

Introduction to Software Testing

Chapter 5.4

Model-Based Grammars

Paul Ammann & Jeff Offutt

<http://www.cs.gmu.edu/~offutt/softwaretest/>

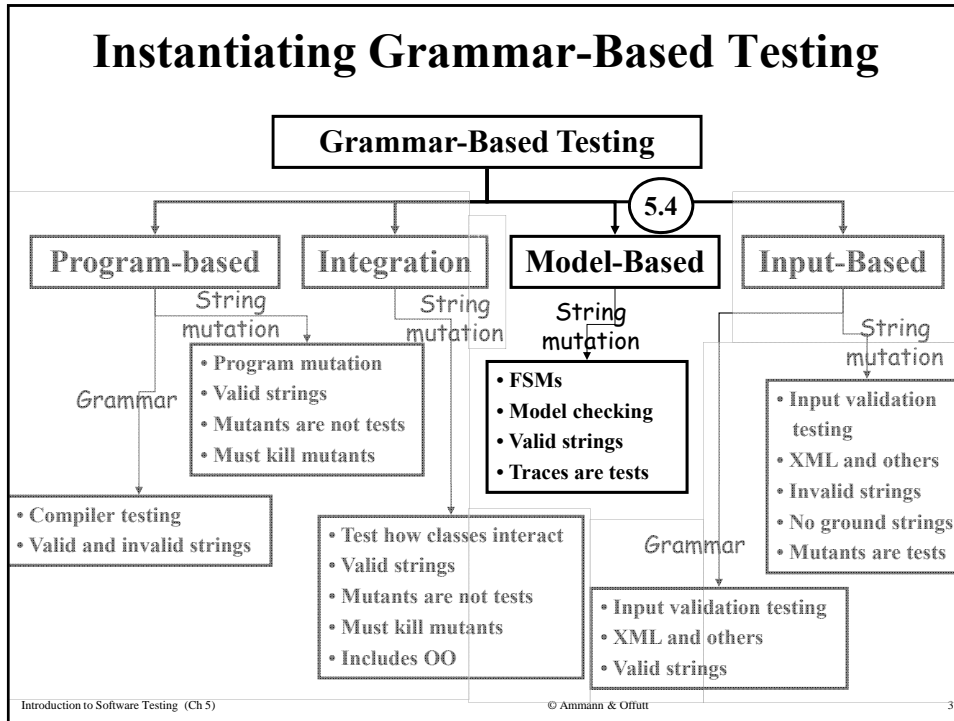
Model-based Grammars

Model-based

Languages that describe software in abstract terms

- **Formal specification languages**
 - Z, SMV, OCL, ...
- **Informal specification languages**
- **Design notations**
 - Statecharts, FSMs, UML notations
- **Model-based languages are becoming more widely used**

Instantiating Grammar-Based Testing



BNF Grammar Testing (5.4.1)

- Terminal symbol coverage and production coverage have only been applied to algebraic specifications
- Algebraic specifications are not widely used
- This is essentially research-only, so not covered in this book

Specification-based Mutation (5.4.2)

- A finite state machine is essentially a graph G
 - Nodes are states
 - Edges are transitions
- A *formalization* of an FSM is:
 - States are implicitly defined by declaring variables with limited range
 - The *state space* is then the Cartesian product of the ranges of the variables
 - *Initial states* are defined by limiting the ranges of some or all of the variables
 - *Transitions* are defined by rules that characterize the source and target of each transition

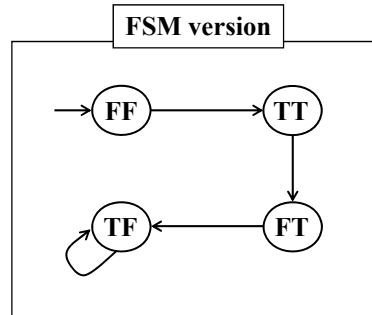
Example SMV Machine

```
MODULE main
#define false 0
#define true 1
VAR
  x, 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;
```

- Initial state : (F, F)
- Value for x in next state:
 - if x=F and y=T, next state has x=T
 - if y=F, next state has x=T
 - if x=T, next state has x=F
 - otherwise, next state x does not change
- Value for y in next state:
 - if (T, F), next state has y=F
 - if (T, T), next state y does not change
 - if (F,T), next state has y=F
 - otherwise, next state has y=T
- Any ambiguity in SMV is resolved by the order of the cases
- “true : x” corresponds to “default” in programming

Example SMV Machine

```
MODULE main
#define false 0
#define true 1
VAR
  x, 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;
```



- **Converting from SMV to FSM is mechanical and easy to automate**
- **SMV notation is smaller than graphs for large finite state machines**

Using SMV Descriptions

- **Finite state descriptions can capture system behavior at a very high level – suitable for communicating with end users**
- **The verification community has built powerful analysis tools for finite state machines expressed in SMV**
- **These tools produce explicit evidence for properties that are not true**
- **This “evidence” is presented as sequences of states, called “counterexamples”**
- **Counterexamples are *paths* through the FSM that can be used as test cases**

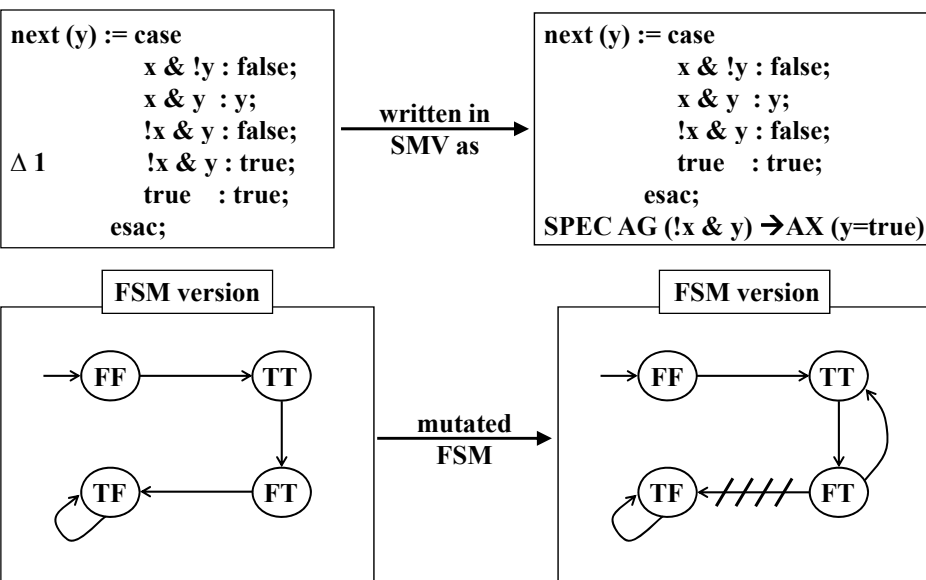
Mutations and Test Cases

- Mutating FSMs requires mutation operators
- Most FSM mutation operators are similar to program language operators

Constant Replacement operator:

- changes a constant to each other constant
- in the *next(y)* case: *!x & y : false* is mutated to *!x & y : true*
- To kill this mutant, we need a sequence of states (a path) that the original machine allows but the mutated machine does not
- This is what model checkers do
 - Model checkers find counterexamples – paths in the machine that violate some property
 - Properties are written in “temporal logic” – logical statements that are true for some period of time
 - *!x & y : false* has different result from *!x & y : true*

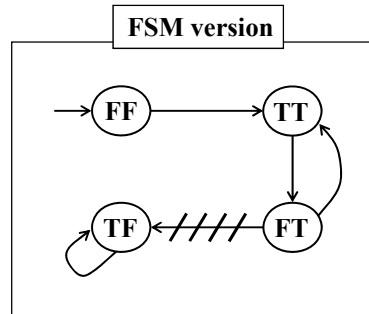
Counter-Example for FSM



Counter-Example for FSM

- The model checker should produce:

```
/* state 1 */ { x = 0, y = 0 }  
/* state 2 */ { x = 1, y = 1 }  
/* state 3 */ { x = 0, y = 1 }  
/* state 4 */ { x = 1, y = 0 }
```



- This represents a test case that goes from nodes FF to TT to FT to TF in the original FSM
 - The last step in the mutated FSM will be to TT, killing the mutant
- If no sequence is produced, the mutant is equivalent
 - *Equivalence is undecidable for programs, but decidable for FSMs*

Model-Based Grammars Summary

- Model-checking is slowly growing in use
- Finite state machines can be encoded into model checkers
- Properties can be defined on FSMs and model checking used to find paths that violate the properties
- No equivalent mutants
- Everything is finite