Mutation Testing

Mutation Testing

- Mutation testing involves making syntactically valid changes to a software artifact and then testing the artifact.
- We consider grammars of software artifacts again. Grammars generate valid strings.
- We use derivations in grammars to generate invalid strings too.
- Testing based on these valid and invalid strings is called mutation testing.
- Mutation testing can be applied to almost all software artifacts: code, design, inputs and requirements.

Mutation testing: Some terms

- Ground string: A string that is in the grammar.
- Mutation operator: A rule that specifies syntactic variations of strings generated from a grammar.
- Mutant: The result of *one* application of a mutation operator.

Mutation testing: Some terms, explained

- The ground string could be a program specified using a grammar, in an input format specified using a mark-up language (which has a grammar) etc.
- Mutation operators can also be applied to a grammar, or dynamically during a derivation.
- Appropriately designed mutation operators can be applied to mutate a software artifact and test it.
- Sometimes, ground strings could be the implicit result of *not* applying a mutation operator.
  - When we mutate the inputs to a program, the program stays the same, only the inputs are changed to check if the program responds as expected to invalid inputs.

Example of mutant

Consider the grammar that we saw in the last lecture:

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

- Strings generated by the grammar: G 17 08.01.90, B 13 06.27.94
- Two valid mutants: B 17 08.01.90, G 43 08.01.90
- Two invalid mutants: 12 17 08.01.90, G 23 08.01

Mutation operators

- Mutation operators have been designed for several programming languages, formal modelling and specification languages, BNF grammars, data definition languages (like XML), query languages etc.
- Mutation operators for programming languages can be applied at unit testing and at software integration testing levels.
- We will see operators for Java and XML in this course.

Mutation operators: Additional definitions

- For a given artifact, let M be the set of mutants, each mutant $m \in M$ will lead to a test requirement.
- The testing goal in mutation testing is to kill the mutants by causing the mutant to produce a different output.
- Given a mutant $m \in M$ 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.
- The derivation D could be represented by the complete list of productions followed, or simply be the final string.

Mutation testing: Coverage criteria

- Mutation Coverage (MC): For each mutant $m \in M$, TR contains exactly one requirement, to kill m.
- The coverage in mutation equates to killing the mutants.
- The amount of coverage is usually written as a percent of mutants killed and is called the mutation score.
- Higher mutation score doesn't mean more effective testing.

Mutation testing: More coverage criteria

- When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct.
- 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 Production Coverage (MPC): For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.

Mutation coverage criteria

- Number of test requirements for mutation is somewhat difficult to quantify. It heavily depends on the sytnax of the software artifact and the considered mutation operators.
- Exhaustive mutation testing is never done, operators are available exhaustively to choose from.
- Typically, mutation testing yields more requirements than other criteria.
- Mutation testing is difficult to apply by hand, automation is also complicated.

Mutation operators: Dos and Don'ts

- Should more than one mutation operator be applied at the same time to create *one* mutant? Should a mutated string contain one mutated element or several?
  Answer: Ideally it should be a "No". Experimental and theoretical evidence indicate that it is difficult to track the program behavior when more than one operator is applied.
- Should every possible application of a mutation operator to a ground string be considered?
  Answer: Answer could be "yes", if done in a careful and systematic way, especially for mutating programs. Answer is a "No", when it is not useful.

Mutation testing: Plan ahead

- We will first learn how to define mutation operators and apply mutation testing to programs.
- Integration testing related mutation will follow.
- For input space mutation, we will discuss how to mutate inputs to create invalid inputs and briefly, understand input languages like XML and defining mutation operators over them.

COURTESY:MEENAKSHI DSOUZA,IIIT ,BANGLORE