Model-Based Grammars

Idaho State University | Computer Science

Isaac Griffith

CS 4422 and CS 5599
Department of Computer Science
Idaho State University

ROAR

Outcomes

At the end of Today's Lecture you will be able to:

- Understand the basics of FSMs
- Understand how FSMs can be mutated
- Understand how FSM mutation can be used in testing

Inspiration

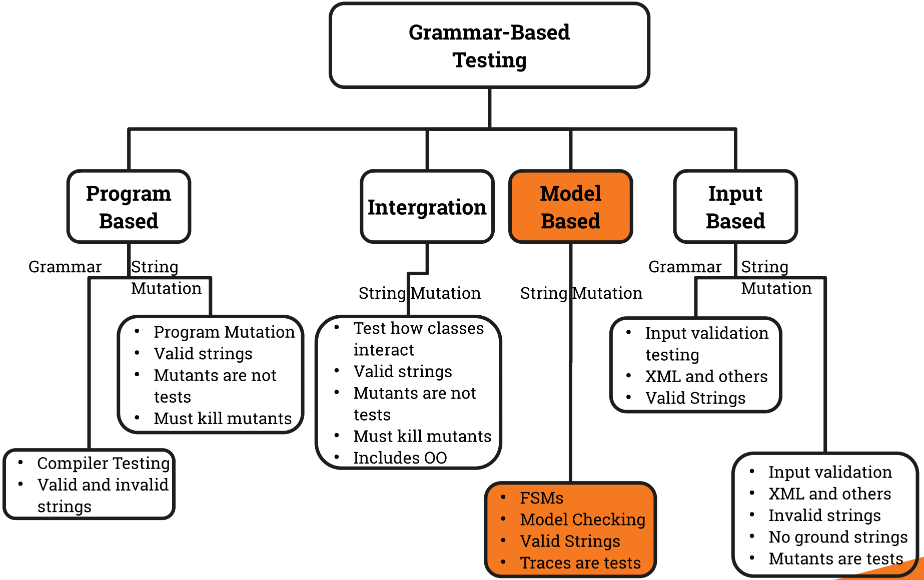
"More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded – indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest." – Boris Beizer

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

- 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

- 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

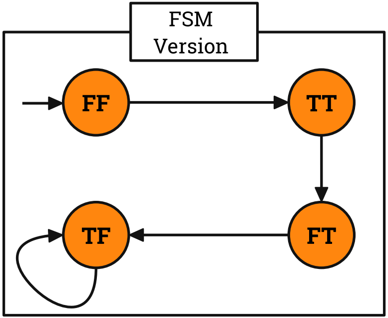
Example SMV Machine

```
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;
```

- 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:

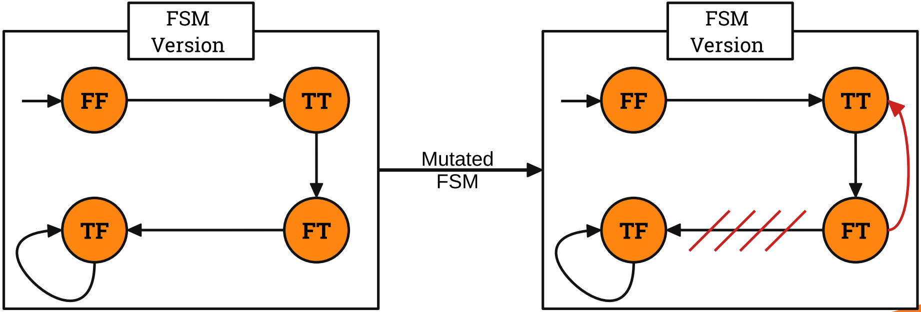
- changes a constant to each other constant
- in the **next(y)** case: **!x & y : false** is mutated to **!x & y : true**
- To kill this mutant, we need a **sequence of states** (a path) that the **original machine allows** but the mutated machine **does not**
- This is what **model checkers** do
  - 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 results from **!x & y : true**

Counter-Example for FSM

```
next(y) := case
            x & !y : false;
            x & y  : y;
            !x & y :  false;
Δ 1         !x & y : true;
            true   : true;
         esac;
```

Written in SMV as →

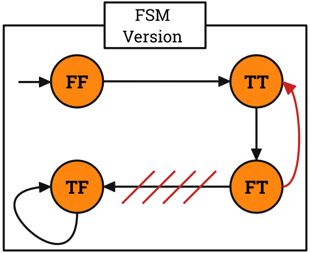
```
next(y) := case
            x & !y : false;
            x & y  : y;
            !x & y :  false;
            true   : true;
         Esac;
SPEC AG (!x & y) → AX (y = true)
```



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**

Are there any questions?