environment or someone enters a command.

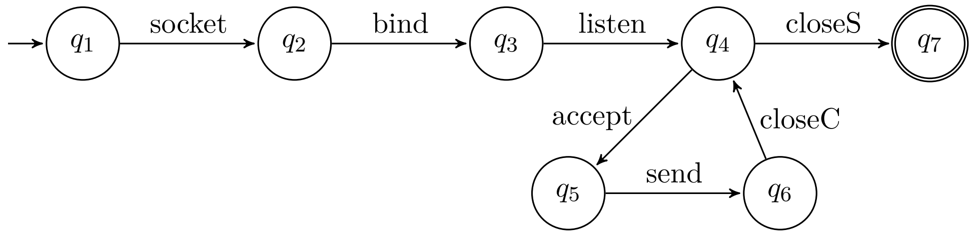
The FSM enables exploration of the software system's state space. A software state consists of values for (possibly abstract) program variables, while a transition represents a change to these program variables. Often transitions are guarded by preconditions and postconditions; the preconditions must hold for the FSM to take the corresponding transition, and the postconditions are guaranteed to hold after the FSM has taken the transition.

- node coverage: visiting every FSM state = state coverage;
- edge coverage: visiting every FSM transition = transition coverage;
- edge-pair coverage (extension of edge coverage to paths of length at most 2): actually useful for FSMS; transition-pair, two-trip coverage.

Examples. The next few graphs represent finite state machines rather than control-flow graphs. Our motivation will be to set up criteria that visit round trips in cyclic graphs.



or perhaps



The next criteria are mostly not for CFGs.

Definition 1 *A* round trip *path is a path of nonzero length with no internal cycles that starts and ends at the same node.*

Criterion 2 Simple Round Trip Coverage. *(SRTC)* TR *contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

Criterion 3 Complete Round Trip Coverage. *(CRTC)* TR *contains all round-trip paths for each reachable node in G.*

Exercise. Create a Finite State Machine for some system that you're familiar with.

Deriving Finite-State Machines

You might have to test software which doesn't come with a handy FSM. Deriving an FSM aids your understanding of the software. (You might be finding yourself re-deriving the same FSM as the software evolves; design information tends to become stale.)

We'll see some tools—iComment and Daikon—for obtaining sequencing constraints from comments/documentation and from the code.

Control-Flow Graphs. Does not really give FSMs.

- nodes aren't really states; they just abstract the program counter;
- inessential nondeterminism due e.g. to method calls;
- can only build these when you have an implementation;
- tend to be large and unwieldy.

Software Structure. Better than CFGs.

- subjective (which is not necessarily bad);
- requires lots of effort;
- requires detailed design information and knowledge of system.

Modelling State. This approach is more mechanical: once you've chosen relevant state variables and abstracted them, you need not think much.

You can also remove impossible states from such an FSM, for instance by using domain knowledge.

Specifications. These are similar to building FSMs based on software structure. Generally cleaner and easier to understand. Should resemble UML statecharts.

General Discussion. Advantages of FSMs:

- enable creation of tests before implementation;
- easier to analyze an FSM than the code.

Disadvantages:

- abstract models are not necessarily exhaustive;
- subjective (so they could be poorly done);
- FSM may not match the implementation.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 12 — February 4, 2019

Patrick Lam *version 1*

Syntax-Based Testing

We are going to completely switch gears now. We will see two applications of context-free grammars:

1. input-space grammars: create inputs (both valid and invalid)
2. program-based grammars: modify programs (mutation testing)

Mutation testing. The basic idea behind mutation testing is to improve your test suites by creating modified programs (*mutants*) which force your test suites to include test cases which verify certain specific behaviours of the original programs by killing the mutants. We'll talk about this in more detail in a week or two.

Generating Inputs: Regular Expressions and Grammars

Consider the following Perl regular expression for Visa numbers:

```
^4[0-9]{12}(?:[0-9]{3})?$
```

Idea: generate "valid" tests from regexps and invalid tests by mutating the grammar/regexp. (Why did I put valid in quotes? What is the fundamental limitation of regexps?)

Instead, we can use grammars to generate inputs (including sequences of input events).

Typical grammar fragment:

$$\begin{aligned}
\text{mult\_exp} &= \text{unary\_exp} \mid \text{mult\_exp}\ \texttt{STAR}\ \text{unary\_arith\_exp} \mid \text{mult\_exp}\ \texttt{DIV}\ \text{unary\_arith\_exp};\\
\text{unary\_exp} &= \text{quant\_exp} \mid \text{unary\_exp}\ \texttt{DOT INT} \mid \text{unary\_exp}\ \text{up};\\
&\ \ \vdots\\
\text{start} &= \text{header}^{?}\text{declaration}^{*}
\end{aligned}$$

Using Grammars. Two ways you can use input grammars for software testing and maintenance:

- recognizer: can include them in a program to validate inputs;
- generator: can create program inputs for testing.

Generators start with the start production and replace nonterminals with their right-hand sides to get (eventually) strings belonging to the input languages. Typically you would generate inputs until you have enough, either randomly sampling from the input space or systematically generating inputs of larger and larger sizes.

Another Grammar.

```
roll     =  action*
action   =  dep | deb
dep      =  "deposit" account amount
deb      =  "debit" account amount
account  =  digit { 3 }
amount   =  "$" digit+ "." digit { 2 }
digit    =  ["0" – "9"]
```

Examples of valid strings.

Note: creating a grammar for a system that doesn't have one, but should, is a useful QA exercise. Using this grammar at runtime to validate inputs can improve software reliability, although it makes tests generated from the grammar less useful.

Some Grammar Mutation Operators.

- Nonterminal Replacement; e.g.
  dep = "deposit" account amount $\Longrightarrow$ dep = "deposit" amount amount

  (Use your judgement to replace nonterminals with similar nonterminals.)
- Terminal Replacement; e.g.
  amount = "$" digit$^+$ "." digit { 2 } $\Longrightarrow$ amount = "$" digit$^+$ "$" digit { 2 }

- Terminal and Nonterminal Deletion; e.g.
  `dep = "deposit" account amount` $\Longrightarrow$ `dep = "deposit" amount`

- Terminal and Nonterminal Duplication; e.g.
  `dep = "deposit" account amount` $\Longrightarrow$ `dep = "deposit" account account amount`

Using grammar mutation operators.

1. mutate grammar, generate (invalid) inputs; or,
2. use correct grammar, but mis-derive a rule once—gives "closer" inputs (since you only miss once.)

Why test invalid inputs? Bill Sempf (@sempf), on Twitter: "QA Engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 999999999 beers. Orders a lizard. Orders -1 beers. Orders a sfdeljknesv."

If you're lucky, your program accepts strings (or events) described by a regexp or a grammar. But you might not use a parser or regexp engine. Generating using the regexp or grammar helps detect deviations, both right now and in the future.

What we'll do is to mutate the grammars and generate test strings from the mutated grammars.

Some notes:

- Can generate strings still in the grammar even after mutation.
- Recall that we aren't talking about semantic checks.
- Some programs accept only a subset of a specified larger language, e.g. Blogger HTML comments. Then testing intersection is useful.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 13 — February 4/6, 2019

Patrick Lam *version 1*

Fuzzing

Consider the following JavaScript code[1].

```
function test() {
    var f = function g() {
        if (this != 10) f();
    };
    var a = f();
}
test();
```

Turns out that it can crash WebKit (`https://bugs.webkit.org/show_bug.cgi?id=116853`). Plus, it was automatically generated by the Fuzzinator tool, based on a grammar for JavaScript.

Fuzzing is the modern-day implementation of the input space based grammar testing that we talked about in last time. While the fundamental concepts were in the earlier lecture, we will see how those concepts actually work in practice. Fuzzing effectively finds software bugs, especially security-based bugs (caused, for instance, by a lack of sufficient input validation.)

Origin Story. It starts with line noise. In 1988, Prof. Barton Miller was using a 1200-baud dialup modem to communicate with a UNIX system on a dark and stormy night. He found that the random characters inserted by the noisy line would cause his UNIX utilities to crash. He then challenged graduate students in his Advanced Operating Systems class to write a fuzzer—a program which would generate (unstructured ASCII) random inputs for other programs. The result: the students observed that 25%-33% of UNIX utilties crashed on random inputs[2].

(That was not the earliest known example of fuzz testing. Apple implemented "The Monkey" in 1983[3] to generate random events for MacPaint and MacWrite. It found lots of bugs. The limiting factor was that eventually the monkey would hit the Quit command. The solution was to introduce a system flag, "MonkeyLives", and have MacPaint and MacWrite ignore the quit command if MonkeyLives was true.)

How Fuzzing Works. Two kinds of fuzzing: *mutation-based* and *generation-based*. Mutation-based testing starts with existing test cases and randomly modifies them to explore new behaviours. Generation-based testing starts with a grammar and generates inputs that match the grammar.

[1]`http://webkit.sed.hu/blog/20130710/fuzzinator-mutation-and-generation-based-browser-fuzzer`
[2]`http://pages.cs.wisc.edu/~bart/fuzz/Foreword1.html`
[3]`http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt`

One detail that I didn't mention last time was the bug detection part. Back then, I just talked about generating interesting inputs. In fuzzing, you feed these inputs to the program and find crashes, or assertion failures, or you run the program under a dynamic analysis tool such as Valgrind and observe runtime errors.

The Simplest Thing That Could Possibly Work. Consider generation-based testing for HTML5. The simplest grammar—actually a regular expression—that could possibly work[4] is `.*`, where `.` is "any character" and `*` means "0 or more". Indeed, that grammar found the following WebKit assertion failure: `https://bugs.webkit.org/show_bug.cgi?id=132179`.

The process is as described previously. Take the regular expression and generate random strings from it. Feed them to the browser and see what happens. Find an assertion failure/crash.

More sophisticated fuzzing. Let's say that we're trying to generate C programs. One could propose the following hierarchy of inputs[5]:

1. sequence of ASCII characters;
2. sequence of words, separators, and white space (gets past the lexer);
3. syntactically correct C program (gets past the parser);
4. type-correct C program (gets past the type checker);
5. statically conforming C program (starts to exercise optimizations);
6. dynamically conforming C program;
7. model conforming C program.

Each of these levels contains a subset of the inputs from previous levels. However, as the level increases, we are more likely to find interesting bugs that reveal functionality specific to the system (rather than simply input validation issues).

While the example is specific to C, the concept applies to all generational fuzzing tools. Of course, the system under test shouldn't ever crash on random ASCII characters. But it's hard to find the really interesting cases without incorporating knowledge about correct syntax for inputs (or, as in the Apple case, excluding the "quit" command). Increasing the level should also increase code coverage.

John Regehr discusses this issue at greater length[6] and concludes that generational fuzzing tools should operate at all levels.

[4]`http://trevorjim.com/a-grammar-for-html5/`
[5]`http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf`
[6]`blog.regehr.org/archives/1039`

Mutation-based fuzzing. In mutation-based fuzzing, you develop a tool that randomly modifies existing inputs. You could do this totally randomly by flipping bytes in the input, or you could parse the input and then change some of the nonterminals. If you flip bytes, you also need to update any applicable checksums if you want to see anything interesting (similar to level 3 above).

Here's a description of a mutation-based fuzzing workflow by the author of Fuzzinator.

> More than a year ago, when I started fuzzing, I was mostly focusing on mutation-based fuzzer technologies since they were easy to build and pretty effective. Having a nice error-prone test suite (e.g. LayoutTests) was the warrant for fresh new bugs. At least for a while. As expected, the test generator based on the knowledge extracted from a finite set of test cases reached the edge of its possibilities after some time and didn't generate essentially new test cases anymore. At this point, a fuzzer girl can reload her gun with new input test sets and will probably find new bugs. This works a few times but she will soon find herself in a loop testing the common code paths and running into the same bugs again and again.[7]

Fuzzing Summary. Fuzzing is a useful technique for finding interesting test cases. It works best at interfaces between components. Advantages: it runs automatically and really works. Disadvantages: without significant work, it won't find sophisticated domain-specific issues.

Related: Chaos Monkey

Instead of thinking about bogus inputs, consider instead what happens in a distributed system when some instances (components) randomly fail (because of bogus inputs, or for other reasons). Ideally, the system would smoothly continue, perhaps with some graceful degradation until the instance can come back online. Since failures are inevitable, it's best that they occur when engineers are around to diagnose them and prevent unintended consequences of failures.

Netflix has implemented this in the form of the Chaos Monkey[8] and its relatives. The Chaos Monkey operates at instance level, while Chaos Gorilla disables an Availability Zone, and Chaos Kong knocks out an entire Amazon region. These tools, and others, form the Netflix Simian Army[9].

Jeff Atwood (co-founder of StackOverflow) writes about experiences with a Chaos Monkey-like system[10]. Why inflict such a system on yourself? "Sometimes you don't get a choice; the Chaos Monkey chooses you." In his words, software engineering benefits of the Chaos Monkey included:

- "Where we had one server performing an essential function, we switched to two."
- "If we didn't have a sensible fallback for something, we created one."
- "We removed dependencies all over the place, paring down to the absolute minimum we required to run."
- "We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available."

[7] `http://webkit.sed.hu/blog/20141023/fuzzinator-reloaded`
[8] `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`
[9] `http://techblog.netflix.com/2011/07/netflix-simian-army.html`
[10] `http://blog.codinghorror.com/working-with-the-chaos-monkey/`

Software Testing, Quality Assurance and Maintenance Winter 2019

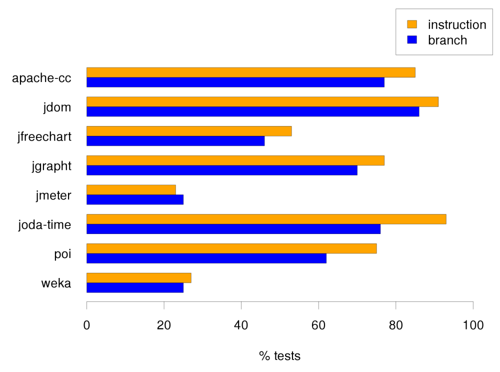
Lecture 14 — February 6, 2019

Patrick Lam *version 1*

[this makes more sense after the summary in L11]

We'll wrap up our unit on defining test suites by exploring the question "How much is enough?" We'll discuss coverage first and then mutation testing as ways of answering this question.

First, we can look at actual test suites and see how much coverage they achieve. I collected this data a few years ago, measured with the EclEmma tool.



We can see that the coverage varies between 20% and 95% on actual open-source projects. I investigated further and found that while Weka has low test coverage, it instead uses scientific peer review for QA: its features come from published articles. Common practice in industry is that about 80% coverage (doesn't matter which kind) is good enough.

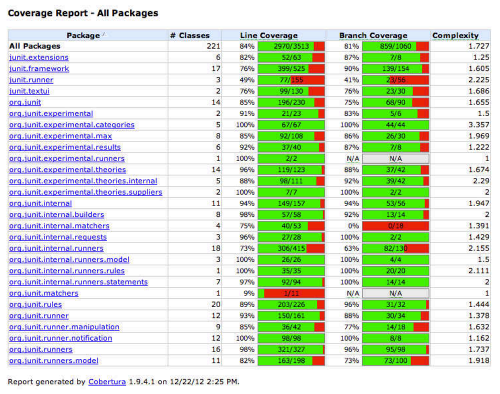
There is essentially no quantitative analysis of input space coverage; for most purposes, input spaces are essentially infinite.

Let's look at a more specific case study, JUnit. The rest of this lecture is based on a blog post by Arie van Deursen:

`https://avandeursen.com/2012/12/21/line-coverage-lessons-from-junit/`

Although you might think of JUnit as something that just magically exists in the world, it is a software artifact too. JUnit is written by developers who obviously really care about testing. Let's see what they do.

Here's the Cobertura report for JUnit:



Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	221	84%	2970/3513	81%	859/1060	1.727
junit.extensions	6	82%	52/63	87%	7/8	1.25
junit.framework	17	76%	399/525	90%	139/154	1.605
junit.runner	3	49%	77/155	41%	23/56	2.225
junit.textui	2	76%	99/130	76%	23/30	1.686
org.junit	14	85%	196/230	75%	68/90	1.655
org.junit.experimental	2	91%	21/23	83%	5/6	1.5
org.junit.experimental.categories	5	100%	67/67	100%	44/44	3.357
org.junit.experimental.max	8	85%	92/108	86%	26/30	1.969
org.junit.experimental.results	6	92%	37/40	87%	7/8	1.222
org.junit.experimental.runners	1	100%	2/2	N/A	N/A	1
org.junit.experimental.theories	14	96%	119/123	88%	37/42	1.674
org.junit.experimental.theories.internal	5	88%	98/111	92%	39/42	2.29
org.junit.experimental.theories.suppliers	2	100%	7/7	100%	2/2	2
org.junit.internal	11	94%	149/157	94%	53/56	1.947
org.junit.internal.builders	8	98%	57/58	92%	13/14	2
org.junit.internal.matchers	4	75%	40/53	0%	0/18	1.391
org.junit.internal.requests	3	96%	27/28	100%	2/2	1.429
org.junit.internal.runners	18	73%	306/415	63%	82/130	2.155
org.junit.internal.runners.model	3	100%	26/26	100%	4/4	1.5
org.junit.internal.runners.rules	1	100%	35/35	100%	20/20	2.111
org.junit.internal.runners.statements	7	97%	92/94	100%	14/14	2
org.junit.matchers	1	9%	1/11	N/A	N/A	1
org.junit.rules	20	89%	203/226	96%	31/32	1.444
org.junit.runner	12	93%	150/161	88%	30/34	1.378
org.junit.runner.manipulation	9	85%	36/42	77%	14/18	1.632
org.junit.runner.notification	12	100%	98/98	100%	8/8	1.162
org.junit.runners	16	98%	321/327	96%	95/98	1.737
org.junit.runners.model	11	82%	163/198	73%	73/100	1.918

Report generated by Cobertura 1.9.4.1 on 12/22/12 2:25 PM.

Stats. Overall instruction (statement) coverage for JUnit 4.11 is about 85%; there are 13,000 lines of code and 15,000 lines ot test code. (It's not that unusual for there to be more tests than code.) This is consistent with the industry average.

Deprecated code? Sometimes library authors decide that some functionality was not a good idea after all. In that case they might *deprecate* some methods or classes, signalling that these APIs will disappear in the future.

In JUnit, deprecated and older code has lower coverage levels. Its 13 deprecated classes have only 65% instruction coverage. Ignoring deprecated code, JUnit achieves 93% instruction coverage. Furthermore, newer code in package `org.junit.*` has 90% instruction coverage, while older code in `junit.*` has 70% instruction coverage.

(Why is this? Perhaps the coverage decreased over time for the deprecated code, since no one is really maintaining it anymore, and failing test cases just get removed.)

Untested class. The blog post points out one class that was completely untested, which is unusual for JUnit. It turns out that the code came with tests, but that the tests never got run because they were never added to any test suites. Furthermore, these tests also failed, perhaps because no one had ever tried them. The continuous integration infrastructure did not detect this change. (More on CI later.)

What else? Arie van Deursen characterizes the remaining 6% as "the usual suspects". In JUnit's case, there was no method with more than 2 to 3 uncovered lines. Here's what he found.

Too simple to test. Sometimes it doesn't make sense to test a method, because it's not really doing anything. For instance:

```
public static void assumeFalse(boolean b) {
  assumeTrue(!b);
}
```

or just getters or `toString()` methods (which can still be wrong).

The empty method is also too simple to test; one might write such a method to allow it to be overridden in subclasses:

```
/**
* Override to set up your specific external resource.
*
* @throws if setup fails (which will disable {@code after}
*/
protected void before() throws Throwable {
  // do nothing
}
```

Dead by design. Sometimes a method really should never be called, for instance a constructor on a class that should never be instantiated:

```
/**
* Protect constructor since it is a static only class
*/
protected Assert() { }
```

A related case is code that should never be executed:

```
catch (InitializationError e) {
  throw new RuntimeException(
  "Bug in saff's brain: " +
  "Suite constructor, called as above, should always complete");
}
```

Similarly, switch statements may have unreachable default cases. Or other unreachable code. Sometimes the code is just highly unlikely to happen:

```
try {
  ...
} catch (InitializationError e) {
  return new ErrorReportingRunner(null, e); // uncovered
}
```

Conclusions. We explored empirically the instruction coverage of JUnit, which is written by people who really care about testing. Don't forget that coverage doesn't actually guarantee, by itself, that your code is well-exercised; what is in the tests matters too. For non-deprecated code, they achieved 93% instruction coverage, and so it really is possible to have no more than 2-3 untested lines of code per method. It's probably OK to have lower coverage for deprecated code. Beware when you are adding a class and check that you are also testing it.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 15 — February 13, 2019

Patrick Lam *version 1.1*

Mutation Testing

The second major way to use grammars in testing is mutation testing. Let's start with an example. Here is a program, along with some mutations to the program, which are derived using the grammar.

```
// original                             // with mutants
int min(int a, int b) {                 int min(int a, int b) {
  int minVal;                             int minVal;
  minVal = a;                             minVal = a;
                                          minVal = b;               // Δ 1
  if (b < a) {                            if (b < a) {
                                          if (b > a) {              // Δ 2
                                          if (b < minVal) {         // Δ 3
    minVal = b;                             minVal = b;
                                            BOMB();                 // Δ 4
                                            minVal = a;             // Δ 5
                                            minVal = failOnZero(b); // Δ 6
  }                                       }
  return minVal;                          return minVal;
}                                       }
```

Conceptually we've shown 6 programs, but we display them together for convenience. You'll find code in `live-coding/minval.c`.

We're generating mutants m for the original program m_0.

Definition 1 *Test case t* kills *m if running t on m gives different output than running t on m_0.*

We use these mutants to evaluate test suites. Here's a simple test suite. I'm abstractly representing a JUnit test case by a tuple; assume that it calls `min()` and asserts on the return value. You can fill out this table.

	Δ 1	Δ 2	Δ 3	Δ 4	Δ 5	Δ 6
$\langle a = 0, b = 1, \exp = 0\rangle$	kill		–			
$\langle a = 1, b = 0, \exp = 0\rangle$	–		–			
$\langle a = 1, b = 1, \exp = 1\rangle$			–			
$\langle a = 1, b = 349, \exp = 1\rangle$			–			

Note that, for instance, Δ 3 is not killable; if you look at the modification, you can see that it is equivalent to the original.

The idea is to use mutation testing to evaluate test suite quality/improve test suites. Good test suites ought to be effective at killing mutants.

General Concepts

Mutation testing relies on two hypotheses, summarized from [DPHG$^+$18].

The *Competent Programmer Hypothesis* posits that programmers usually are almost right. There may be "subtle, low-level faults". Mutation testing introduces faults that are similar to such faults. (We can think of exceptions to this hypothesis—if the code isn't tested, for instance; or, if the code was written to the wrong requirements.)

The *Coupling Effect Hypothesis* posits that complex faults are the result of simple faults combining; hence, detecting all simple faults will detect many complex faults.

If we accept these hypotheses, then test suites that are good at ensuring program quality are also good at killing mutants.

Mutation is hard to apply by hand, and automation is complicated. The testing community generally considers mutation to be a "gold standard" that serves as a benchmark against which to compare other testing criteria against. For example, consider a test suite T which ensures statement coverage. What can mutation testing say about how good T is?

Mutation testing proceeds as follows.

1. *Generate mutants:* apply mutation operators to the program to get a set of mutants M.
2. *Execute mutants:* execute the test suite on each mutant and collect suite pass/fail results.
3. *Classify:* interpret the results as either killing each mutant or not; a failed test suite execution implies a killed mutant.

Although you could generate mutants by hand, typically you would tend to use a tool which parses the input program, applies a mutation operator, and then unparses back to source code, which is then recompiled.

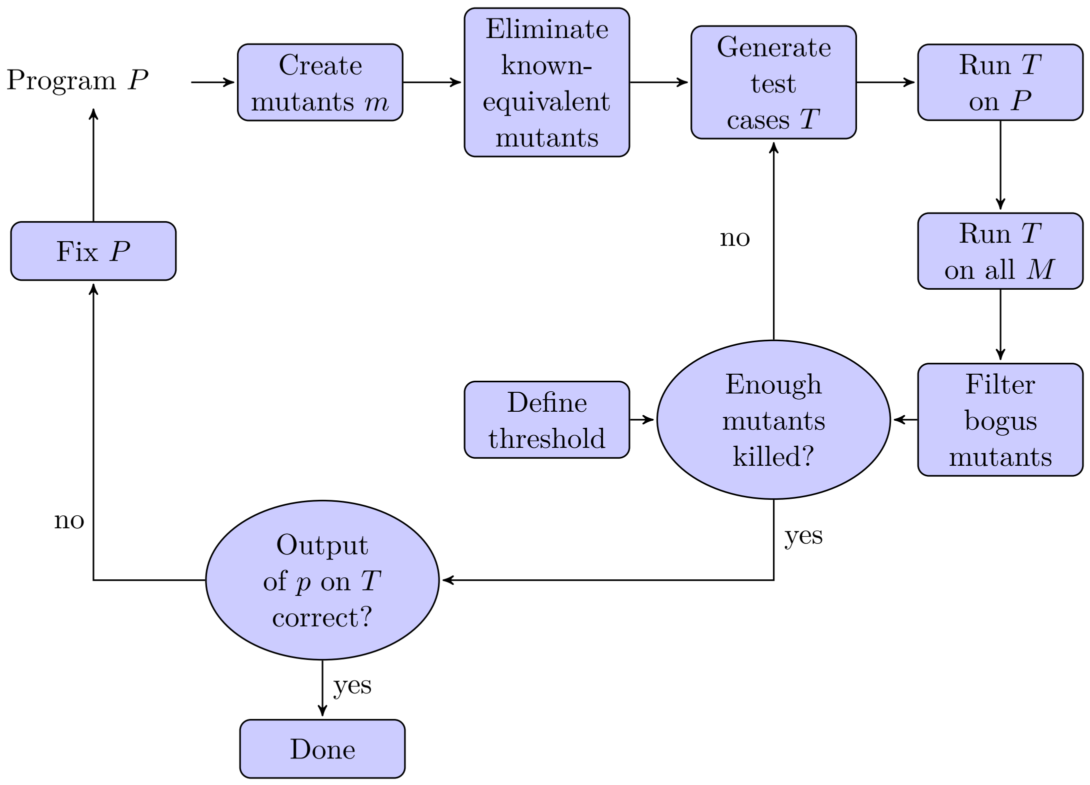
Executing the mutants can be computationally expensive, since you have to run the entire test suite on each of the mutants. This is a good time to use all the compute infrastructure available to you.

Generating and executing are computationally expensive, but classifying is worse, because it requires manual analysis. In particular, a not-killed result could be due to an equivalent mutant (like Δ 3 above); compilers can help, but the problem is fundamentally undecidable. Alternately, not-killed could be because the test suite isn't good enough. It's up to you to distinguish these cases. On the next page, "bogus mutants" denotes equivalent, stillborn and trivial mutants.

You would normally want to then craft new test cases to kill the non-equivalent mutants that you found.

Testing Programs with Mutation

Here's a picture that illustrates a variant of the above workflow.



Generating Mutants

Now let's see how to generate mutants; this ties in to the grammar-based material I talked about last time. For mutation testing, strings will always be programs.

Definition 2 *Ground string: a (valid) string belonging to the language of the grammar (i.e. a programming language grammar).*

Definition 3 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 4 *Mutant: the result of one application of a mutation operator to a ground string.*

The workflow is to parse the ground string (original program), apply a mutation operator, and then unparse.

It is generally difficult to find good mutation operators. One example of a bad mutation operator might be to change all boolean expressions to "true". Fortunately, the research shows that you don't need many mutation operators—the right 5 will do fine.

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *Too much work; choose randomly.*

Killing Mutants. We can also define a mutation score, which is the percentage of mutants killed.

To use mutation testing for generating test cases, one would measure the effectiveness of a test suite (the mutation score), and keep adding tests until reaching a desired mutation score.

So far we've talked about requiring differences in the *output* for mutants. We call such mutants **strong mutants**. We can relax this by only requiring changes in the *state*, which we'll call **weak mutants**.

In other words,

- *strong mutation*: fault must be *reachable*, *infect* state, and **propagate** to output.
- *weak mutation*: a fault which kills a mutant need only be *reachable* and *infect state.*

Supposedly, experiments show that weak and strong mutation require almost the same number of tests to satisfy them.

Let's consider mutant $\Delta 1$ from above, i.e. we change `minVal = a` to `minVal = b`. In this case:

- reachability: unavoidable;
- infection: need $b \neq a$;
- propagation: wrong `minVal` needs to return to the caller; that is, we can't execute the body of the `if` statement, so we need $b > a$.

A test case for strong mutation is therefore $a = 5, b = 7$ (return value = ␣, expected ␣), and for weak mutation $a = 7, b = 5$ (return value = ␣, expected ␣).

Now consider mutant $\Delta 3$, which replaces `b < a` with `b < minVal`. This mutant is an equivalent mutant, since `a = minVal`. (The infection condition boils down to "false".)

Equivalence testing is, in its full generality, undecidable, but we can always estimate.

Program Based Grammars

The usual way to use mutation testing for generating test cases is by generating mutants by modifying programs according to the language grammar, using mutation operators.

Mutants are *valid programs* (not tests) which ought to behave differently from the ground string.

Mutation testing looks for tests which distinguish mutants from originals.

Example. Given the ground string `x = a + b`, we might create mutants `x = a - b`, `x = a * b`, etc. A possible original on the left and a mutant on the right:

```
int foo(int x, int y) { // original
  if (x > 5) return x + y;
  else return x;
}
```

```
int foo(int x, int y) { // mutant
  if (x > 5) return x - y;
  else return x;
}
```

In this example, the test case $\langle 6, 2 \rangle$ will kill the mutant, since it returns 8 for the original and 4 for the mutant, while the case $\langle 6, 0 \rangle$ will not kill the mutant, since it returns 6 in both cases.

Once we find a test case that kills a mutant, we can forget the mutant and keep the test case. The mutant is then *dead*.

The other thing that can happen when you are running a test case on a mutant is that it loops indefinitely. You'd want to use a timeout when running testcases, and then you have a timeout failure, which you can presumably use to distinguish the mutant from the original.

Uninteresting Mutants. Three kinds of mutants are uninteresting:

- *stillborn*: such mutants cannot compile (or immediately crash);
- *trivial*: killed by almost any test case;
- *equivalent*: indistinguishable from original program.

The usual application of program-based mutation is to individual statements in unit-level (per-method) testing.

References

[DPHG+18] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. Evaluation of mutation testing in a nuclear industry case study. In *IEEE Transactions on Reliability*, pages 1406–1419, 2018.