Software Testing

Sibylle Schupp[1]

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

Spring 2022

Lecture 12

Outline

Recap

- Mutation testing (mutation analysis): generate (valid or invalid) strings from a grammar.
- Assume the mutant is a valid string. Mutation coverage (MC): for each mutant, TR contains one requirement, to kill that mutant.
- Generation of mutants: based on mutation operators.

Effective mutation operators

Definition

Let M be a set of mutation operators and $O \subset M$. If tests that are designed to kill mutants created by O also kill (with a high chance) mutants created by all remaining mutation operators, then O defines an effective set of mutation operators.

- Effective in practice: insertion plus modification of unary/binary operators
- Additional design considerations:
  - Mutation operators should mimic typical programmer mistakes.
- At method level, mutation operators for different programming languages are similar.

Mutation operators for Java (1/6)

- Absolute Value Insertion (ABS): Each arithmetic expression (and subexpression) is modified by the functions `abs()`, `negAbs()`, and `failOnZero()`. Ex.:

```
a = m * (o + p);
// 3 mutants (out of 5 * 3 = 15)
// a = abs (m * (o + p));
// a = m * abs ((o + p));
// a = failOnZero (m * (o + p));
```

- Arithmetic Operator Replacement (AOR): Each occurrence of one of the arithmetic operators (`+,-,*,/,%`) is replaced by each of the other operators. In addition, each arithmetic operator is replaced by the special mutation operators `leftOp` and `rightOp`. Ex.:

```
a = m * (o + p);
// mutants (examples)
// a = m + (o + p);
// a = m - (o + p);
// a = m leftOp(o + p);
```

Mutation operators for Java (2/6)

- Relational Operator Replacement (ROR): Each occurrence of one of the relational operators (`<, <=, >, >=, =, !=`) is replaced by each of the other operators and by the special operators `falseOp` and `trueOp`.

```
if (X <= Y)
// mutants (examples)
// if (X > Y)
// if (X < Y)
// if (X falseOp Y)
```

- Conditional Operator Replacement (COR): Each occurrence of one of the logical operators `&&, ||, &, |, ^` (with and without conditional evaluation) is replaced by each of the other operators and the special mutation operators `falseOp`, `trueOp`, `leftOp`, and `rightOp`. Ex.:

```
if (X <= Y && a > 0)
// mutants (examples)
// if (X <= Y || a > 0)
// if (X <= Y leftOp a > 0)
```

Mutation operators for Java (3/6)

- Shift Operator Replacement (SOR): Each occurrence of one of the shift operators (`<<`, `>>`, `>>>`) is replaced by each of the other operators and by the special operator `leftOp`.

```
byte b = (byte) 16;
b = b >> 2;
// mutants (examples)
//   b = b << 2;
//   b = b leftOp 2;
```

- Logical Operator Replacement (LOR): Each occurrence of one of the logical operators (bitwise and, bitwise or, exclusive or) is replaced by each of the other operators. In addition, each logical operator is replaced by the special mutation operators `leftOp` and `rightOp`. Ex.:

```
int x = 60; int b = 13; int c = a & b;
// mutants (examples)
// int c = a | b;
// int c = a rightOp b;
```

Mutation operators for Java (4/6)

- Assignment Operator Replacement (ASR): Each occurrence of one of the assignment operators

  `=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=`

  is replaced by each of the other operators.

```
a = m * (o + p);
// mutants (examples)
//   a += m * (o + p);
//   a -= m * (o + p);
```

- Unary Operator Insertion (UOI): Each unary operator (arithmetic `+`, arithmetic `-`, conditional `!`, logical `~`) is inserted in front of each expression of the correct type.

```
a = m * (o + p);
// mutants (examples)
//   a = m * -(o + p);
//   a = -(m * (o + p));
```

Mutation operators for Java (5/6)

- Unary Operator Delete (UOD): Each unary operator (arithmetic +, −, logical negation !, bitwise complement ~) is deleted.

```
if (!(X <= Y) && !Z)
// UOD yields 2 mutants:
//   if (X <= Y) && !Z)
//   if (!(X <= Y) && Z)
```

- Scalar Variable Replacement (SVR): Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.

```
a = m * (o + p);
// mutants (examples)
//   o = m * (o + p);
//   a = o * (o + p);
//   a = m * (o + o);
//   p = m * (o + p);
```

Mutation operators for Java (6/6)

- Bomb Statement Replacement (BSR): Each statement is replaced by a special `Bomb()` function.

```
a = m * (o + p);
// mutant
//   Bomb()
```

Comparing coverage criteria

We modify the previous definition of subsumption.

- Idea: each coverage criterion defines test requirements. With each requirement associate a mutant that will be killed only by tests that satisfy the requirement.
  - A coverage criterion is satisfied iff all mutants are killed that are associated with the requirements for that criterion.
  - The mutation operator(s) that ensure the coverage of a criterion yield(s) the criterion.
- We say that mutation testing subsumes a criterion if there exist one or more mutation operators that yield that criterion.

Graph coverage

- Assume weak mutation coverage.
  - Def.: Given a program P, a test t, and a mutant m that modifies location l in P. Then t weakly kills m if the state of the execution of P is different from the state of the execution of m directly after l.
- Node coverage (statement coverage)
  - The BSR (bomb statement replacement) operator yields node coverage.
  - Insert a "bomb()" statement before each node. Killing each mutant requires a set of a tests so that each statement is reached.
- Edge coverage (branch coverage)
  - The ROR (relational operator replacement) operator yields edge coverage.
  - Example: "if (X <= Y)". For edge coverage, then-branch and else-branch need to be taken. Associate:
    - Mutant 1: "if (X falseOp Y)" (= if false)
    - Mutant 2: "if (X trueOp Y)" (= if true)
  - Kill mutant 1 with X, Y such that $X \leq Y$; kill mutant 2 with X, Y such that $X > Y$.
    - Killing the mutants requires a test set with edge coverage.

Graph coverage (cont'd)

- Why not use strong mutation coverage?
  - Recall: The output of a test t on a program P is different from the output of t on a mutant m.
- Problem: strong mutation too strict. Ex. (edge coverage):

```
int min(int A, int B) {
  int minVal;                    // edge coverage &
  minVal = A;                    // associated mutants as before:
  if (B<A) {                      if (B falseOp A)
                                  if (B trueOp A)
    minVal = B;
  }
  return minVal;
}
```

  - Consider the test set {(A=3,B=3), (A=4,B=3)}.
    Clear: test set has edge coverage.

Graph coverage (cont'd)

```
int min(int A, int B) {
  int minVal;                    // edge coverage &
  minVal = A;                    // associated mutants as before:
  if (B<A) {                      if (B falseOp A)
                                  if (B trueOp A)
    minVal = B;
  }
  return minVal;
}
```

- The test set {(3,3), (4,3)} does not strongly kill all associated mutants.

Mutant	Output	Test	Output	
if B falseOp A	A	A=4,B=3	B	$4 \neq 3$, mutant killed
if B trueOp A	B	A=4,B=3	B	$B = B$, mutant not killed
		A=3,B=3	A	$A = B$, mutant not killed

- For *strong* mutation coverage, ROR does not subsume edge coverage.
- Not surprising, though: strong mutation coverage formulates a global property, while edge coverage is formulated per branch.

Graph coverage (data-flow coverage)

- All-defs data-flow coverage (ADC)
  - Recall the notation: Let n be a node, v be a variable $\in$ def(n).
    - A DU path $du(n, v)$ is a simple path from n to a use(n) that is def-clear with respect to v.
    - A def-path set is the set of all DU paths $S = du(n, v)$ for n and v.
  - Recall ADC: for each def-path set $S = du(n, x)$, TR contains at least one path.
- Assume strong killing. The statement deletion operator yields ADC coverage.
  - Let s be a definition statement of variable v. Let m be the mutant that deletes s. Let t be a test that reaches s, thus weakly kills m (deleting a definition implies infection). If t also strongly kills m (propagation), then:
    1. Case 1: v is an output variable ("uses"). Then m killed implies that t is a def-clear path from node s to the output.
    2. Case 2: v is used at some later point without redefinition (and then causes an incorrect output state). Again, t must be a def-clear path.

Logic coverage

- Clause coverage
  - Def.: Let c be a clause. Then TR contains two requirements: c evaluates to true and c evaluates to false.
- ROR, COR, LOR yield clause coverage.
- Ex.: $x \leq y$ && $a \geq 0$.

mutant	killing test	state (orig)	state (mutant)
false && $a \geq 0$	(true, true)	true	false
true && $a \geq 0$	(false, true)	false	true
$x \leq y$ && false	(true, true)	true	false
$x \leq y$ && true	(true, false)	false	true

- For each mutated clause c, the killing test sets $\neg c$. But ROR,COR,LOR replace each clause with both true and false.
- Thus, mutation (with the operators above) subsumes clause coverage.

Logic coverage (cont'd)

- General active clause coverage
  - Recall: A major clause c_i of a predicate p determines p if the minor clauses c_j of p, $j \neq i$, have values so that changing the truth value of c_i changes the truth value of p.
  - GACC: For each major clause c and predicate p choose minor clauses so that c determines p. TR contains two requirements: c evaluates to true and c evaluates to false. The values for the minor clauses need not be the same.
- Call p_c the predicate for the test of major clause c. Associate the mutant that replaces clause c of p by "true", $p_{c=\text{true}}$, and the mutant $p_{c=\text{false}}$.
  - Then, the mutant $p_{c=\text{true}}$ is killed by a test that causes p_c and $p_{c=\text{true}}$ to have different values. But then, c determines p.
  - Similarly for $p_{c=\text{false}}$.
- Thus, ROR, COR, LOR yield GACC.

No subsumption

Some coverage criteria cannot be subsumed by any set of mutation operators.

- Combinatorial criteria, because we assume that each mutation operator is applied once (to each subexpression).
- RACC, CACC (restricted, correlated active clause coverage), because they impose requirements on pairs of tests (of minor clauses)

Historically first: mutation in FORTRAN

Source: A. J. Offutt and W. M. Craft, Using Compiler Optimization Techniques to Detect Equivalent Mutants, STVR 94

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	`DO` statement end replacement
DSA	`DATA` statement alterations
GLR	`GOTO` label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	`RETURN` statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

Table 1: **Mothra Mutation Operators for Fortran 77.**

Programming Languages

Source: Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing", IEEE TSE 37 (5), 649-678 (covers 1970–2009)

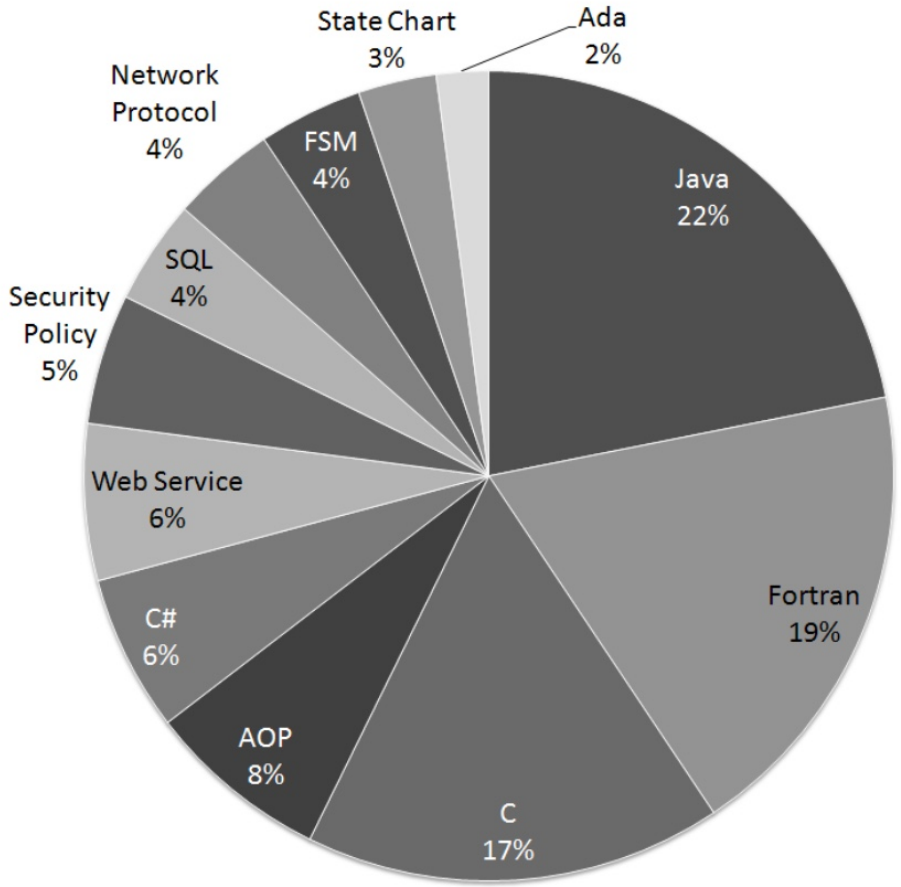


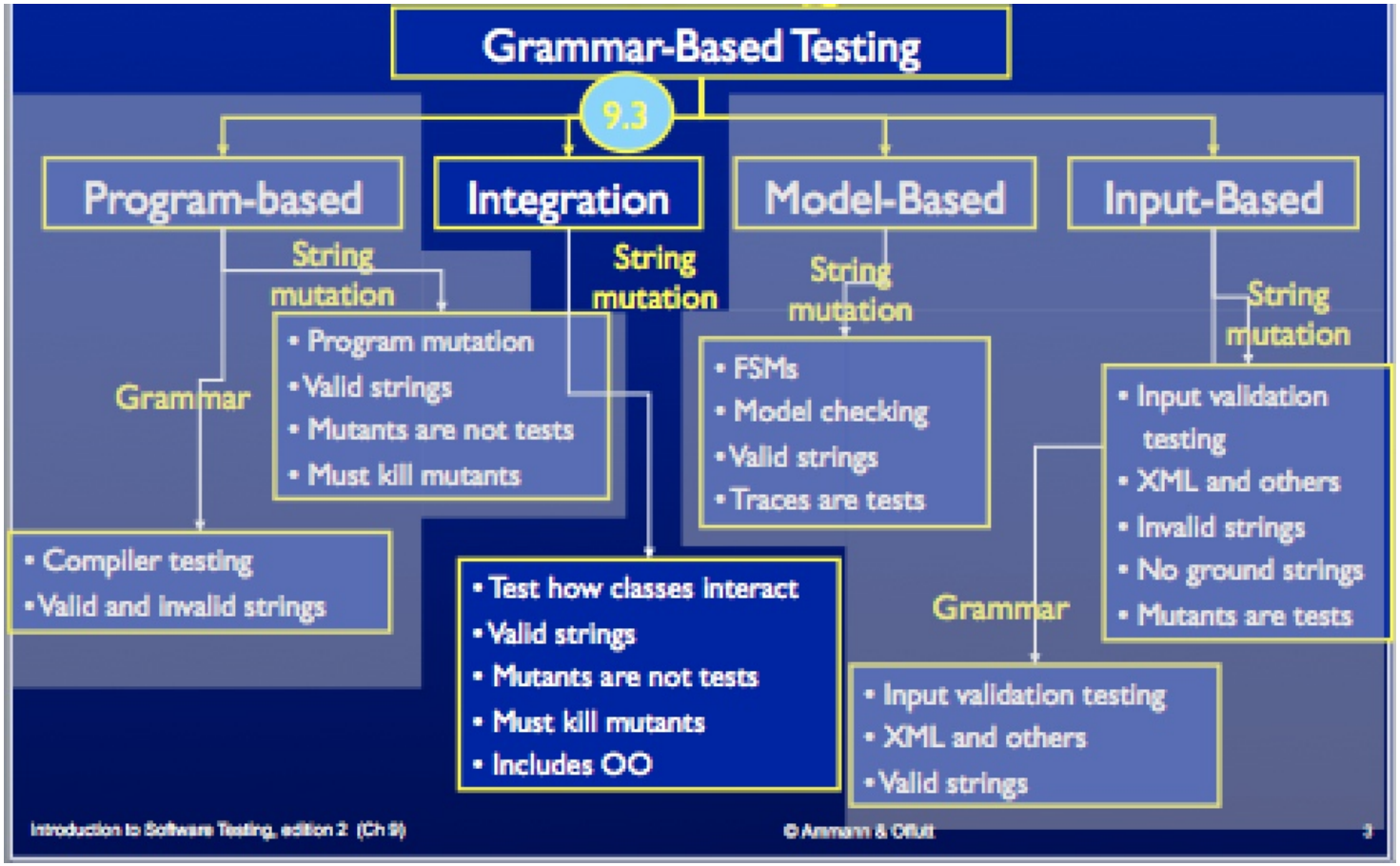
Fig. 6. Percentage of publications addressing each language to which Mutation Testing has been applied

Outline

Integration testing and object-oriented testing

- Testing integration of classes, packages, components
  - Coupling
- Tests features unique to object-oriented languages
  - Inheritance, dynamic binding, polymorphism

Grammar-based testing: integration



Integration mutation

- Integration mutation focuses on mutating the connections between components.
  - Also called interface mutation
- Often mimics mismatch of assumptions between caller and callee.
  - Caller assumes kilometers, callee returns miles.
  - Callee assumes initialized variable, caller provides uninitialized.
  - General: pre- and post-condition mismatch.

Types of mutation operators

```
int f(...) {
    ...
    ret = g(b,c);
    ..
}
```

- Change a calling method by modifying values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying values that enter and leave a method.
  - Parameters, class and package variables
- Change a called method by modifying return statements from the method.

Integration mutation operators

IPVR	Integration Parameter Variable Replacement Each parameter in a method call is replaced by each other variable in the scope of the method call (of compatible type)
IUOI	Integration Unary Operator Insertion Each expression in a method call is modified by inserting all possible unary operators in front and behind it
IPEX	Integration Parameter Exchange Each parameter in a method call is exchanged with each parameter of compatible types in that method call
IMCD	Integration Method Call Deletion Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value
IREM	Integration Return Expression Modification Each expression in each return statement in a method is modified by applying the UOI and AOR operators

Example (integration operators)

- Example IPVR (integration parameter variable replacement)

```
MyObject a, b;
callMethod (a);
// mutant
//    callMethod (b);
```

- Example IUOI (integration unary operator insertion)

```
callMethod (a);
// mutant
//    callMethod (a++);
//    callMethod (++a);
//    callMethod (a--);
```

- Example IPEX (integration parameter exchange)

```
Max(a,b)
// mutant
//    Max(b,a)
```

Example (integration operators)

- Example IMCP (integration method call deletion)

```
X = Max(a,b)
// mutant
//   X = new Integer (0);
```

- Example IREM (integration return expression modification)

```
int callMethod () {
    return a + b;
// mutant
//   return ++a + b;
//   return a - b;
```

Outline

Object-oriented mutation

- Object-based features
  - Information hiding, encapsulation (protecting design from parts of the implementation)
  - Access control (e.g., Java: private, protected, public, package)
  - Overloading
- Object-oriented features
  - Inheritance, dynamic binding, polymorphism

In-class exercise

```
class A {
    public void Foo() { System.out.println("A::Foo()"); }

}
class B extends A  {
    public void Foo() { System.out.println("B::Foo()"); }
}
class Test {
    public static void main(String[] args)
    {
        A a;
        B b;
        a = new A();
        b = new B();
        a.Foo();   // output --> "A::Foo()"
        b.Foo();   // output --> "B::Foo()"

        a = new B();
        a.Foo();   // output -->  ??
    }
}
```

OO features

- Method overriding
  - A child class declares an object or method with a name that is already declared in an ancestor class
  - vs. overloading: different constructors or methods in the same class have the same name.
- Variable hiding
  - A variable in a child class has the same name and type of an inherited variable
- Class constructors
- Polymorphism
  - Polymorphic attribute: An object reference that can take on various types.
  - Polymorphic method: A method that can accept parameters of different types because it has a parameter that is declared of type superclass.

Mutation operators in MuJava

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess modifier **c**hange
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition change
	IOR	**O**verriding method **r**ename
	ISI	**super** keyword **i**nsertion
	ISD	**super** keyword **d**eletion
	IPC	Explicit call to a **p**arent's **c**onstructor deletion
Polymorphism	PNC	**n**ew method call with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter variable **d**eclaration with child class type
	PCI	Type **c**ast operator **i**nsertion
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PRV	**R**eference assignment with other comparable **v**ariable
	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
Java-Specific Features	JTI	**this** keyword **i**nsertion
	JTD	**this** keyword **d**eletion
	JSI	**static** modifier **i**nsertion
	JSD	**static** modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	Java-supported **d**efault **c**onstructor deletion
	EOA	**R**eference **a**ssignment and content assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	EAM	**A**cessor **m**ethod change
	EMM	**M**odifier **m**ethod change

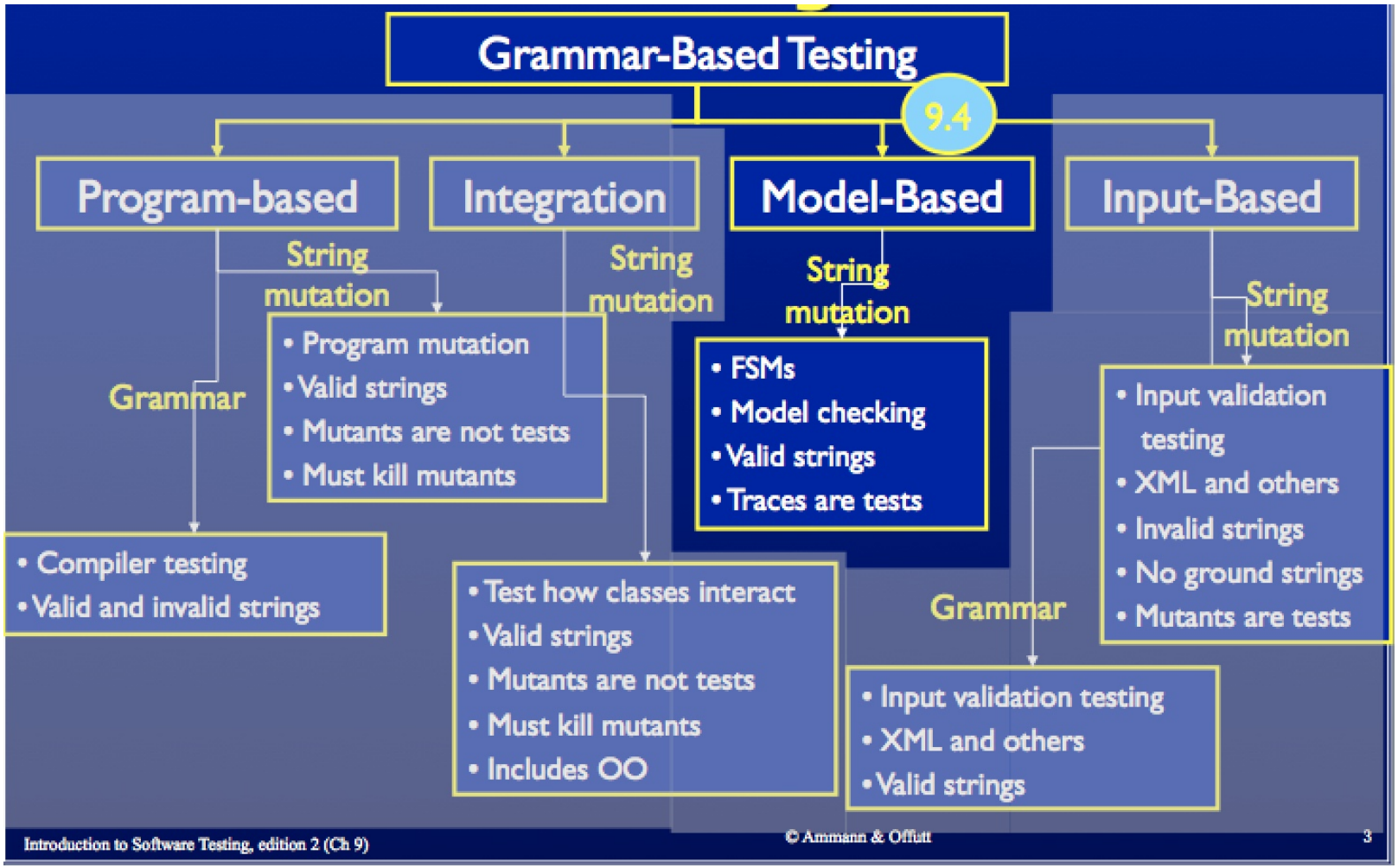
Outline

Model-based grammars

Model-based grammars include specification-based grammars as well as design notations.

- Formal specification languages
  - OCL, SMV, Z, ...
- Informal specification languages
- Design notations
  - Statecharts, FSMs, UML

Grammar-based testing: model-based testing



Model checking

- Verification method (software, hardware)
  - Sound
  - Automated
- Models are (finite) state machine.
- What is checked?
  - Input: a model M and a property ϕ (logic formula)
  - Task: does M have property ϕ ("satisfy formula ϕ")?
  - Output: yes, otherwise counterexample
- Model checking algorithms
  - (Exhaustive) searches through the state space of the model.

Example (NuSMV)

```
MODULE main

VAR
    x : boolean;
    y : boolean;
ASSIGN
   init (x) := FALSE;
   init (y) := FALSE;

   next (x) := case
     !x & y : TRUE;
     !y     : TRUE;
     x      : FALSE;
     TRUE   : x;
   esac;

   next (y) := case
     x & !y : FALSE;
     x & y  : y;
     !x & y : FALSE;
     TRUE   : TRUE;
   esac;
```

NuSMV

http://nusmv.fbk.eu

1. SMV is the name of a specification language and of a model checker.
2. NuSMV is an open-source implementation of SMV.
3. A NuSMV model defines states and transitions.
   - States are introduced through variable declarations.
   - The state space is the Cartesian product of the value ranges of all variables.
   - Initial states can be explicitly defined, by restricting values of variables.
   - Transitions are defined by "next" commands.
4. NuSMV either simulates the model or verifies its properties.
   - In simulation mode, it produces "traces" (paths) through the state space.
   - In verification mode, it searches through the state space. If a property is not verified, it returns a counter example in form of a trace.

Simulation in NuSMV

```
NuSMV > simulate -i -a -k 6
******** Simulation Starting From State 1.1 ********
*************** AVAILABLE STATES *************
  ========== State ==========
  x = TRUE
  y = TRUE
There's only one available state. Press Return to Proceed.
Chosen state is: 0
*************** AVAILABLE STATES *************
  ========== State ==========
  0) ------------------------------
  x = FALSE
  y = TRUE
*************** AVAILABLE STATES *************
  ========== State ==========
  0) ------------------------------
  x = TRUE
  y = FALSE
*************** AVAILABLE STATES *************
  ========== State ==========
  x = TRUE
  y = FALSE
```

Using model checking for FSM mutation testing

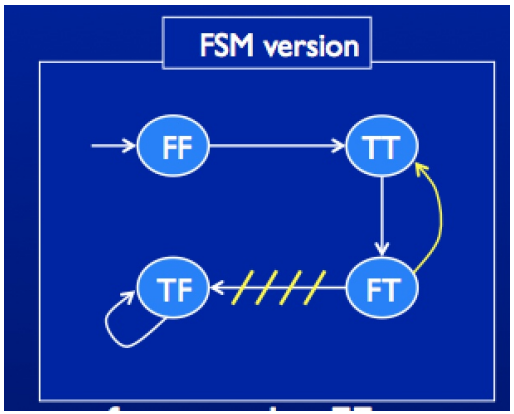
- As before:
  - Express a FSM as a model (in the sense of a model checker).
  - Devise appropriate mutation operators: Constant replacement operator, LOR, ...
  - Generate a set of mutants of the FSM by using mutation operators
  - Find for each mutant a test that kills it
- New:
  - Find a test for killing the mutant through model checking.
    - What is a killing test? A trace that is possible in the original FSM, but not in the mutated FSM.

Mutations and test cases

- Consider the constant replacement operator
  - Changes a constant to another constant
  - Ex.: case "next(y)"
    - original: $!x\&y$: *false*
    - mutant: $!x\&y$: *true*
- Killing mutants
  - Need a sequence of states ("trace") that is allowed in the original machine, but not in the mutant.
  - If mutation would *not* change the original FSM, the model checker could prove the behavior it specifies. Formulate mutation as "property."
    - Ex.: "whenever $!x\&y$, then y true in the next transition"
  - Ask the model checker to verify the mutation-motivated property for the original FSM. Two possible outcomes:
    - A counterexample (a path that violates the property) = killing test
    - No counterexample. Then the mutant is equivalent.

Counterexamples

- Mutation:



- Counterexample: test case (path) FF-TT-FT-TF kills the mutant.
- If no sequence is produced, mutant is equivalent
  - Detection of equivalent mutants decidable for FSMs (!)
- Technically: formulate mutation in a logic language, here the temporal logic CTL:
  - Ex.: $AG((!x\&y) -> AXy = TRUE)$
    "whenever $!x\&y$, then y true in the next transition"

Mutant (NuSMV)

```
MODULE main
VAR
     x : boolean;
     y : boolean;
ASSIGN
    init (x) := FALSE;
    init (y) := FALSE;

    next (x) := case
      !x & y : TRUE;
      !y     : TRUE;
      x      : FALSE;
      TRUE   : x;
    esac;

    next (y) := case
      x & !y : FALSE;
      x & y  : y;
      !x & y : FALSE;                  -- to mutate: x & !y : TRUE
      TRUE   : TRUE;
    esac;
AG ((!x & y) -> AX (y = TRUE))         -- mutation
```

Using NuSMV for FSM testing

```
NuSMV > go
NuSMV > check_property
-- specification AG ((!x & y) -> AX y = TRUE)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    x = FALSE
    y = FALSE
  -> State: 1.2 <-
    x = TRUE
    y = TRUE
  -> State: 1.3 <-
    x = FALSE
  -> State: 1.4 <-
    x = TRUE
    y = FALSE
NuSMV >
```

Summary (model-based grammars)

- Model checking: growing importance
- FSMs can be encoded in model checkers
  - Mutation expressed as property that should hold in the original FSM
  - Model checking is used to find paths (traces) that violate the property
- Using model checkers for mutation-based testing of FSMS, equivalent mutants can be detected automatically.

References

- AO, 9.2.2, 9.3, 9.4
- Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing”, IEEE Transactions on Software Engineering 37 (5), 649-678