CITS5501 Software Testing and Quality Assurance
Syntax-based testing

Arran Stewart and Rachel Cardell-Oliver

Overview of Syntax-Based Testing Topics

- Motivation - What is syntax-based testing?
- Theory - Formal Grammars and Coverage Criteria
- Applications of Syntax-based testing
- Input-Space Mutation Testing
- Program-Based Mutation Testing (lecture 2)

Reading: Ammann and Offutt. Introduction to Software Testing, 2016. *Ch 9 Syntax-Based Testing*

Motivation

Modelling and Testing

Recall: In brief, we **come up with tests** by looking at requirements and specifications, and thinking about the system – **modelling it** – in different ways.

Testing Strategies

So far we have seen several different types of **models** that can be used for coming up with **tests**:

- Input Space Partitioning (Equivalence classes of inputs)
- Logic-based Testing (Look inside the parts of boolean expressions making up a decision point)
- Graph-based Testing (on Program Control Flows)
- **Syntax-based Testing** (this lecture)

Syntax-based Testing

Syntax refers to the rules and regulations for writing an artefact correctly

What sort of things can be tested with syntax-based test methods?

Anything thing that can be modelled by a **syntactic description** such as a **grammar**:

- Input commands eg command line applications
- Syntax of types eg email addresses, datetime format, URLs etc
- Data and file formats eg XML
- Computer Program Syntax eg Java or Python programs etc

Where are Grammars used in Software Development?

Developers use **grammars and syntax** of all the time, though they may not realize it.

Whenever we see a requirement such as "a date should be in the format `YYYY-MM-DD`", we're making use of a grammar (though only very informally expressed).

Analysing a date

If some requirement says that a parameter to a function, or an item in a database, should be "in the format `YYYY-MM-DD`", what it (usually) means, but more explicitly stated is:

- Any of the Y's, M's or D's can be replaced by a digit in the range 0–9 – if you provide a date that can't be generated in such a fashion, we might say you've provided a *syntactically ill-formed* date.
- There are other rules about validity (e.g. if the first "M" is replaced by a 1, then the second "M" can only be in the range 0–2), but we usually don't consider those to be *syntax* errors.
  - Dates which violate these rules are usually said to violate the *semantics* of dates, or *semantic constraints*

We'll see shortly how to define the syntax of dates using a grammar

Using Grammars to Generate Tests

- Syntactic descriptions can be obtained from many sources:
  - program source code
  - design documents
  - input descriptions (e.g. file formats, network message formats, etc)
- Grammars can be used to build **recognizers** (programs which decide whether a string is in the grammar – i.e., parsers) and also **generators**, which produce strings of symbols.
- Tests are created with two general goals
  - Cover the syntax in some way
  - Violate the syntax (invalid tests)

When to Use Grammars to Generate Tests

- Should we apply the techniques we see in this lecture to *every* example of syntactic validation / use of grammars?
- Usually not – we will usually focus on **areas of high risk** (e.g. that are easy to get wrong, or have bad impacts when we get them wrong).
- Parsing command-line arguments is sufficiently important that we should probably test it.

Theory

Grammars

Grammars just give us a way of formally specifying what things are and are not syntactically correct

Every grammar defines what is called a *language*

A language is a set of acceptable strings

A grammar for the date might look like this:

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

<date>  ::=  <digit> <digit> <digit> <digit> "-"
             <digit> <digit>  "-"
             <digit> <digit>
```

Grammars

The following grammar is *equivalent* to the previous one – in that they define the exact same set of strings – but this one provides a few hints as to the *semantics* of bits of the string (and is probably a bit easier to read).

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

<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "-" <month> "-" <day>
```

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The notation is a simplified form of BNF (Backus-Naur Form)

The following symbols are used in this notation:

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

So the first line says, "A 'digit' is defined as being either the string"0", or the string "1", or . . . "

(These symbols are sometimes called "meta-syntactic symbols", meaning symbols used to define a syntax.)

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The things in quotes are called **terminal symbols** – they are the equivalent of "words" in our language.

They are like atoms, in that they are the smallest, indivisible parts of our language.

In our case, the terminals are all strings containing a single digit.

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

The things between angle brackets are called **non-terminal symbols**

The above grammar contains five **rules** (also called **productions**)

In the sorts of grammar we will consider,[1] every rule is of the form:

non-terminal "::=" *sequence of terminals and non-terminals*

[1] Called *context-free grammars* or CNFs. See https://en.wikipedia.org/wiki/Context-free_grammar.

Notation

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<year>  ::=  <digit> <digit> <digit> <digit>
<month> ::=  <digit> <digit>
<day>   ::=  <digit> <digit>
<date>  ::=  <year> "—" <month> "—" <day>
```

To be precise: the simplest possible right-hand side (RHS) of a rule will be a sequence of terminals and non-terminals, meaning "these strings, concatenated together".

(For example – the RHS of the last rule, which means "an expansion of the 'year' rule, then a hyphen, then an expansion of the 'month' rule, then a hyphen, then an expansion of the 'day' rule.)

When we specify a grammar, there will normally be a **start symbol** representing the "top level" of whatever construct we're specifying.

More Notation

We can also insert on the RHS the following symbols, between or after terminals and non-terminals:

- bars to indicate "or" (alternatives)
- an asterisk (called the "Kleene star") to indicate "zero or more of the preceding thing"
- a plus sign to indicate "one or more of the preceding thing"
- a range of numbers (e.g. "3–4") to indicate a number of possible instances of the preceding thing.

More Notation (cont)

Notation Examples

- `"dog" | "cat"` – either the string `"dog"`, or the string `"cat"`
- `"dog"*` – zero or more instances of the string `"dog"`
- `"dog"+` – one or more instances of the string `"dog"`
- `"0"-"7"` – digits from "0" to "7" inclusive

And we can use parentheses to group things.

- `( "dog" | "cat" )+` – one or more instances of these two alternatives

Notation – asterisk

An example: the following is a fairly typical way of defining valid *identifiers* in many programming languages:

```
<digit> ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" |
             "7" | "8" | "9"
<letter> ::= "a" | "b" |    ...
<underscore> ::= "_"
<identifier> ::= ( <letter> | <underscore> )
                 (<letter> | <underscore> | <digit>)*
```

This means, “An identifier always starts with a letter or underscore; it is followed by any number (possibly zero) of characters drawn from the set of letters, digits and the underscore character”.

Example: Writing a Grammar for Program Inputs

Suppose you are writing an API for accessing a student's grade in a particular unit:

```
"getGrade" studentID unitCODE
```

getGrade is written in quotes to show this is a terminal symbol.

Write a grammar for this command line input format.

Compare your rules with another student. Did you miss any cases, or find simpler ways to express?

Remember you are specifying the *syntax* not necessarily the *semantics* of the inputs.

Coverage

BNF Coverage Criteria

- Once we have written a grammar to model our system
- If we're developing tests based on syntax . . .
- The most straightforward coverage criterion is:
  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.

  (Since every terminal must be part of some production.)

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 grammar G.

- But except in special cases, this will be impractical

Example: Coverage Criteria

- Example grammar:

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

- The number of tests to get **Terminal Symbol Coverage** is bounded by the number of terminal symbols (10, 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 for this grammar. The same is true for, say, the set of all possible Java programs.
- Even for finite grammars (e.g. some file formats), **Derivation Coverage** will usually require an infeasibly large number of tests

Applications

Applications of Syntax-Based Testing

Background

Much of the software we rely on makes use of grammars (though not always explicitly).

For very simple programs, we might analyse the arguments "by hand".

For complex programs – we typically use a *command-line argument parser* to work out whether a user has supplied a valid set of arguments (and what we should do with them).

Whenever we *validate* entries into web forms or databases, we are often are defining a syntax to do so.

First: A useful grammar writing tool

BNF Playground is a web app for designing and testing context free grammars using Backus-Naur Form or Extended Backus-Naur Form

https://bnfplayground.pauliankline.com/

Task: Define the **date grammar** (seen earlier) in BNF Playground

Compile it then use the GENERATE RANDOM button to generate valid dates.

Notice these generated dates are *syntactically* valid, but not always *semantically* valid

You can also enter test strings to *test* for valid date productions.

Command Line Apps

Command-line programs often take arguments – sometimes adhering to very complex rules, as we saw in the first lecture.

```
grep file pattern searcher
SYNOPSIS
     grep [–abcdDEFGHhIiJLlMmnOopqRSsUVvwXxZz]
          [–A num] [–B num] [–C num]
          [–e pattern] [–f file] [––binary–files=value]  ...
```

The syntax of command line apps can be defined by a grammar

See the mini-Docker example in the grammars and syntax-based testing lab for this topic

Input Syntax

Grammars are used to define whether something is a valid

- phone number
- postcode
- URL
- HTML page
- email address

and many other formats.

Domain-Specific Languages

Often, grammars will be useful to define what are called "domain-specific languages" (**DSLs**) which describe entities in a domain and things to do with them – e.g. Makefiles are an example of this.

Syntaxes are typically used to define such languages.

Syntactically well-formed Java class

And of course, every **programming language** is defined by a grammar or syntax – when we violate the syntax, the compiler tells us we've committed a "syntax error".

Syntactically well-formed Java class

```
class MyClass { }
```

Syntactically ill-formed:

```
class { MyClass  }
```

Questions

- **Q.** What's the dividing line between what we call "syntax" ("Student numbers are of the form: NNNN-NNN-NN, where `N` is a digit") and semantics ("If the first digit of the month is 1, the second can only be '0' or '1' or '2' ")?

Questions

- **Q.** What's the dividing line between what we call "syntax" ("Student numbers are of the form: NNNN-NNN-NN, where N is a digit") and semantics ("If the first digit of the month is 1, the second can only be '0' or '1' or '2' ")?
- **A.** Usually, if a rule can be described using BNF, it's called a syntactical rule; if not, it's a semantic rule.

  Languages using such rules are called **context-free languages**.

  (Why "context-free"? Because we can't have rules like the one just described, that say "*If* the previous item was '1', then *this* item can only be '0' or '1' or '2' – the grammars we use never require us to supply *contextual* information of this sort.)

Generators – Network traffic

- Being able to **generate** things that follow a syntax-like structure is extremely useful for testing.
- We can use BNF 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).

Generators – Network traffic (cont)

- Not all the validity rules for a TCP packet can be expressed in a syntactic 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 (cont)

- 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.)

File 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.

File Generator Tools

- 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.

Mutating Inputs

Motivation

- It is quite common to require a program to *reject malformed inputs*, and this property should definitely be tested as a form of stress testing.
- Malformed inputs may slip past the attention of programmers who are focused on *happy path*
- From a practical perspective, invalid inputs sometimes matter a great deal because they hold the key to unintended functionality. For example, **unhandled invalid inputs** often represent **security vulnerabilities**

Mutation operators

- When mutating grammars, the mutants are the tests and we create valid and invalid strings
- How to mutate grammar rules: (see A and O for details)
  - **Replace** term/nonterm
  - **Delete** term/nonterm
  - **Duplicate** term/nonterm

Reading: Ammann and Offutt. *Section 9.5*

Example: Generating Malformed-Inputs

A and O. Sec 9.5 Question 3

Consider the following BNF with start symbol A:

```
A ::= B"@"C"."B
B ::= BL | L
C ::= B | B"."B
L ::= "a" | "b" | "c" | ... | "y" | "z"
```

Example: Generating Malformed-Inputs (cont)

Consider the following six possible test cases from the grammar on the last page:

```
t1 = a@a.a
t2 = aa.bb@cc.dd
t3 = mm@pp
t4 = aaa@bb.cc.dd
t5 = bill
t6 = @x.y
```

For each of the six tests, state whether the test sequence is either

1. "in" the BNF, and give a derivation, or
2. sequence is "out" of the BNF, and give a mutant derivation that results in that test. Use only one mutation per test, and use it only one time per test.

Summary

Summary of Syntax-Based Testing Topics

- Motivation - What is syntax-based testing?
- Theory - Formal Grammars and Coverage Criteria
- Applications of Syntax-based testing
- Input-Space Mutation Testing
- Program-Based Mutation Testing (lecture 2)