CITS5501 Software Testing and Quality Assurance
Randomized testing

Unit coordinator: Arran Stewart

Overview

- Random testing
- Property-based testing
- Fuzzing

Randomized testing

ISP

When we looked at Input Space Partitioning (ISP), we say techniques for manually creating test cases for the functions in a software component.

However, another possibility is to create tests *randomly*.

The technique of choosing elements from the input domain randomly, and using them as tests – is known as **random testing**.

Random testing

Some advantages of **random testing**:

- Very easy to generate test cases
  - (However, they may not tell you anything useful)
- Lacks human biases – doesn't have preconceptions about what input values might be "better"

Property-based testing

Suppose we are trying to apply the idea of random testing to a method we say in lecture 5, `collapseSpaces`:[1]

```
/** Remove/collapse multiple spaces.
*
* @param String string to remove multiple spaces from.
* @return String */
public static String collapseSpaces(String argStr);
```

[1] From the `StringUtils` class of Apache Velocity (http://velocity.apache.org/), version 1.3.1.

Example – collapseSpaces

```
/** Remove/collapse multiple spaces.
*
* @param String string to remove multiple spaces from.
* @return String */
public static String collapseSpaces(String argStr);
```

One obvious problem with random testing is how to know what the expected value of a test should be.

For instance, suppose we generate the random string `"R1Z cz vaqxCwugL"` . . . how do we know what the expected result should be of passing this to `collapseSpaces`, except by invoking `collapseSpaces`?

Example – collapseSpaces

To apply the idea behind **property-based testing**, we don't try to *predict* what the expected result should be.

Rather, we try to formulate *laws* or *properties* which should always hold between the input and the output of a function.

For instance, suppose we ran `collapseSpaces` on some arbitrary string, and discovered that the output string was *longer* than the input string – should that be possible?

Example – collapseSpaces

To apply the idea behind **property-based testing**, we don't try to *predict* what the expected result should be.

Rather, we try to formulate *laws* or *properties* which should always hold between the input and the output of a function.

For instance, suppose we ran `collapseSpaces` on some arbitrary string, and discovered that the output string was *longer* than the input string – should that be possible?

It should not. So we can state this as a "law" about `collapseSpaces`:

> The output of `collapseSpaces` should never be longer than the input.

Example – collapseSpaces

What other “laws” can we come up with?

Example – collapseSpaces

What other "laws" can we come up with?

Some possibilities:

- `collapseSpaces` should never remove a non-space character.
    - How to test this?
    - We can run a `stripSpaces` function (hopefully trivial to implement) on the input and the output; if the law holds, the results should be identical.

Example – collapseSpaces

So the task of doing property-based testing reduces to the following:

- For each of our two tests (let's call them "`outputNeverLongThanInput`" and "`neverRemovesNonSpace`") . . .
- . . . generate many random strings.
  - We can generate *completely* random strings, drawn from any possible Unicode "character"
  - We can also generate strings that have a high chance of containing contiguous spaces
- Invoke `collapseSpaces` on the random string
- Assert that the "law" we have stated, relating input and output, holds.
  - (These "laws" are another sort of **invariant** – a rule about our software that always should hold true if the software is correct.)

Benefits of property-based testing

We now gain the benefits of random testing (easy to generate test cases), without the drawback of needing an "oracle" to tell us the expected result for our test.

Other benefits:

- In practice, a quick and fairly easy way of checking our functions for silly mistakes – *and* it forces us to clarify our thinking about what the preconditions and postconditions of our function are.
- Lets us "bootstrap" tricky functions:
  - Write a not necessarily efficient, but very straightforward implementation of a tricky function (say, for `editDistance()`, the "edit distance between two strings")
  - Start developing a better, more efficient version of the function ("`fastEditDistance`")
  - Use property-based testing to check that the following "law" holds: "the result of applying `fastEditDistance` is always the

Property-based testing frameworks

Testing frameworks that perform property-based testing include:

- Hypothesis, for Python
- QuickTheories, for Java
- jsverify, for JavaScript
- QuickCheck, the inspiration for most of the others, for Haskelll
- . . . Many more listed by David R. MacIver, the developer of Hypothesis.
- We will look at some of these testing frameworks in more detail.

Property-based testing in JUnit 5

JUnit 5 doesn't directly support property-based testing – the nearest we could get is to use parameterized JUnit tests.

We could use the `@MethodSource` annotation to specify that our tests should draw their paramaters from some particular method (call it `generateRandomString`, say), and then use the `java.util.Random` class to generate our random strings.

This isn't particularly convenient, however – a good property-based testing framework will provide us with many methods for generating random data of different sorts, perhaps then checking it for usefulness ("Does this string actually contain some spaces?") and then applying our "laws" to it.

Generators

One technique that comes in useful when doing property-based testing is to make use of syntax-based *generators*.

We saw that we can use syntax-based rules to generate complex data structures (e.g. for Word documents, or JPEG graphics files).

We can apply similar techniques to quickly generate almost any data structure we like.

Good property-based testing libraries also help us create rules so the framework can *shrink* test values for which a test fails – reduce them to a smaller, simpler example for which the test *still* fails.

reverse example

Suppose we have a method `String reverse(String s)` that reverses a string – can you think of any general “law” we can state that relates the input to the output?

reverse example

Suppose we have a method `String reverse(String s)` that reverses a string – can you think of any general "law" we can state that relates the input to the output?

- One is: if we reverse an already-reversed string, we get the original input back again.

```
String testSelfInverse(String s) {
  String newString = reverse(reverse(s));
  assertEqual(s, newString);
}
```

- Another is: the output string should always be the same length as the input string.

List.remove example

- Consider the following method specification:

  `List.remove(Object o)`: Search the list for elements which are equal to object `o` (using `.equals()`). If there are any, then the first such element is removed. Otherwise, the method does nothing.

- If L_{before} is the length of the list before we execute `remove()`, and L_{after} is the length of the list after we execute it, then the following invariant holds:

  $$(L_{after} = L_{before}) \vee (L_{after} = L_{before} - 1)$$

  Let's call this invariant Inv_1, for short.

- It is certainly good practice to write tests for `remove()` based on Input Space Partitioning – e.g. constructing small lists that do or don't contain the element being searched for, and constructing test inputs based on that.

Property-based testing

- But if we can identify invariants like Inv_1, that we think will *always* hold, then we can generate random data to improve our confidence that this is so.
- If our test framework generates a few thousand sample lists, and our invariant holds for all of them, we can be fairly confident that this theory about our method is true.
  (We cannot be *certain* – we might have failed to generate a test case that exercises some particular fault – perhaps our method fails on extremely long lists, and we never generated those – but our confidence is definitely improved.)

Other uses of random testing

Random testing can be very useful when we want to generate large volumes of valid or invalid data.

For instance for:

- Load testing – How does our software perform under high loads – the largest volumes of data we expect to receive?
  (We would like to ensure our software correctly handles the expected load.)
- Stress testing – How does our software behave when we *exceed* the expected maximum?
  (We often would still like our system to *degrade gracefully*, even though it may not be able to handle all the results.)
- Robustness testing – How well does our system handle malformed inputs?
  (We don't expect that it will give "correct" results – but we usually expect it not to suffer, say, a segmentation fault/crash.)

Other uses of random testing

Question: have you used software that crashed or otherwise misbehaved when given large inputs?

Other uses of random testing

Question: have you used software that crashed or otherwise misbehaved when given large inputs?

If so, it probably could have benefitted from load testing, stress testing, and/or robustness testing.

Fuzzing

Fuzzing

One sort of robustness testing is called **fuzzing** or **fuzz testing**. It usually makes use of randomly generated inputs.

The general idea is this:

- Start with a program under test – e.g. one that converts PNG files to `.bmp` (bitmap) files.
- Provide it with invalid or unexpected data, and monitor the program's behaviour.
- If it crashes, throws expections that reach the end user, or fails assertions we have included in our code (statements that test invariants) – it's not handling the input data **robustly**.
- (We could also monitor the program for other bad behaviour such as *memory leaks* – acquiring and not releasing RAM, leading to the program "hogging" a computer's available memory).

Variants

There are many clever variants on this basic idea.

- In the simplest case, we could just use completely random inputs – but that's rarely effective, as they often don't look at all similar to valid inputs, and don't trigger "interesting" execution paths
- **Mutation-based fuzzers.** We can start with a just a small sample of inputs (e.g. some valid and invalid JPEG files, using our previous example), and randomly alter them to produce new inputs.
    - Can repeatedly mutate our existing inputs to produce more and more "generations" of input files
    - Useful for getting our inputs "past" a program's initial syntax checks
    - Usually doesn't involve any "deep" understanding of the input structure

Variants

- **“Smart” vs “dumb” fuzzers.** Our fuzzer might have some “understanding” of the input structure (i.e. the syntax of a JPEG file) to aid it in generating new inputs.
- **White-, grey-, or black-box**.
    - In the simplest case (black-box), we just monitor a program for crashes, treating it as a “black box” without understanding its internal structure.
    - But ideally, we would like our fuzzer to get good *code coverage* of the program under test.
    - White- and grey-box fuzzers while analyse or *instrument* the code of the program under test, and try to generate inputs which will e.g. take the program down execution paths it hasn't been before.

Some fuzz-testing tools

- AFL (standing for “American Fuzzy Lop”)
- javafuzz – coverage guided fuzzer for java
- JQF – coverage-guided semantic fuzzing for Java
- honggfuzz – evolutionary, feedback-driven fuzzing based on code coverage

Advaantages of fuzz testing

- Simple to do – in the simplest case, just provide some sample inputs, and the path to your program, and start the fuzzer going.
- Cheap – doesn't require extensive computational resources
- Effective – very good for finding security flaws in memory-unchecked languages (e.g. C, C++).