CITS5501 Software Testing and Quality Assurance
Mutation testing

Unit coordinator: Arran Stewart

March 27, 2018

Models of software

- We've seen software modelled using graphs
  - (Can be applied to: source code, use cases, design, etc)
- We've seen how its possible to model the *input space*
  - (Effectively models software components as input/output functions)
- We'll now see models of the software *syntax*
  - As with other sorts of modelling: this can be applied at various levels

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
- 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

BNF grammars

- Let's consider BNF grammars (taken from automata theory)
- 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.

Coverage criteria

- If we're developing tests based on syntax ...
- The most common and straightforward coverage criteria: 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 expect in special cases, this will be impractical

Bounds on coverage

- Example grammer:

```
<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 format), DC will usually require an infeasibly large number of tests

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