Software Testing

Sibylle Schupp[1]

[1] Institute for Software Systems/Institut für Softwaresysteme
Hamburg University of Technology (TUHH)

Spring 2022

Lecture 11

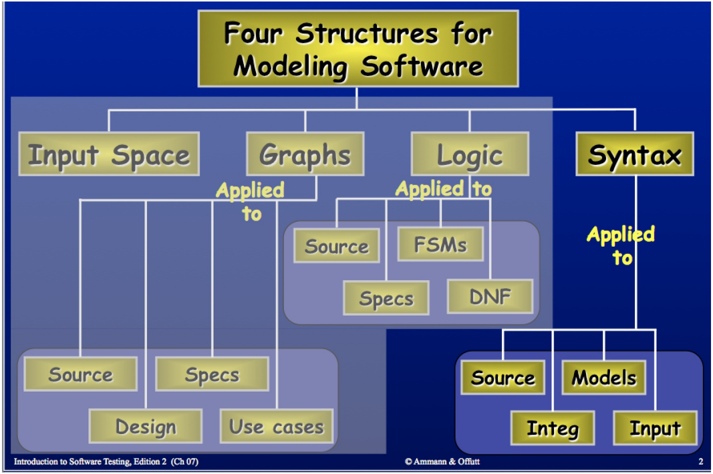
Outline

1. Syntax-based testing I

Introduction

- RIPR model: reachability, infection, propagation, revealability
    - Graph coverage: reachability
    - Logic coverage: infection
    - (Input space partitioning: independent of RIPR model)
- Model-based testing so far:
    - Input domain model
    - Graph model
    - Logic model
- Syntax-based testing
    - Propagation
    - Syntax as model

Syntax coverage



Four Structures for Modeling Software

Input Space

Graphs

Logic

Syntax

Applied to

Applied to

Applied to

Source

FSMs

Specs

DNF

Source

Specs

Design

Use cases

Source

Models

Integ

Input

Introduction to Software Testing, Edition 2 (Ch 07)

© Ammann & Offutt

2

Outline

1. Syntax-based testing I
   - Grammars and mutation
   - Program-based coverage

Using the syntax for testing

- Many artifacts follow syntax rules:
    - Programs, input descriptions, design documents, . . .
- The rules are often expressed as grammar.
    - Common grammars: regular grammars, context-free grammars
    - Theoretical foundation: automata theory
- Possible test goals
    - Cover the syntax in some way
    - Violate the syntax

Regular expressions

Definition

Let Σ be an alphabet and denote by ϵ the empty string. The set of regular expressions is defined inductively.

1. ϵ and every $a \in \Sigma$ is a regular expression.
2. If r, s are regular expressions, then their concatenation, choice, and repetition is a regular expression.
3. Every regular expression is obtained from the previous two rules.

		example ($\Sigma = \{0, 1, 2, 3, 4, r, s, x\}$)
Operators	choice	$r + s$
	sequence	rs
	repetition	r^*
Additional op.	(range)	$[0 - 3]$
	fixed rep.	x^n

Example (regular expression)

- Example: $\Sigma = \{G, B, n, s, t\}$, $(Gsn|Btn)^*$
  - Interpretation: G, B methods, commands, events; n, s, t parameters or values
- Each regular expression defines a set of strings. A string that is element of that set is said to be in the grammar (or in the language).
  - Gsn, Btn, $BtnGsn$, $BtnBtn$, ...
- A test case is a string that satisfies the regular expression.
  - Example: $\Sigma = \{G, B, 0, \ldots 9, a, \ldots z\}$, regular expression: $(G[0-9]^*[a-z]^* \mid B[0-9]^*[a-z]^*)^*$
  - $G99a, B1abc, G0aB1b, \ldots$

BNF grammars: example

Backus-Naur form

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”

Notation

- A grammar consists of a set of productions. A production is a pair (lhs, rhs), where rhs rewrites lhs. Productions with the same lhs can be combined via | and $*$.
- Terminal symbols are enclosed in quotes, all other symbols are called non-terminals. The first symbol (Stream) is the start symbol.

Using grammars

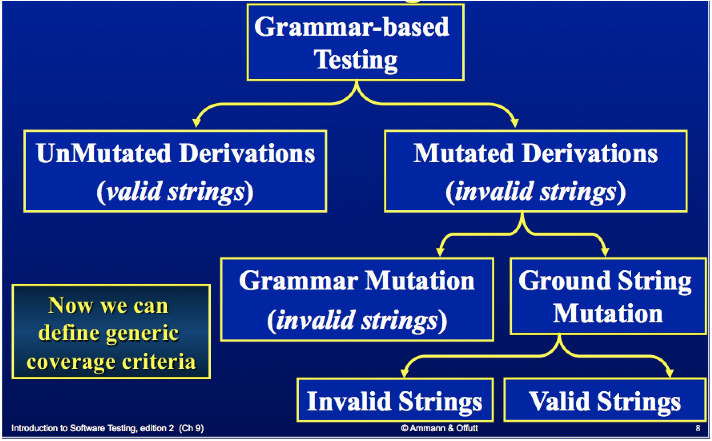
In grammar-based testing, tests are strings. A string of terminals obtained by applying a sequence of derivations is said to be in the grammar (or in the language).

Definition

A ground string is a string in the grammar.

- Recognition
  - Is a test (string) in a grammar?
    - Parsing problem
    - Useful for input validation
  - Ex.: G2508.01.90 is in the grammar
    - Proof by derivation: start with the start symbol, apply production rules to obtain a tree the leaves of which, concatenated, form that string.
- Generation: given a grammar, derive tests (strings) from it.

Classification



Grammar-based Testing

UnMutated Derivations (*valid strings*)

Mutated Derivations (*invalid strings*)

Grammar Mutation (*invalid strings*)

Ground String Mutation

Invalid Strings

Valid Strings

Now we can define generic coverage criteria

Introduction to Software Testing, edition 2 (Ch 9)

© Ammann & Offutt

8

Terminal symbol coverage and production coverage

Definition

- For terminal symbol coverage (TSC), *TR* contains each terminal symbol in the grammar.
- For production coverage (PDC), *TR* contains each production in the grammar.

Discussion

- PDC subsumes TSC.
- Grammars can be considered graphs (PDC equivalent to edge coverage.)
- Other grammar coverage criteria?

Derivation coverage

Definition

For derivation coverage (DC), *TR* contains every possible string that can be derived from the grammar.

Discussion

- Infeasible
- Compare the size of the test set for the Stream grammar
  - TSC: 13 symbols, thus max. 13 tests
  - PDS: 18 productions, thus max. 18 tests
  - DC: consider just the number of subtrees of node “action”: $2 \cdot 10^9$ derivations $=$ subtrees $=$ strings possible!
- Other criteria? What about tests that are not in the grammar?

Mutation testing: idea

- Grammars describe both valid and (implicitly) invalid strings.
- Both types can be produced by mutating a valid string.
  - Mutating valid strings can result in valid as well as invalid strings.
- Mutation testing
  - Proceed systematically, using well-defined rules
  - A.k.a. mutation analysis

Mutants and mutation operators

Recall that a ground string is a string in the grammar.

Definition

A <u>mutation operator</u> is a rule for generating syntactic variations of ground strings. A <u>mutant</u> is the result of the application of a mutation operator.

Example:

- Ground string $G2508.01.90$
  - Valid mutant $B2508.01.90$
  - Invalid mutant $F2508.01.90$

Practical issues

- Should more than one mutation operator be applied to the same string?
    - Usually not (interference), but higher-order mutants exist
- Should every possible application of an operator be considered?
    - Typically yes for program-based mutations
- For which languages can mutation operators be defined?
    - Programming languages (Fortran, . . . , Java)
    - Specification languages (NuSMV. . . .)
    - Modeling languages (UML statecharts, activity diagrams)
    - Input grammars (XML, . . .)

Killing mutants

Definition

Given a mutant m for a derivation D. A test t is said to kill m iff the output of t on D is different from the output of t on m.

Discussion

- Does the mutated ground string yield a string that exhibits different behavior?
  - “Output of t” will be interpreted in different ways.
  - Ex: D, m programs. Then, the output of the two programs is compared.
- D can be represented as list of productions or as the final string.

Mutation coverage (MC): valid strings

Definition

Let M be a set of mutants. Given a mutant $m \in M$. For mutation coverage (MC) TR contains one requirement per m, namely to kill m. The amount of mutants killed is called the mutation score.

- For valid strings, the testing goal is to kill a mutant: coverage $\sim$ killing
- Ex.: consider ground string G2509.01.90 and its mutant B2509.01.90 (valid). Assume both strings represent subroutines. A killing test finds parameters that result in different return values.

Mutation coverage: invalid strings

Definition

- For mutation operator coverage (MOC), *TR* contains for each mutation operator exactly one requirement, to create a mutated string *m* that is derived using that operator.
- For mutation production coverage (MPC), *TR* contains for each mutation operator and each production that operator can be applied to the requirement to create a mutated string *m* from that production.

- If mutation results in invalid strings, the testing goal is simply to run mutants.
- In this case, mutation operators define test requirements directly.

Example (mutation coverage)

Consider the Stream grammar from before.

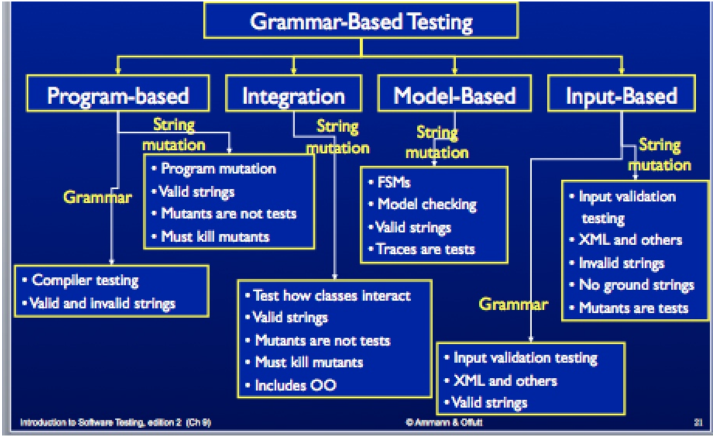
- We introduce three (non-standard) mutation operators:
  - 1: Change G to B; 2: change B to G; 3: replace digit by another digit
- Now consider the two ground strings $G2509.01.90$ and $B2106.27.94$
  - Applying the mutation operators each on the two ground strings yields, e.g., $B2509.01.90$, $G2106.27.94$, $G2309.01.90$, $B1106.27.94$ (all valid).
  - TR: find test cases that kill the four mutants
- Now assume the following 2 mutation operators: change “G” to “F”, change “B” to “C”
  - TR for MOC: apply the mutation operators to the ground strings above
  - Tests (for MOC coverage): $F2509.01.90$ and $C2106.27.94$ (all invalid)

In-class exercise

Mutation testing as gold standard

- Gold standard for comparing other test methods (see below for more)
  - More effective, more expensive
  - Number of tests depends on
    - Size of the syntactic description
    - Number of mutation operators
- Also applicable if there is no oracle available!
- Automation
  - Hard (and expensive) to apply by hand

Classification (grammar-based testing)



Grammar-Based Testing

Program-based

- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants

- Compiler testing
- Valid and invalid strings

Integration

- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

Model-Based

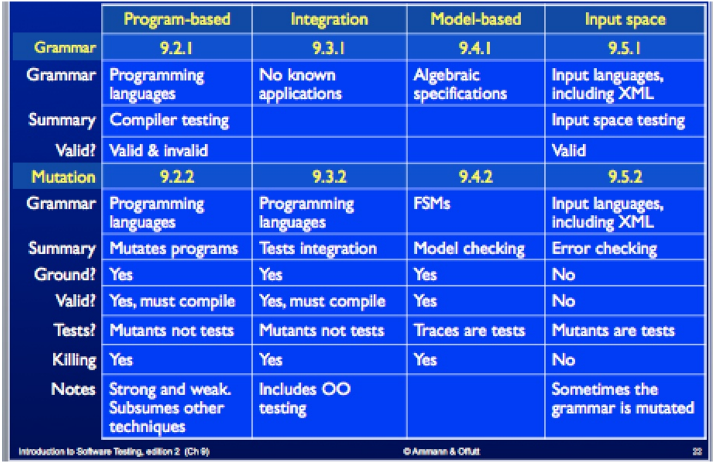
- FSMs
- Model checking
- Valid strings
- Traces are tests

Input-Based

- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests

- Input validation testing
- XML and others
- Valid strings

Overview (valid and invalid tests)



	Program-based	Integration	Model-based	Input space
Grammar	9.2.1	9.3.1	9.4.1	9.5.1
Grammar	Programming languages	No known applications	Algebraic specifications	Input languages, including XML
Summary	Compiler testing			Input space testing
Valid?	Valid & invalid			Valid
Mutation	9.2.2	9.3.2	9.4.2	9.5.2
Grammar	Programming languages	Programming languages	FSMs	Input languages, including XML
Summary	Mutates programs	Tests integration	Model checking	Error checking
Ground?	Yes	Yes	Yes	No
Valid?	Yes, must compile	Yes, must compile	Yes	No
Tests?	Mutants not tests	Mutants not tests	Traces are tests	Mutants are tests
Killing	Yes	Yes	Yes	No
Notes	Strong and weak. Subsumes other techniques	Includes OO testing		Sometimes the grammar is mutated

Introduction to Software Testing, edition 2 (Ch 9) © Ammann & Offutt 22

Outline

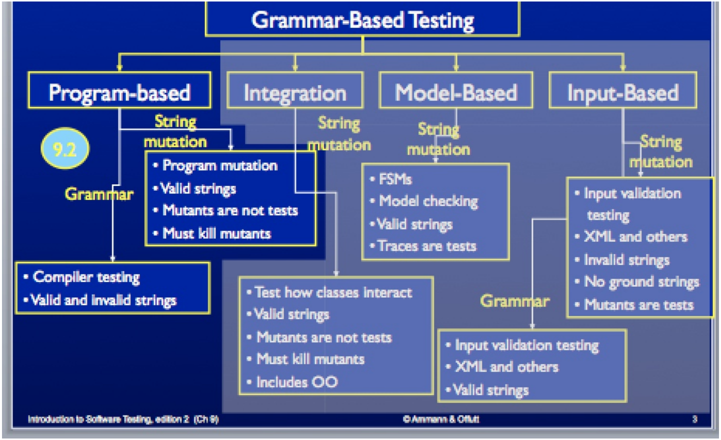
1 Syntax-based testing I

- Grammars and mutation
- Program-based coverage

Program-based grammars

- Syntax-based testing originates in program testing.
- Mutation testing
    - Commonly used for unit testing and integration testing
- BNF-based testing
    - Used for testing language-based tools, e.g., compilers

Overview (program-based grammars)



Grammar-Based Testing

Program-based

9.2

String mutation
- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants

Grammar
- Compiler testing
- Valid and invalid strings

Integration

String mutation
- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

Model-Based

String mutation
- FSMs
- Model checking
- Valid strings
- Traces are tests

Input-Based

String mutation
- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests

Grammar
- Input validation testing
- XML and others
- Valid strings

Introduction to Software Testing, edition 2 (Ch 9)

© Ammann & Offutt

3

Program-based grammars: mutations

- Ground string: the program under test
- Mutation operators modify the ground string, create mutant programs.
    - Mutants must compile (valid strings).
    - Mutants are not tests (test requirements), but are used to define TRs.
- Tests must "make a difference." We refine the previous definition:

Definition

Given a mutant m for a ground string program P. A test t is said to kill m iff the output of t on P is different from the output of t on m.

- Different mutation operators are defined for different programming languages and different testing goals.

Classes of mutants

From a testing purpose, not all mutants are desirable. One distinguishes:

- Dead mutant
    - Killed by a test case
- Stillborn mutant
    - Syntactically illegal
- Trivial mutant
    - Killable by almost any test case
- Equivalent mutant
    - Impossible to kill by any test (same behavior as original)

Example (program-based mutation)

```
int Min (int A, int B)
{
      int minVal;
      minVal = A;
      if (B < A)   {
          minVal = B;
      }
      return (minVal);
} // end Min
```

- What are reasonable mutations?

Example (mutants)

```
int Min (int A, int B)
{
     int minVal;
     // minVal = A;          // original
     minVal = B;             // replace variable by another variable
     // if (B < A)           // original
     if (B > A)              // replace operator
     if (B < minVal)         // replace variable by another variable
     {
        minVal = B;
        Bomb();             // insert immediate runtime failure
        minVal = A;         // replace variable by another variable
        minVal = failOnZero(B)   // insert runtime failure
     }                           // if B==0
     return (minVal);
} // end Min
```

- 6 mutants, each represents a separate program (6 diff. programs)
- "runtime failure": only if program point reached

Strong and weak killing

Definition

Let M be a set of mutants.

- Given a mutant $m \in M$ that modifies a location l in a program P. A test t is said to strongly kill m iff the output of t on P is different from the output of t on m.
- Given a mutant $m \in M$ that modifies a location l in a program P. A test t is said to weakly kill m iff the state of the execution of P on t is different from the state of the execution of m on t immediately after l.

- RIP: Weakly killing satisfies reachability and infection, but not propagation.

Weak mutation

For weak mutation coverage (WMC), TR contains for each $m \in M$ exactly one requirement, to weakly kill m.

- Easier in practice than strong mutation: less analysis
- In practice: most test sets that weakly kill all mutants also strongly kill (many of) them. (Or so we hope.)

Example (weak mutation)

```
int Min (int A, int B) // the first mutant
{
    int minVal;
    // minVal = A;
    minVal = B;     // (*) replace variable by another variable
    if (B < A)
    {
        minVal = B;
    }
    return (minVal);
} // end Min
```

- A weakly killing test: A=5, B=3
  - State after (*) infected (different value for minVal)
- Conditions for RIP
  - Reachability: line 2 always reachable
    Infection: $A \neq B$ (minVal has different value),
    Propagation $\neg(B < A)$ and $A \neq B$ (Infection), thus $B > A$.
- For $B \leq A$, weak kills of this mutant do not also kill strongly

Example (equivalent mutant)

```
int Min (int A, int B) // third mutant
{
     int minVal;
     minVal = A;
     // if (B < A)      // original
     if (B < minVal)   // (*) replace variable by another variable
     {
        minVal = B;
     }
     return (minVal);
} // end Min
```

- The mutant is equivalent
- Argument: by substitution
- No infection: state after (*) not infected;
  in ($B < minVal$) both values $B, minVal$ unchanged

Example (strong versus weak mutation)

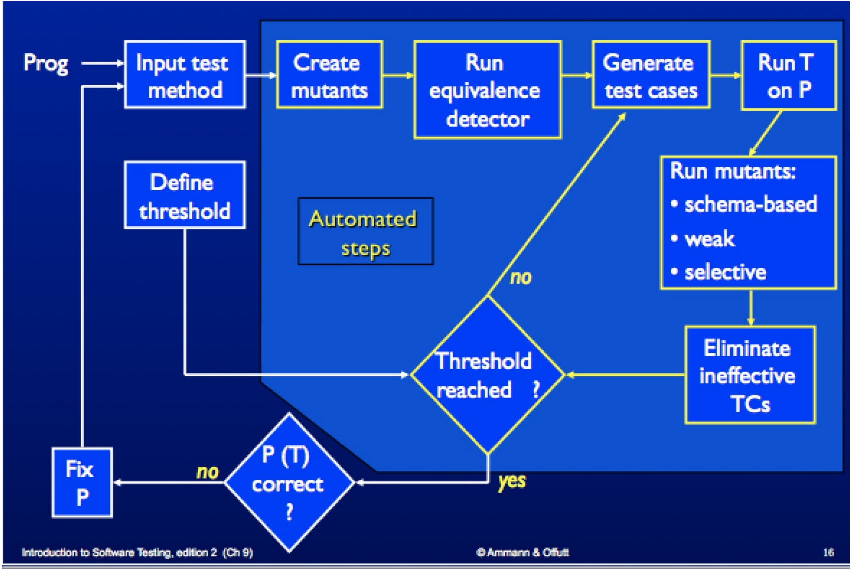
```
boolean isEven (int X)
{
    if (X < 0) {
        // X = 0 - X; // original
        X = 0;        // replace variable by constant
    }
    if (double) (X/2) == ((double X)/ 2.0)
        return true;
    else
        return false;
} // end isEven
```

- Consider test X = -6.
- RIP conditions?
    - Reachability: $X < 0$
    - Infection $X \neq 0$. Test weakly kills mutant.
    - Propagation: test does not strongly kill mutant.
- Condition for strong killing?

In-class exercise

```
public static int findVal(int[] numbers, int val) {   // pre: val>0
  int findVal = -1;
  // for (int i=1; i<numbers.length; i++) { // original
  for (int i=0; i<numbers.length; i++) {     // mutant
      if (numbers[i] == val) {
          findVal = i;
      }
  }
  return (findVal);
}
```

Work flow


Prog

Input test method

Create mutants

Run equivalence detector

Generate test cases

Run T on P

Run mutants:
- schema-based
- weak
- selective

Eliminate ineffective TCs

Threshold reached ?

no

yes

Define threshold

Automated steps

P (T) correct ?

no

Fix P

Introduction to Software Testing, edition 2 (Ch 9) © Ammann & Offutt 16

References

- AO, 9.1, (9.2.2)