CITS5501 Software Testing and Quality Assurance
Syntax-based testing

Unit coordinator: Arran Stewart

Recap – models and testing

- Sometimes, we model a method (or module, or subsystem, or system) as a *function* from inputs to outputs.
- And then we can ask, "What does the input domain look like? Could I partition it up, to make the task of testing more tractable? Can I identify particularly important values in those partitions?"
- This approach leads to Input Space Partitioning.

Models and testing

- If the thing we're modeling is a "unit", then we've now developed a unit test. If it is a collection of units, or a system, then we've now developed an integration test or a system test.

Models and testing

- We've seen we can modelling a portion of a system as a graph.
- And we can then ask "Have I thoroughly explored the execution/traversal of this graph? What test inputs would I need, in order to explore this particular path through the graph?"
- This can lead us to developing new tests (which exercise particular paths through the graph), and/or noticing problems with existing tests (they leave some paths unexplored), or noticing problems with the graph we're looking at (e.g. we've found "dead code", which currently is impossible to execute; but which could cause problems if it becomes "live").

Models and testing

- And we can apply graph-based techniques to anything we could model as a graph . . .
  - classes, and the connections (coupling or inheritance) between them
  - ER diagrams
  - data flow between variables
  - data flow between components (e.g. in a secure system)
  - organization charts? Perhaps. . .

Models and testing

- Models are *abstractions* and *simplifications*, and usually, there's no one model that serves all purposes.
- Often, we can make a model more realistic, but only at the cost of more complexity.
  - e.g. When (statically) modeling control flow in software, we may have to make simplifications, because we don't know exactly what the dynamic behaviour will be.
- It is up to us to decide on a balance between abstraction and realism. Both can help highlight problems we otherwise might have missed.

Syntax-based testing

- Many sorts of software artifact can be modeled as something Amman and Offut call “syntax”.
- Basically, this means “potentially recursive structures, individual instances of which can be modeled as trees (directed acyclic graphs)”.
- We will see several examples of that, and in particular, mutation testing.

Using the Syntax to Generate Tests

- Many software artifacts follow strict syntax rules
  - e.g. The syntax for programming languages is often expressed as a grammar in using a formalism such as BNF (Backus-Naur Form)
- Syntactic descriptions can be obtained from many sources:
  - program source code
  - design documents
  - input descriptions (e.g. file formats, network message formats, etc)
- Tests are created with two general goals
  - Cover the syntax in some way
  - Violate the syntax (invalid tests)

An example of syntax-generated tests

- *Mutation-based fuzzers* use a body of inputs, and generate new ones (some valid, some invalid) by repeatedly mutating existing inputs
- Often the fuzzers aim to *crash* the program (get it to exit unexpectedly, and/or, in the case of memory-unsafe languages like C and C++, violate memory integrity).
- e.g. We could start with a set of valid PNG files, and use a mutation-based fuzzer to produce many variants of these
- Often we'll want to be sure that our software handles any sort of input *gracefully* – regardless of whether the input is valid or invalid, the program should give some sort of "proper" result (even if that is just an error message). It *shouldn't* (usually) go into an erroneous state.

What is a grammar?

A syntax is defined by a *grammar*, and the best way to explain what a grammar is, is to show an example.

Let us suppose we want to define a very simple language, which lets us write mathematical expressions involving single-digit numbers. The language lets us add and subtract them using "+" and "−", and group things using parentheses.

Example – arithmetic expressions

Some string will be *valid* in our language (like "(3 + 2) - 5") and some will not (like "3++-(").

The quivalent of "words" in our language are called *terminal symbols* – they are like atoms, in that they are the smallest, indivisible parts of our language.

They consist of the numerals 0-9, and the symbols "+ - ( )".

Example – arithmetic expressions

We can group these symbols into categories:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

This says "A *digit* is defined as either the numeral"`0`", or the numeral"`1`", or the numeral"`2`", ... (etc.)

We read `::=` as "is defined as" or "can be expended to", and `|` as "or".

Example – arithmetic expressions

Things that aren't terminals are called non-terminal symbols.

We can say, "An *expression* is either a digit, or, a smaller expression plus some other smaller expression."

```
<expression> ::= <digit> | <expression> "+" <expression>
```

Example – arithmetic expressions

Our whole grammar:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"

<expression> ::= <digit>
                 | <expression> "+" <expression>
                 | <expression> "-" <expression>
                 | "(" <expression> ")"
```

BNF grammars

- Formally, what we have used here is called a "Backus-Naur Form (BNF) specification of a context-free grammar", but we won't worry too much about the technical details.
- A BNF specification is a set of derivation rules, also called production rules
- They look like this:

```
<integer> ::= <digit>|<integer><digit>
```

- We read this as: "An *integer* consists of either (a) a *digit*, or (b) an integer followed by another digit"

BNF grammars (cont'd)

- To define "digit":

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

- The thing on the left-hand side is a *nonterminal* (e.g. `integer`, `digit`);
  a symbol that never appears on the left-hand side is a *terminal* (e.g. `0`, `1`)

BNF grammars (cont'd)

- BNF rules are of the form:

  `<symbol> ::= expression`

- *expression* contains one or more *sequences* of symbols; sequences are separated by a vertical bar ("|"), representing a *choice*
- There will normally also be a *start symbol*, representing the "top level" of whatever construct we're specifiying.
- e.g. for some programming language:

  ```
  <program_file> ::=  <import_statements><declarations><definitions>
  ```

- Each possible rewriting (i.e., each alternative) of a non-terminal is called a *production*.

BNF grammars (cont'd)

- Special symbols in BNF:

  ::= means "is defined as".
  | means "or"
  < and > are used to surround non-terminal names.

Use of grammars

- Grammars can be used to build *recognizers* (programs which decide whether a string is in the grammar – i.e., parsers) and also *generators*, which derive strings of symbols.

Alternative formalisms

- BNF works well for things in textual format (including the source code of programming language files, HTML documents, JSON documents, and so on).
- But for data in binary format (for instance, TCP packets or JPEG files), a frequently-used formalism is ASN.1 ("Abstract Syntax Notation One").
- We won't be examining ASN.1 in detail, but similar considerations apply.

Coverage criteria

- If we're developing tests based on syntax ...
- The most straightforward coverage criterion:
  use every terminal and every production rule at least once

Terminal Symbol Coverage (TSC) Test requirements contain each terminal symbol t in the grammar G.

Production Coverage (PDC) Test requirements contain each production p in the grammar G.

Coverage criteria (cont'd)

- Production coverage subsumes terminal symbol coverage;
  if we've used every production, we've also used every terminal.

Coverage criteria – an impractical one

- We could aim to cover all possible strings

Derivation Coverage (DC) Test requirements contain every possible string that can be derived from the grammar G.

- But except in special cases, this will be impractical

Bounds on coverage

- Example grammar:

```
<integer> ::= <digit>|<integer><digit>

<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
```

- The number of tests to get TS coverage is bounded by the number of terminal symbols (ten, here)
- To get production coverage, that depends on the number of productions (here: 2 for the first rule, 10 for the second – so, 12)
- Whereas the number of strings that can be generated – needed for derivation coverage – is actually infinite.
  - (likewise for, say, the set of all possible Java programs)
- Even for finite grammars (e.g. some file formats), DC will usually require an infeasibly large number of tests

Data structures

- Typically, for any format we specify syntactically (like JPEG, GIF etc.), we'll have an accompanying data structure that mirrors that the structure of the syntax, in order to manipulate in-memory objects representing that format.
- E.g. see the `JpegImageData` class from the Apache Commons Imaging library for Java, or the `png_struct_def` for the `libpng` C library.

Data structures

- But even for data structures which *don't* represent something necessarily stored in binary or textual format, we can consider them as having a syntax-like structure.
- For instance, what is a linked list? In Java-like syntax:

```
class node<V> {
  V value;
  node<V> next;
};
```

It is *either*:
(1) a null pointer, or (2) a value prepended to a list.

Data structures

- Note that this seems quite similar in structure to our grammar for integers.
- Let's consider linked lists of booleans
- Suppose we write the null pointer, an empty list, as "[]", and nodes containing boolean values as "T" and "F". and represent prepending as a colon, ":"
- Then we can actually write a BNF grammar for linked lists.

“Linked list” grammar

```
<bool> ::= "T" | "F"

<list> ::=  "[]" | <bool>":"<list>
```

“Linked list” grammar

```
<bool> ::= "T" | "F"

<list> ::=  "[]" | <bool>":"<list>
```

- So `[]` is a list, as is `T:[]`, and `F:T:[]`.
- `[]:[]` is not a valid list, nor is `[]:T`.

Data structures

- What about a linked list, where the value type `V` is, say, another class, `Person`:

```
class Person {
  PersonID  personID;
  bool isStaff;
  int  age;
}
```

- Which in turn refers to the `PersonID` class.

Data structures

- What grammars and data structures have in common is that they both have terminals (in data structures, these are atomic or "primitive" values that we cannot, or choose not to, break down any further), and they define aggregate structures in terms of simpler structures, in a potentially recursive way.
- So we can use much the same principles to see if our tests of them have good coverage, or to generate them randomly, etc.

Trees

- We can draw a *tree* structure for an expression adhering to some particular syntax called a *parse tree*:[1]

  (Here, "S" stands for "sentence", "VP" for "verb phrase", "V" for "verb", "DP" for "determiner phrase" – basically something that picks out a particular entity.)

[1] Image from https://commons.wikimedia.org/wiki/File:Precedent_example_1_decl_sent.png

Trees

- The parse tree shows what productions should be followed to parse (or alternatively, to generate) a particular string.
- In practice, the trees formed by *data structures* (as opposed to grammars) are of a slightly different sort – they are *abstract syntax trees* rather than parse trees – but we will not be too concerned with the details.

Generators

- Suppose we had the grammar:

```
<Sentence> ::= <NounPhrase><Predicate>
<NounPhrase> ::= "Alice" | "Bob" | "the hacker"
<Predicate> ::= <Verb><NounPhrase>
<Verb> ::= "hires" | "defeats"
```

- Then we can see that "Alice hires Bob" and "Bob defeats the hacker" are valid strings in the language this grammar defines (modulo some whitespace).
- And we can see how we could easily generate random valid sentences that conform to these rules.
- Being able to generate things that follow a syntax-like structure is extremely useful for testing.

Generators – network traffic

- We can use it to create traffic generators, for instance – we could generate random valid TCP traffic with which to test a router.
- TCP packets follow a syntax-like structure, so it's fairly straightforward to generate them randomly.
  A TCP packet consists of: 2 bytes representing a source port (0 through 65535), 2 bytes representing a destination port, then 4 bytes representing a "sequence number", then ... (see the TCP specification for detailed rules).
- Not all the validity rules for a TCP packet can be expressed in a syntactical way – for instance, it contains a checksum towards the end, which is calculated based on previous information – but quite a bit can.
- This is very handy for "stress" or "load" or "performance" testing – generating large amounts of data, and seeing how our system performs under the load.

Generators – http traffic

- HTTP requests for web pages also follow a syntax, so we could easily generate random HTTP traffic (for instance, to stress-test a web-server, and see how it performs under high load).
- The full syntax for HTTP requests is larger than this,[2] but the start of a simplified version of it would look something like:

```
<request> ::= <GETrequest> | <POSTrequest>
<GETrequest> ::=  "GET" <space> <URI> <space> <HTTPversion>
                  <lineend> <getheaders> <getbody>
...
```

(i.e., HTTP requests are either `GET` or `POST` requests, and `GET` requests start with the keyword `GET` then a space, then a URI, and so on...)

[2] See IETF RFC 2616, https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

Generators – http traffic

- The vast majority of randomly generated HTTP requests would not be for valid URIs, and would result in 404 errors.
- If we wanted to generate, not just random HTTP requests, but requests that actually hit part of a website, we can add in additional constraints to ensure that happens.
- (E.g. We might start by only generating URLS that begin with `https://myblog.github.io/`, if we were testing a blog site hosted on GitHub.)

Generators

- Likewise, HTML and XML documents, JSON, and many other formats all follow syntactical rules, so we can randomly generate them.
- Likewise for custom formats we may come up with.
  - e.g. If we were writing a word processor, we might want to be able generate very large random documents in our word-processor format, to see how our program holds up.

Generators

- For common formats, there are often already data generators with many capabilities:
  - Tools for constructing and generating network traffic: Ostinato, Scapy Traffic Generator, flowgrind, jtg ... see this list for many more.
  - HTTP request generators: see for example httperf
  - Random bitmap generators: see for example random.org
- If not, it is perfectly possible to write our own.

Generators and data structures

Things to note when generating data structures:

- In languages with pointers or references, it may be possible to have data structures that contain *cycles*, meaning they are no longer trees but graphs.
- For instance, we could have two linked list nodes A and B, and make A's next reference point to B, and B's point to A. (A cyclic linked list.)
- It's still possible to generate random data of that sort, but doing so takes us beyond our current scope.

More complex rules for validity

- There may be rules for validity of a format (like the existence of checksums) that can't be captured by a grammar.
- This is frequently the case, actually. BNF lets us describe what are known as "context-free" grammars, and a specification for a format may include requirements that are impossible or inconvenient to specify using BNF.
  - e.g. In a valid Java program, variables have to be declared before they are used; it's an error to assign a string literal to an int; and many other rules.
- We may be able to use simple calculations to generate or verify those.
  (e.g. to verify or generate a checksum)
- Or we may have to apply more complex rules – these are outside the scope of this unit.

Using generators for testing

- Generating random, valid values is useful for performance testing, as just described – but it is also useful for *property-based testing*, which we will see more of later.
- What is property-based testing? It's a sort of (usually randomized) testing which checks that invariants about functions hold.

Property-based testing

- 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.)

Property-based testing

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.

Applications of syntax-based testing

- *Mutation-based fuzzers* use a body of inputs, and generate new ones (some valid, some invalid) by repeatedly mutating existing inputs
- e.g. We could start with a set of valid PNG files, and use a mutation-based fuzzer to produce many variants of these
- Often we'll want to be sure that our software handles any sort of input *gracefully* – accepting it if valid, but detecting the situation when input is invalid

Mutation testing

- Grammars describe both valid and invalid strings
- A *mutant* is a variation of a valid string
  - Mutants may be valid or invalid strings
- Mutation is based on “mutation operators” and “ground strings”

What is mutation ?

We are performing mutation analysis whenever we

- use well defined rules (i.e. operators)
- defined on syntactic descriptions (i.e. grammars)
- to make systematic changes
- to the syntax or to objects developed from the syntax
  - the objects are “ground strings”

Mutation testing – definitions

- Ground string: A string in the grammar
  - (The term "ground" basically means "not having any variables" – in this context, not having any non-terminals)
- Mutation operator: A rule that specifies syntactic variations of strings generated from a grammar
- Mutant: The result of one application of a mutation operator
  - A mutant is a string

Killing Mutants

- When ground strings are mutated to create valid strings, the hope is to exhibit different behavior from the ground string
- Killing Mutants : Given a mutant m for a derivation D and a test t, t is said to "kill" m iff the output of t on D is different from the output of t on m

Syntax-based coverage criteria – mutant coverage

- We can define a coverage criterion in terms of killing mutants:

Mutation Coverage (MC) For each mutant m, the test requirements contains exactly one requirement, to kill m.

- Coverage in mutation equates to number of mutants killed
- The amount of mutants killed is called the mutation score

Coverage criteria – creating invalid strings

- When creating invalid strings, two simple criteria –
- It makes sense to either use every operator once or every production once

Mutation Production Coverage (MPC) For each mutation operator, TR contains several requirements, to create one mutated string m that includes every production that can be mutated by that operator.

Mutation Operator Coverage (MOC) For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.

Mutation example

A grammar:

```
Stream ::= action*
action ::= actG | actB
actG ::= "G" s n
actB ::= "B" t n
s ::= digit{1-3}
t ::= digit{1-3}
n ::= digit{2}   "." digit{2}   "." digit{2}
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |
          "7" | "8" | "9"
```

- Uses "*", the "Kleene star", to represent "zero or more"
- Uses braces to represent "n to m occurrences" or "n occurrences"

Mutation example (cont'd)

- A ground string:

```
G 23 08.01.90
B 19 06.27.94
```

Mutation example (cont'd)

- Some mutation operators:
  - Exchange `actG` with `actB`
  - replace digits with any other possible digit

Mutation example (cont'd)

- Using mutation operator coverage (MOC):

```
G 23 08.01.90
B 19 06.27.94
```

mutated to:

```
B 23 08.01.90
B 15 06.27.94
```

Mutation example (cont'd)

- Using mutation operator coverage (MOC):
  - *B* 22 08.01.90 *G* 19 06.27.94
  - G *13* 08.01.90 B 11 06.27.94
  - G *3* 3 08.01.90 B *12* 06.27.94