Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 22 — March 1, 2010

Patrick Lam *version 1*

Syntax-Based Testing

We will see two applications of grammars:

1. to create inputs (both valid and invalid)
2. to modify programs (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.

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:

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

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.

We specify three coverage criteria for inputs with respect to a grammar G.

Criterion 1 Terminal Symbol Coverage *(TSC). TR contains each terminal of grammar G.*

Criterion 2 Production Coverage *(PDC). TR contains each production of grammar G.*

Criterion 3 Derivation Coverage *(DC). TR contains every possible string derivable from G.*

PDC subsumes TSC. DC often generates infinite test sets, and even if you limit to fixed-length strings, you still have huge numbers of inputs.

Mutation Testing

The second major way to use grammars in testing is mutation testing. Strings are usually programs, but could also be inputs, especially for testing based on invalid strings.

Definition 1 *Ground string: a (valid) string belonging to the language of the grammar.*

Definition 2 *Mutation Operator: a rule that specifies syntactic variations of strings generated from a grammar.*

Definition 3 *Mutant: the result of one application of a mutation operator to a ground string.*

Mutants may be generated either by modifying existing strings or by changing a string while it is being generated.

It is generally difficult to find good mutation operators. One example of a bad mutation operator might be to change all predicates to "true".

Note that 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.

More on ground strings. Mutation manipulates ground strings to produce variants on these strings. Here are some examples:

- the program that we are testing; or
- valid inputs to a program.

If we are testing invalid inputs, we might not care about ground strings.

Credit card number examples. Valid strings:

Invalid strings:

We can also create mutants by applying a mutation operator during generation, which is useful when you don't need the ground string.

NB: There are many ways for a string to be invalid but still be generated by the grammar.

Some points:

- How many mutation operators should you apply to get mutants? *One.*
- Should you apply every mutation operator everywhere it might apply? *More work, but can subsume other criteria.*

Killing Mutants

Generate a mutant m for an original ground string m_0.

Definition 4 *Test case t* kills *m if running t on m produces different output than running t on m_0.*

(The book uses a derivation D rather than a ground string m_0.)

Criterion 4 Mutation Coverage *(MC). For each mutant m, TR contains requirement "kill m".*

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

Criterion 5 Mutation Operator Coverage *(MOC). For each mutation operator* `op`*, TR contains requirement to create a mutated string m derived using* `op`*.*

Criterion 6 Mutation Production Coverage *(MPC). For each mutation operator* `op` *and each production p that* `op` *can be applied to, TR contains requirement to create a mutated string from p.*