Lecture CDT414

Model Based Testing

Eduard Paul Enoiu

December 2014

@eduardpaulenoiu; **www.testinghabits.org**

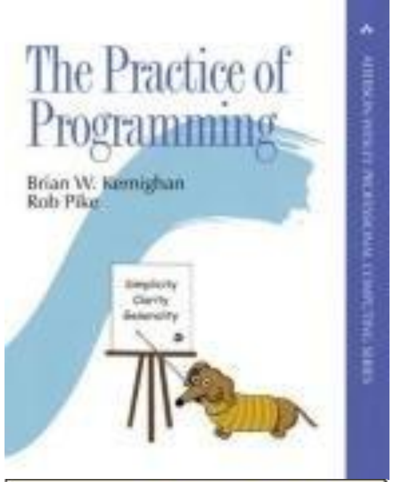
Before we start

- What do I know about *Model Based Testing*, anyway?
- I've written programs and tested them (sometimes by using models)
  - So have most of you, I would bet
- Split my time at **Mälardalen University** and **Bombardier Transportation** between model-based testing and automated testing research.
- I am the co-author of the **CompleteTest** testing tool together with Adnan.

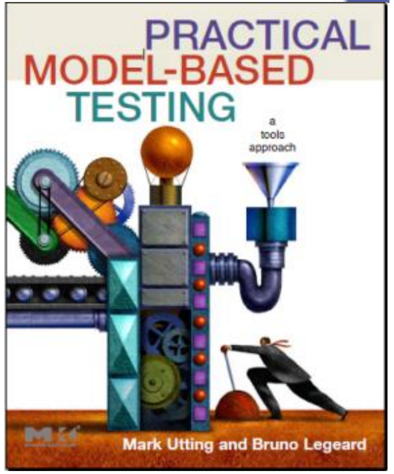
Today

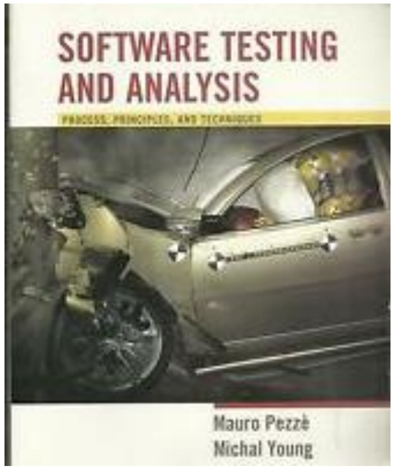
- Overview of Model-Based Testing
- Modelling Software
- Strategies for Generating Tests using Models
- Executing Tests Generated from Models
- Summary and Conclusions

Read All About It



- No textbook for this class
- Books I like that have something important to say about model based testing:
  - *The Practice of Programming*, Kernighan and Pike
  - *Practical Model-Based Testing: A Tools Approach*, Mark Utting, Bruno Legeard
  - *Software Testing and Analysis*, Pezze and Young
    - Try Chapter 14
    - I like it myself
    - Recommended by colleagues who've taught classes on testing (and are first-rate testing researchers)
    - Book is thorough and cleverly organized, provokes some real thought about how to test programs and models

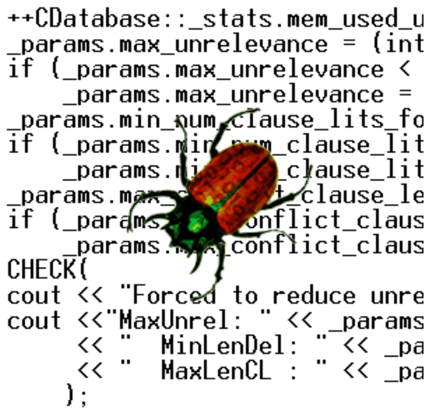



SOFTWARE TESTING AND ANALYSIS
PROCESS, PRINCIPLES, AND TECHNIQUES
Mauro Pezzè
Michal Young

Overview of Model-Based Testing

Basic Definitions: Model Based Testing

- What is Model Based Testing?
  - Running a program based on a spec (a.k.a. model)
  - In order to find faults
    - a.k.a. defects
    - a.k.a. errors
    - a.k.a. flaws
    - **a.k.a. BUGS**

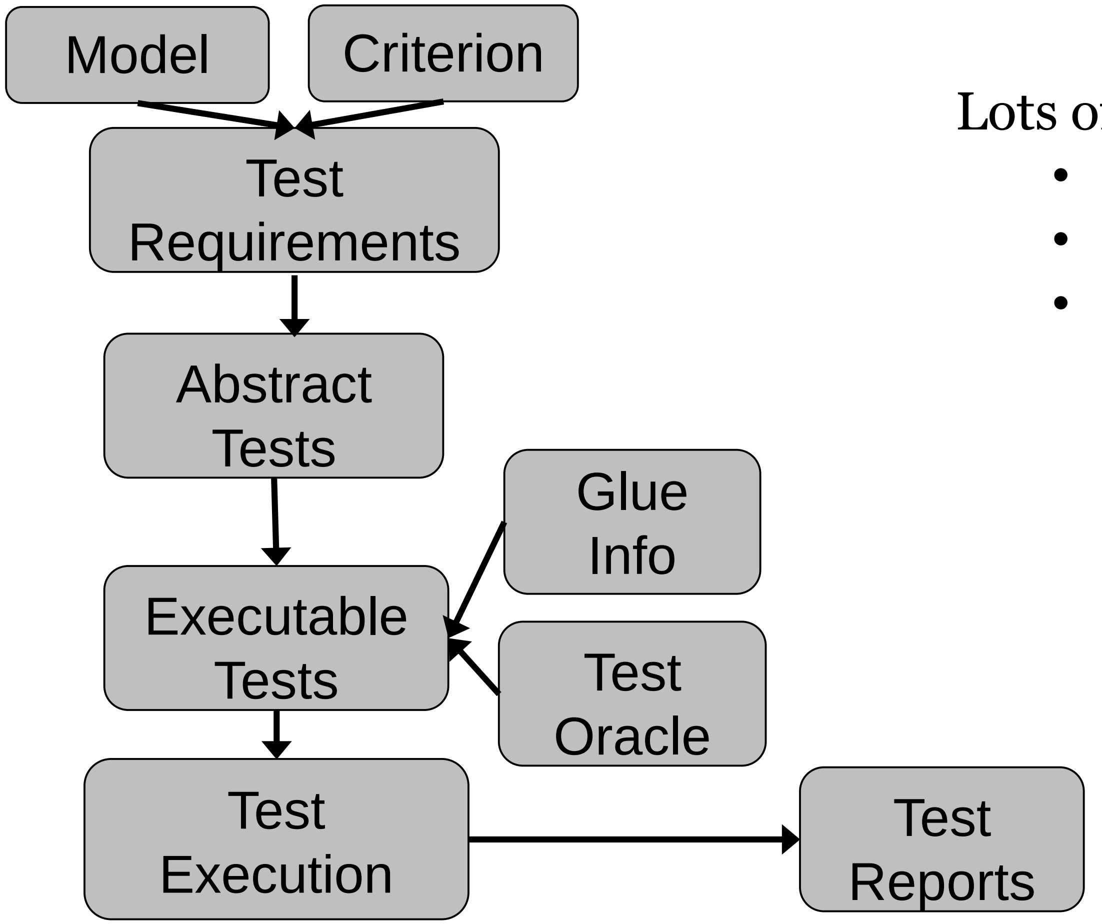


```
++CDatabase::_stats.mem_used_u
_params.max_unrelevance = (int
if (_params.max_unrelevance <
    _params.max_unrelevance =
_params.min_num_clause_lits_fo
if (_params.min_num_clause_lit
    _params.mi     _clause_lit
_params.ma       t_clause_le
if (_params     onflict_claus
    _params.max_conflict_claus
CHECK(
cout << "Forced to reduce unre
cout <<"MaxUnrel: " << _params
     << "  MinLenDel: " << _pa
     << "  MaxLenCL : " << _pa
    );
```

Benefits of Model-Based Testing

- Better tests
- Lower cost
- Early detection of requirements errors
- Traceability
- Automation
- Less overlap among tests

Model Based Testing Process



Lots of research and tools for:

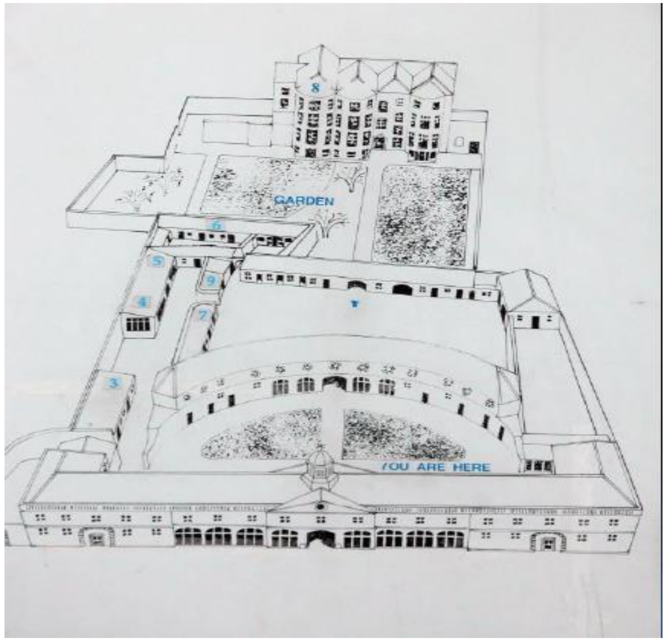
- Test criteria
- Creating models
- Executing tests

Modeling Software

Models

- A reduced/abstract representation of a system that highlights the properties of interest from a given viewpoint
  - Unnecessary details are removed
  - Highlight subject of interest
- To deal with complexity of systems development
  - Abstract to focus on a particular point of interest
  - It facilitates studying and understanding the behavior of complex systems
- To improve communication
  - A good model is often better than a textual description
- To reduce development flaws
  - Detecting errors early in e.g. requirements

Model of a Castle



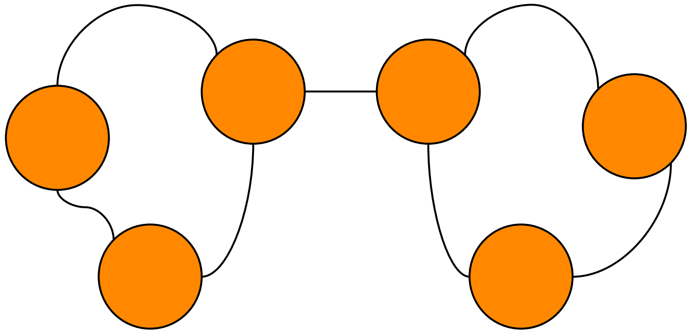
Model of an Actor

Useful models

- Abstract
    - Emphasize important aspects while removing irrelevant ones
- Understandable
    - Expressed in a form that is readily understood by observers
- Accurate
    - Faithfully represents the modeled system
- Predictive
    - Can be used to answer questions about the modeled system
- Inexpensive
    - Cheaper to construct and study than the modeled system

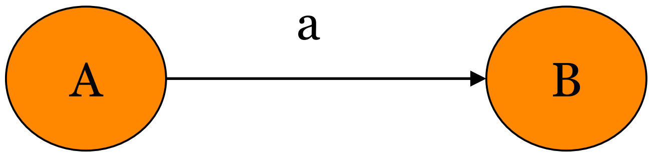
Models: Definition

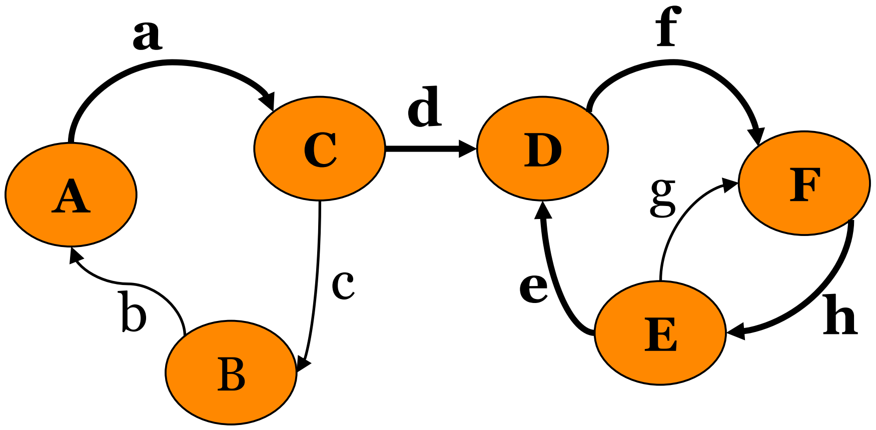
- A graph G is a model composed of
  - N: A set of nodes or vertices
  - A: A set of arcs or edges connecting these nodes
- Graphs represent relationships between objects for which the nodes stand



Models: Definition

- Label: an identifier that names an element of a graph. The arc from A to B is labeled a.
- Path: a sequence of arcs such as any two adjacent arcs in the sequence share a common node.

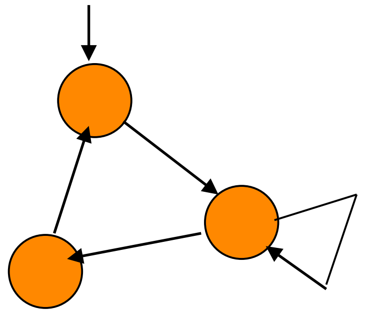


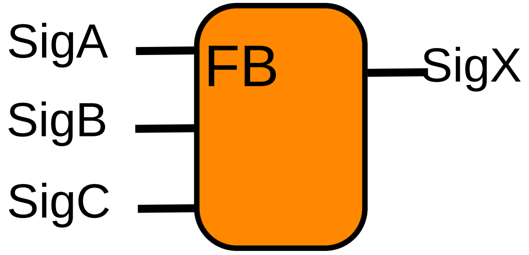


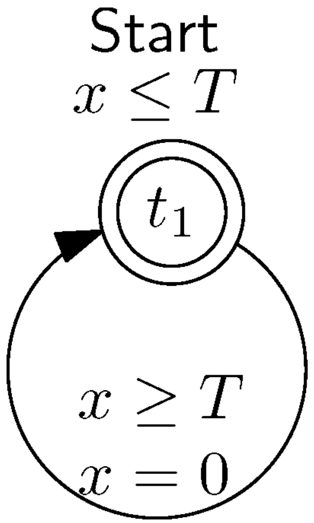
<a, d, f, h, e> is a path

Examples of models

- Structural models
  - UML class diagrams
- Functional models
  - State/transition diagrams
  - Finte state automata/machines
- Control models
  - Matlab/Simulink
  - Function Block Diagrams
- Timing models
  - Timed automata
  - Simulink





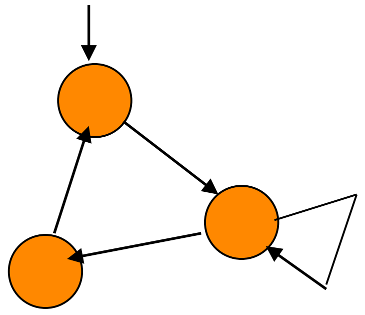


Models: Uses

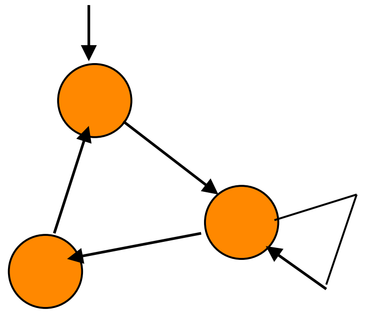
- Maps
- Traffic system dependencies
- Computer networks
- Control flow graphs
- Data flow graphs
- Routine call graphs

State Machines

- A State is an abstraction of values of those variables of a software system that govern the way it works. In particular, a state defines:
  - The inputs that can be applied by the user when the software is in the state
  - The manner in which an input affects the system
    - Output
    - Other response (change of states, internal variable value change, etc...)



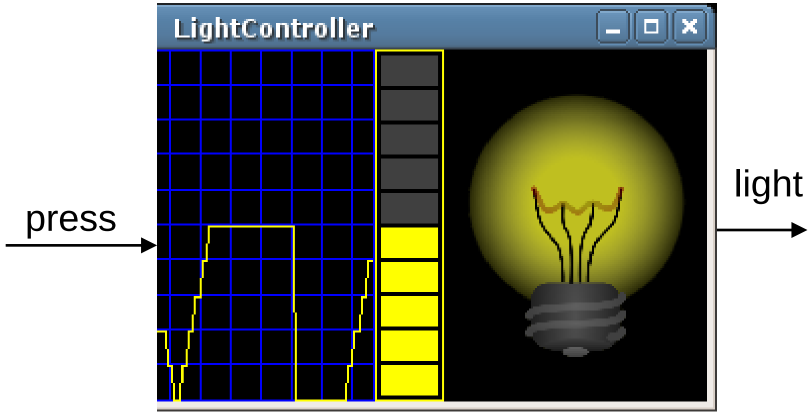
Finite State Machines



- A finite state machine is an abstract mathematical structure that is composed of:
  - A finite set of states
    - One of these states is the initial state. A state machine always starts in an initial state
    - A number of these states are designated as final states
    - A machine terminates successfully if it is in a final state
  - A finite set of inputs
  - A complete definition of how the machine makes a transition from one state to another

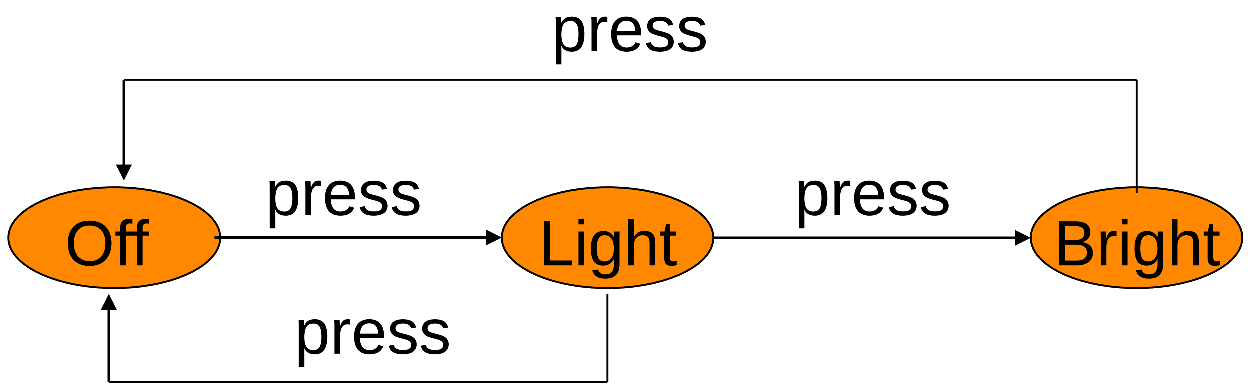
Modeling Example

- Light switch:
  - Requirements: If a user quickly presses the light control twice, then the light should get brighter; on the other hand, if the user slowly presses the light control twice, the light should turn off.



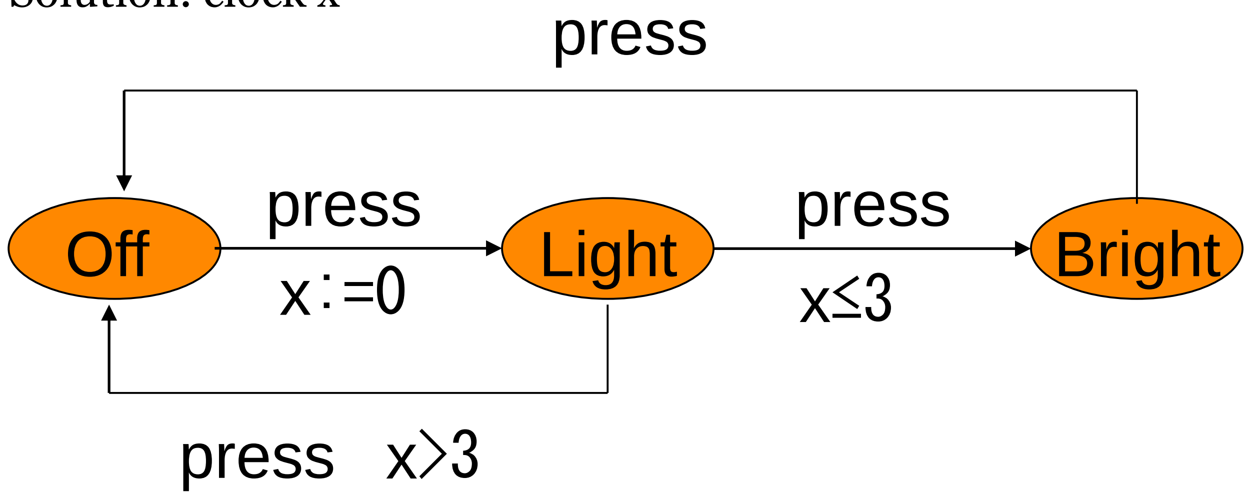
Modeling Example

- Light switch:
  - Requirements: If a user quickly presses the light control twice, then the light should get brighter; on the other hand, if the user slowly presses the light control twice, the light should turn off.



Modeling Example

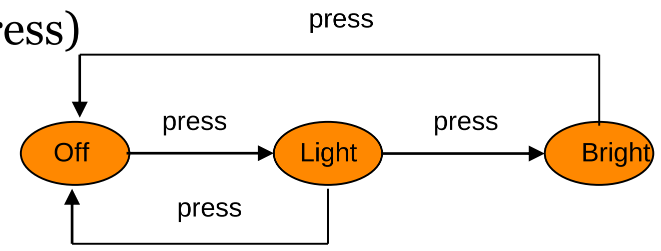
- Requirements: If a user quickly presses the light control twice, then the light should get brighter; on the other hand, if the user slowly presses the light control twice, the light should turn off.
- Solution: clock x



Strategies for Generating Tests using Models

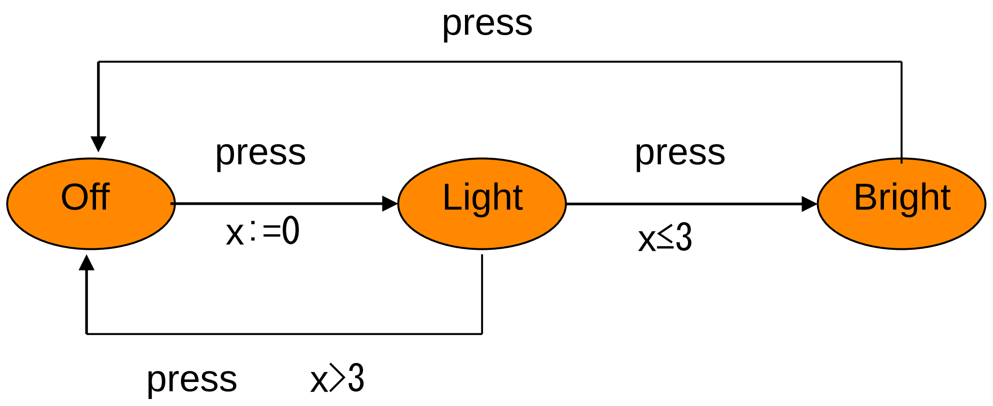
The Big Picture

- Testing (almost always) is an attempt to
  - Cover some measure of a structure
  - Nodes of a graph (e.g., light switch example)
  - Inputs that give different outputs (e.g., press)
  - All possible inputs
  - Logical expression evaluations
  - Predicates over program variables
  - Pairs of where a variable is defined and where it is used (data flow)
- Usually, we can’t even guarantee that coverage directly correlates to more bugs found.

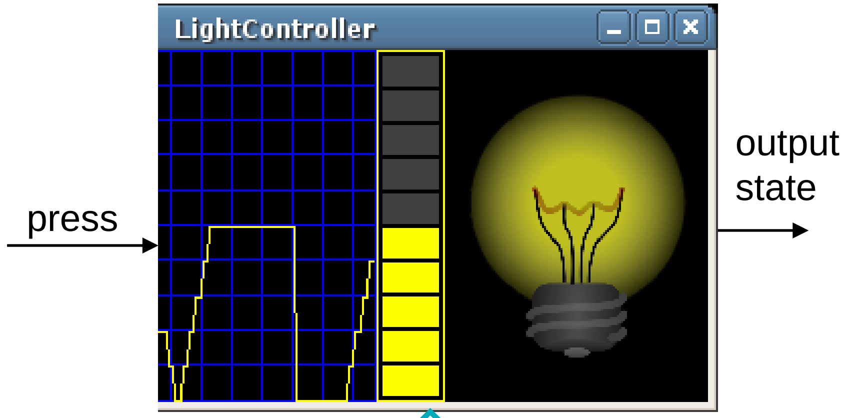


From Models to Test Cases

Spec a.k.a. Light Controller Model



Software Implementation

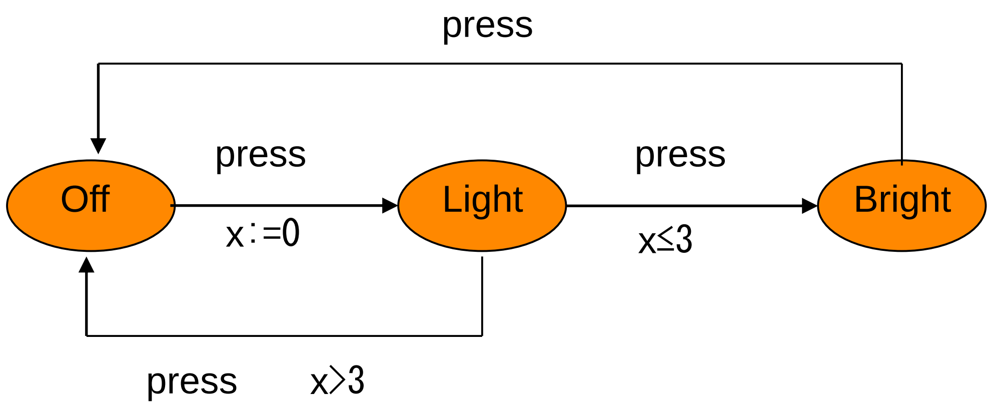


RQ1: Check that the light can become bright.

TC1: start
input: press output: (Light,0)
input: press output: (Bright, 3)
stop

”Covering” Finite State Machines

- **State Coverage**
  - Every state in the model should be visited by at least one test
- **Transition Coverage**
  - Every transition between states should be traversed by at least one test case



Strategies for searching trough models

- Model-Based Testing can focus on exploring lots of
  - Executions
    - Random testing
  - Paths
    - Concolic testing
  - States?
    - Model checking

Model Checking

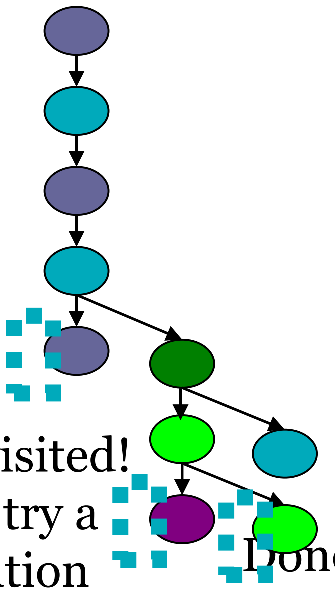
- A model checker is a tool for exploring a *state space*
- Basic idea: generate every reachable state of a *transition system*
  - Think of states as nodes in a graph
  - Directed edges mean "from this state, this is a possible next state of the program"
  - Multiple outgoing edges where there is input/thread scheduling/other nondeterminism

Model Checking

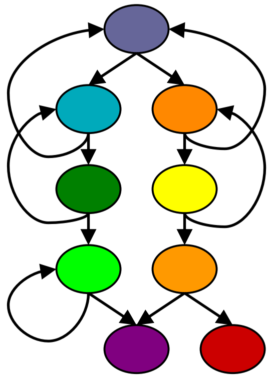
- We will be looking at one particular kind of model checking
  - Using UPPAAL
  - To explore the state spaces of programs
- Other model checkers used for this approach to testing
  - SPIN (C programs)
  - Java PathFinder 2 (NASA Ames)
  - Bogor (U Kansas/Nebraska)

Model Checking

- Model-checking by *executing the program*
  - Backtracking search for all *states*



Will explore, as a side-effect, many executions and many paths, but the goal is to explore states



CFG

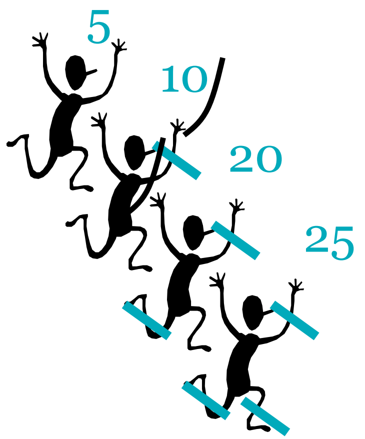
State already visited! Backtrack and try a different operation

Done with test! Backtrack and try a different operation

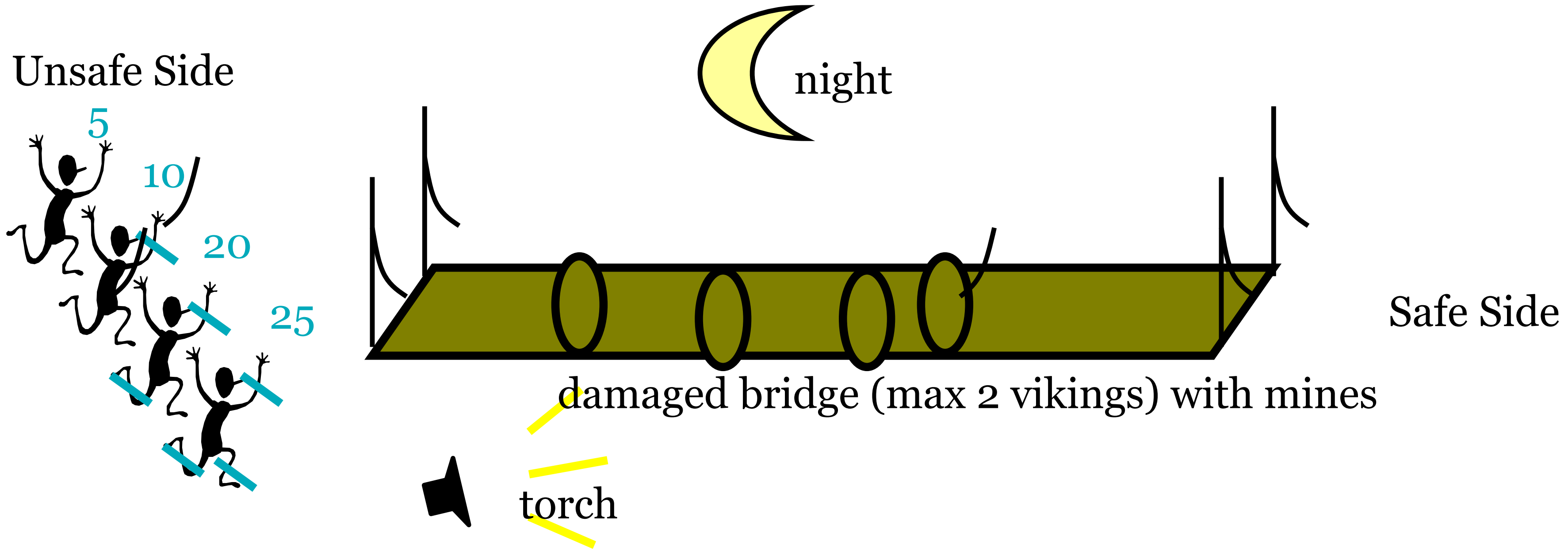
State already visited! Backtrack and try a different operation

Exercise: Test Generation using Models

- Example
  - Testing mission scenarios in a video game.
  - Vikings crossing a bridge

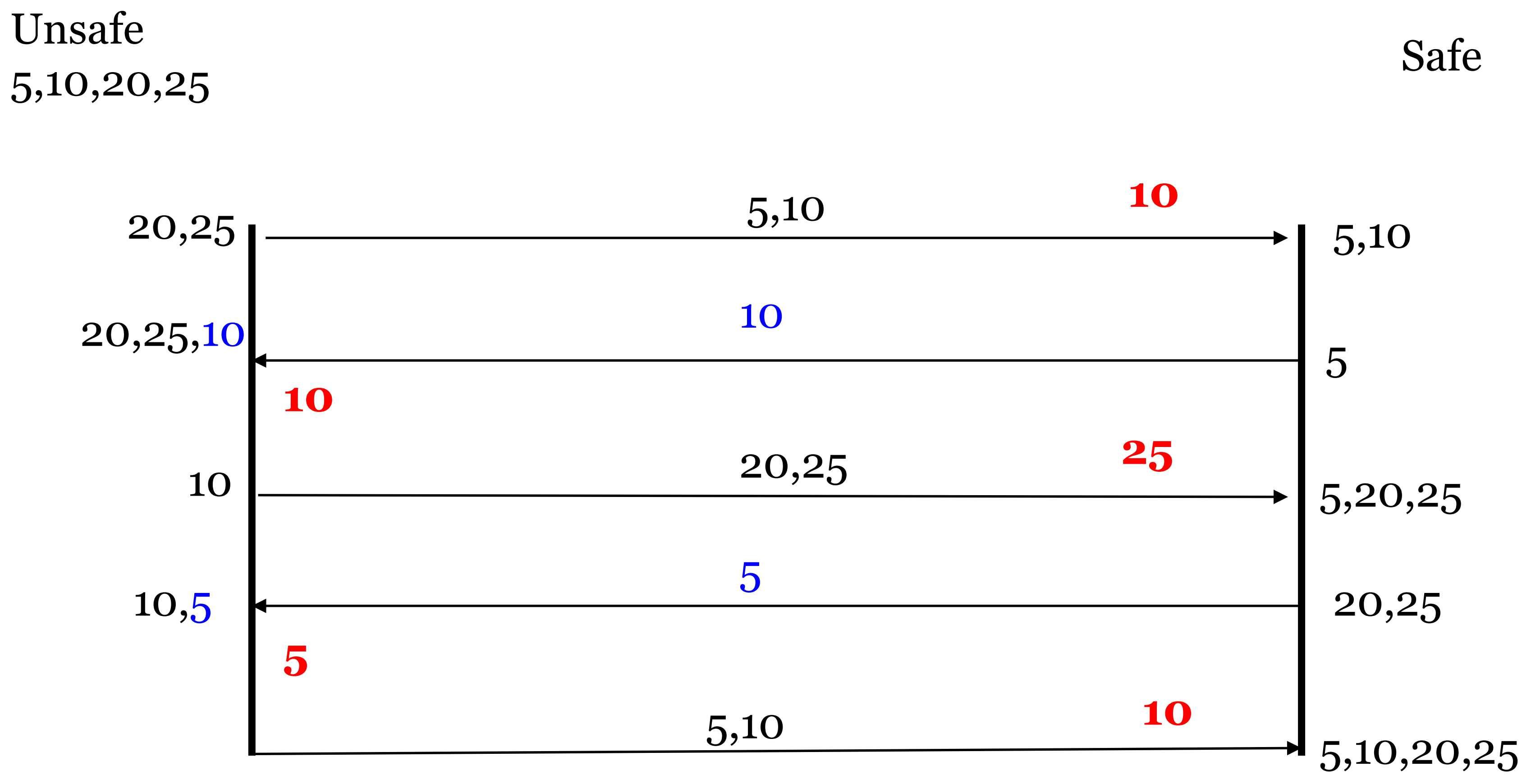


Exercise during break: Test Generation

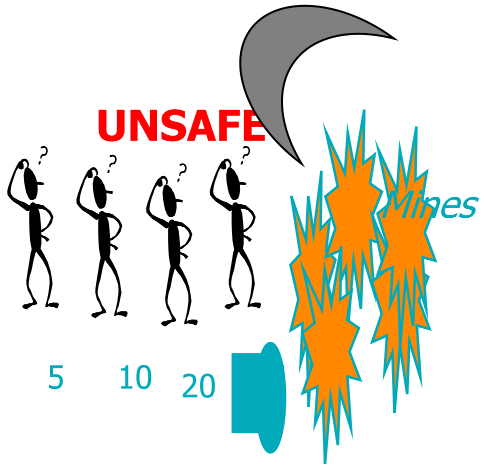


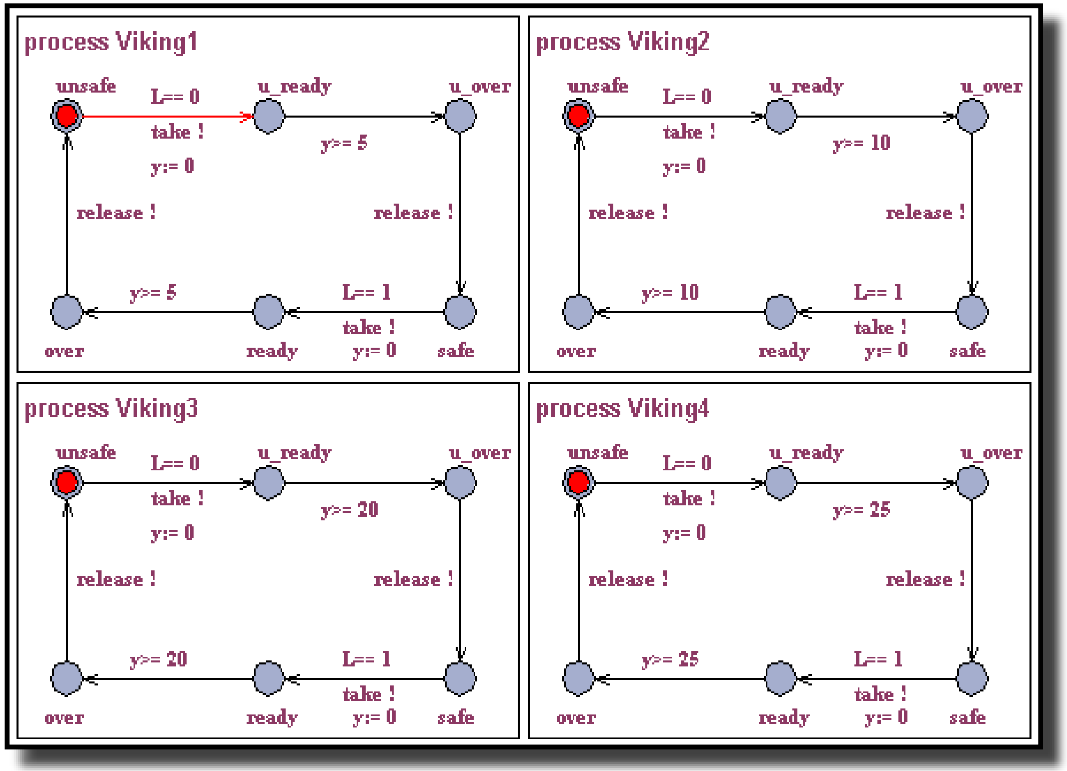
If possible find a test that shows
that all four vikings
can reach safe side in 60 min.

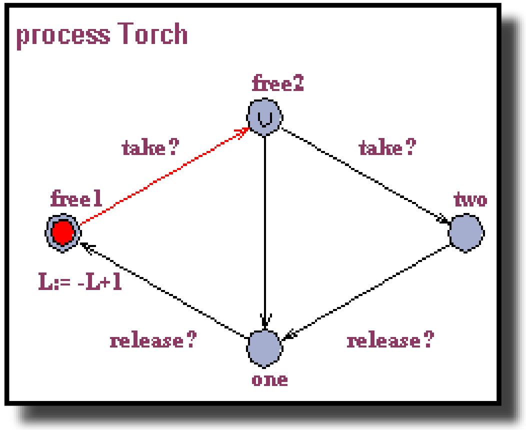
My solution to the exercise



Exercise: Test Generation using Models







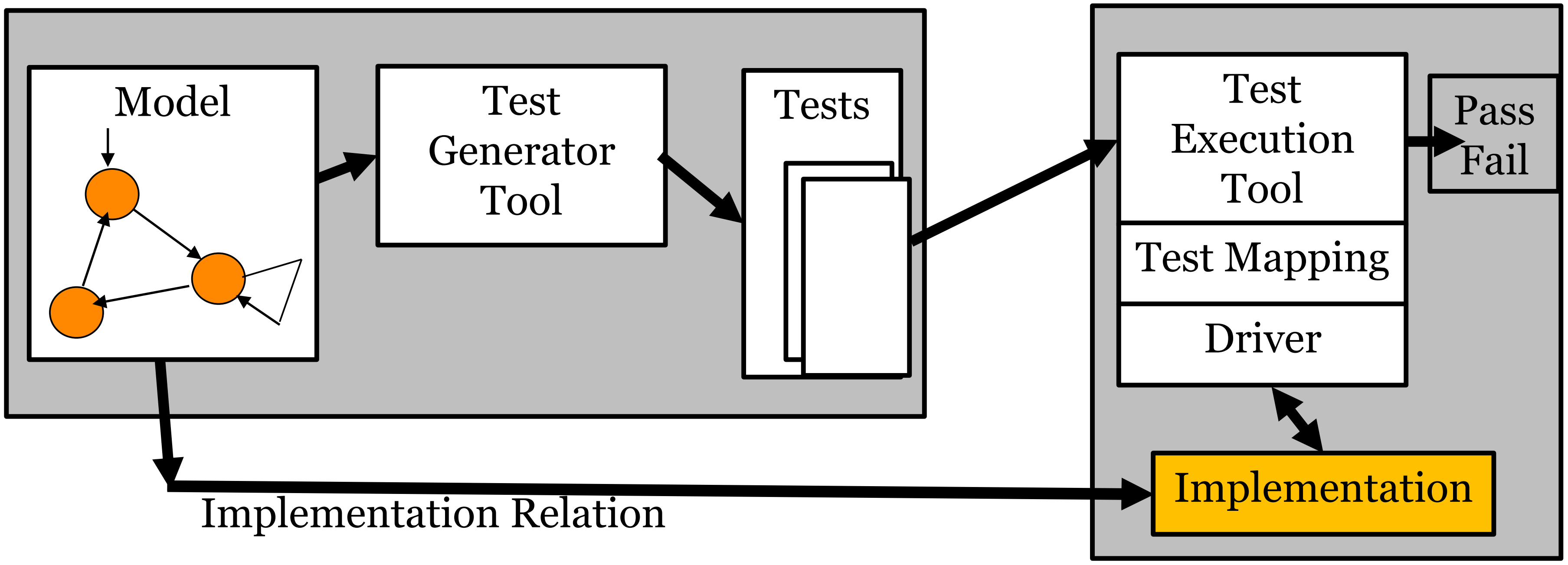
- Can be modeled and solved with model-checking

Executing Tests Generated from Models

MÄLARDALEN UNIVERSITY
SWEDEN

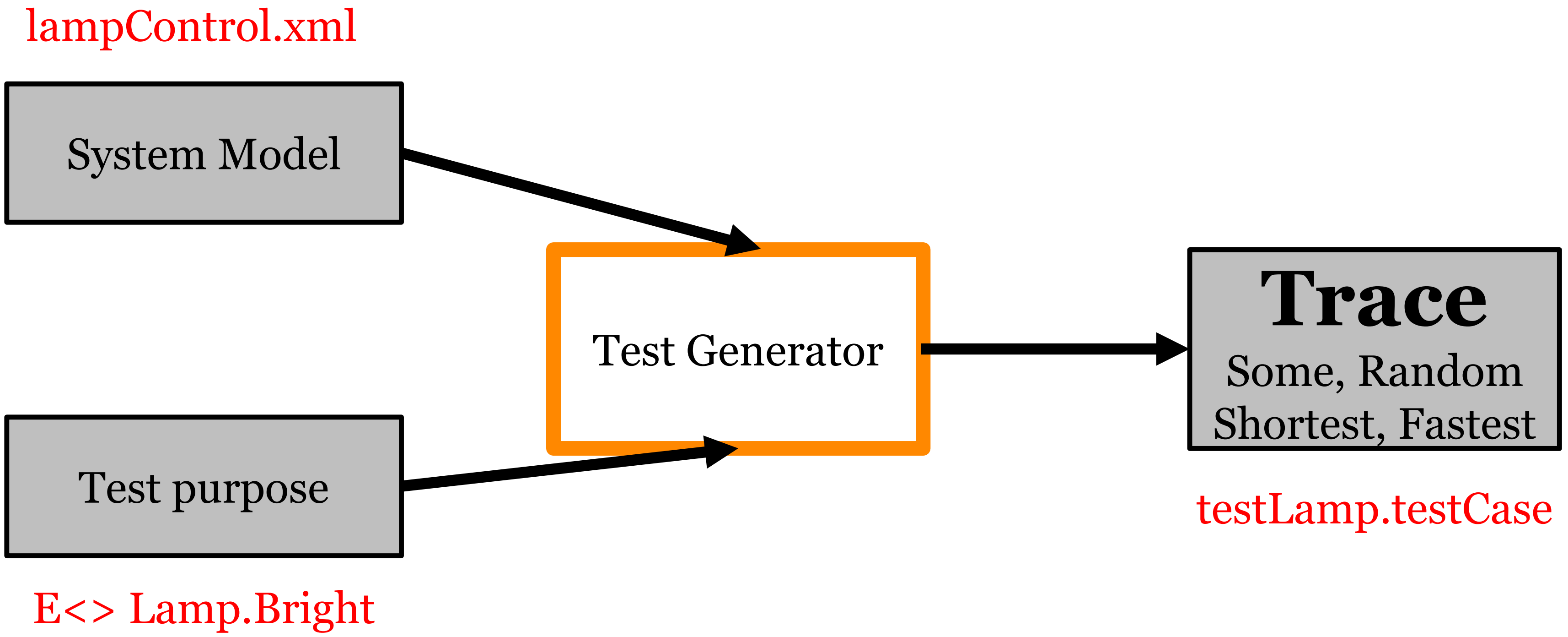
Model-Based Testing: Conformance

- Does the behavior of he implementation comply to that of the specification?



Test Generation and Execution

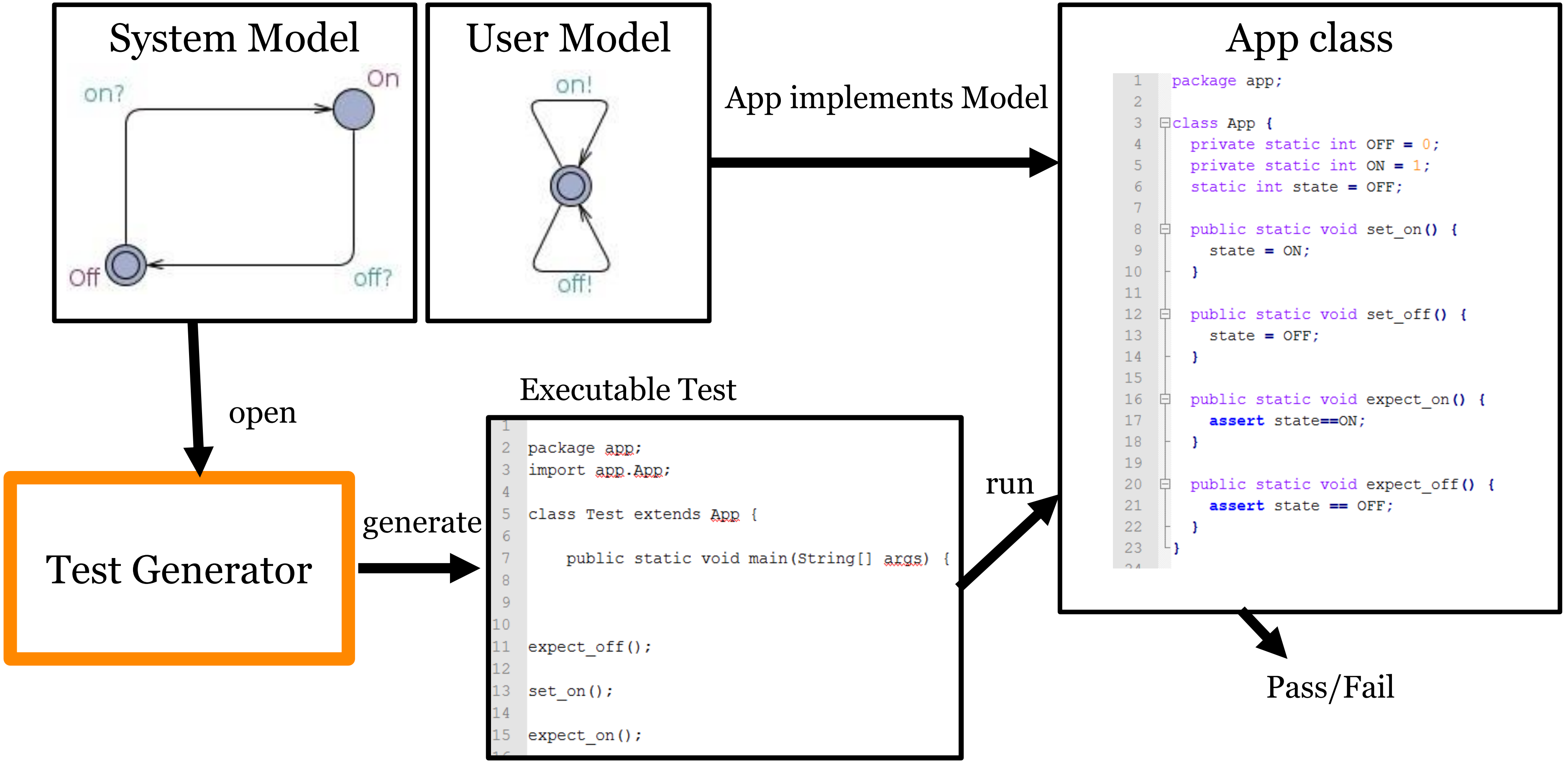
- Use traces as tests by using a model checker



Model-Based Testing Exercise

- Yggdrasil is a test-case generation feature of UPPAAL
- I will explain model-based testing by developing and testing a simple Lamp system with on/off capabilities.
- I should write a:
  - A model, decorated with test-code.
  - A correct implementation of the system
  - The system implemented in Java
  - Execution scripts

Model-Based Testing Exercise



```
package app;

class App {
  private static int OFF = 0;
  private static int ON = 1;
  static int state = OFF;

  public static void set_on() {
    state = ON;
  }

  public static void set_off() {
    state = OFF;
  }

  public static void expect_on() {
    assert state==ON;
  }

  public static void expect_off() {
    assert state == OFF;
  }
}
```

```

package app;
import app.App;

class Test extends App {

    public static void main(String[] args) {



expect_off();

set_on();

expect_on();
```

Model-Based Testing Exercise with Mutation

- **Mutation testing** is used to design new tests and evaluate the quality of existing tests. Mutation testing involves modifying a program in small ways. *(1978, DeMillo and Lipton)*

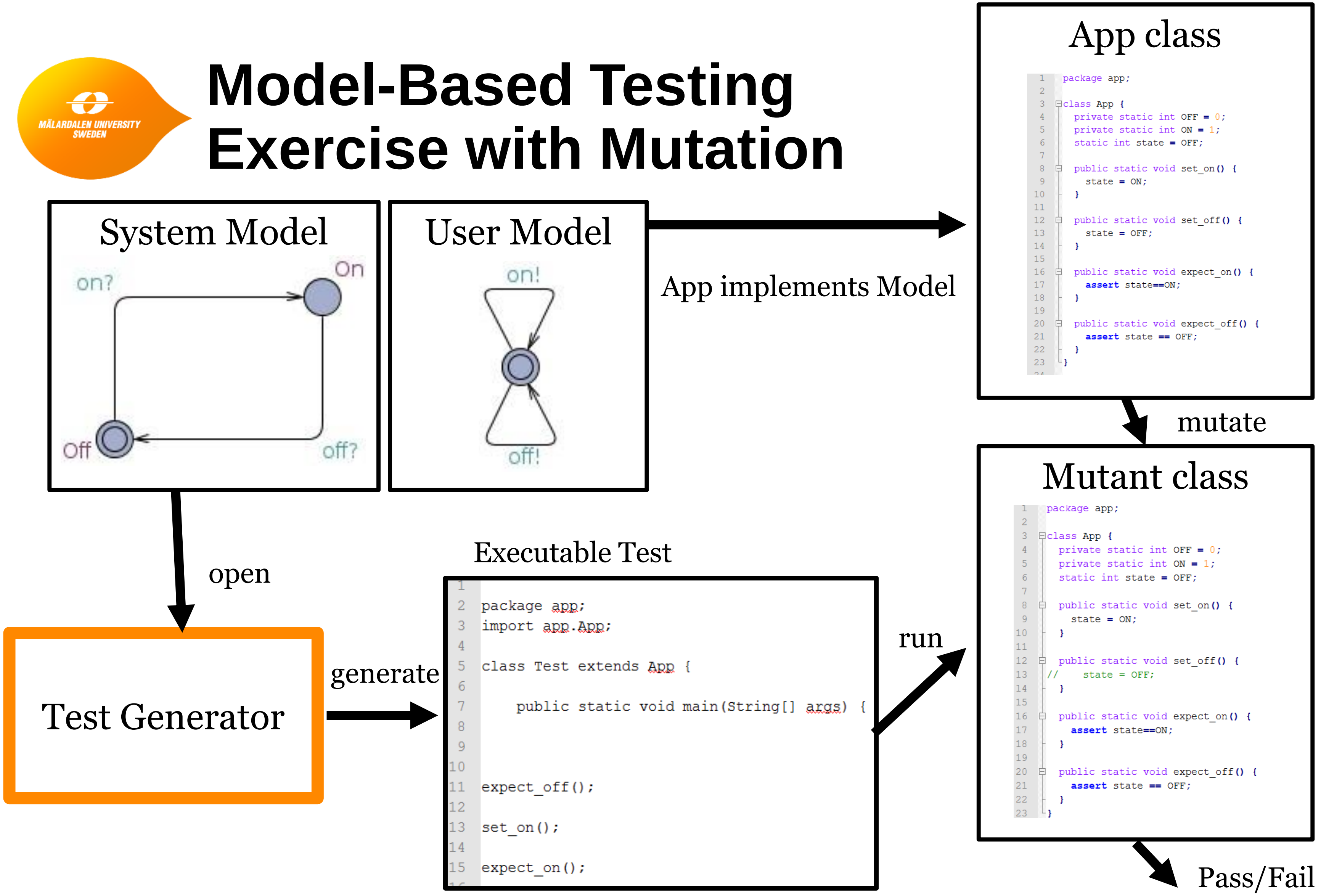
- Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant.

- The purpose is to help the tester develop effective tests or locate weaknesses in the tests used for the program or in sections of the code that are seldom or never accessed during execution.

Model-Based Testing Exercise with Mutation

- Yggdrasil is a test-case generation feature of UPPAAL
- I will explain model-based testing by developing and testing a simple Lamp system with on/off capabilities.
- I should write a:
  - A model, decorated with test-code.
  - A mutant with an implementation error
  - The mutant is implemented in Java
  - Execution scripts

Model-Based Testing Exercise with Mutation



App class

```
package app;

class App {
  private static int OFF = 0;
  private static int ON = 1;
  static int state = OFF;

  public static void set_on() {
    state = ON;
  }

  public static void set_off() {
    state = OFF;
  }

  public static void expect_on() {
    assert state==ON;
  }

  public static void expect_off() {
    assert state == OFF;
  }
}
```

Mutant class

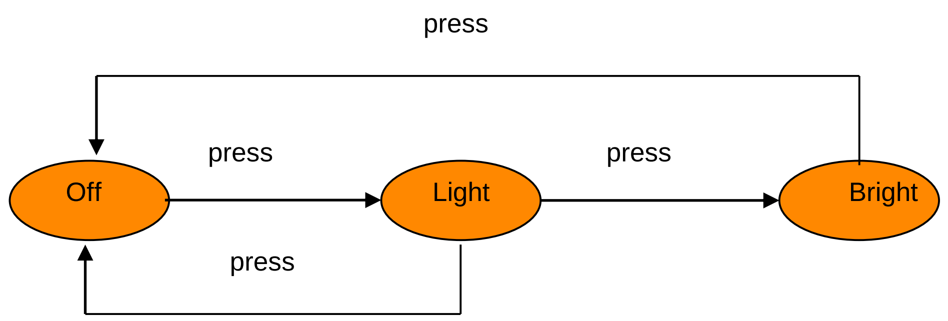
```
package app;

class App {
  private static int OFF = 0;
  private static int ON = 1;
  static int state = OFF;

  public static void set_on() {
    state = ON;
  }

  public static void set_off() {
//    state = OFF;
  }

  public static void expect_on() {
    assert state==ON;
  }

  public static void expect_off() {
    assert state == OFF;
  }
}
```

Executable Test

```

package app;
import app.App;

class Test extends App {

    public static void main(String[] args) {



expect_off();

set_on();

expect_on();
```

Home Assignment (optional)

- Download Windows, Linux or Mac, version 4.1.19 of UPPAAL from **uppaal.org**
- Extend the lamp switch such that it works as follows



- Implement it in Java using the template and generate executable tests using Yggdrasil and UPPAAL:
  - using depth search and a depth of 20.
  - Using a test property: Lamp becomes Bright

Summary and Conclusions

Advantages

- Automate generation of
  - Large suites of tests and lengthy tests
  - Complicated input sequences
- Likely to expose failures
  - Caused by weird combinations of inputs

Advantages

- Provides an additional basis for
  - Coverage-based evaluation of test progress
  - Measurement quality of the product
- A model is a precise communication tool
- Good vehicle of presentation to non-technical staff
- While building a model
  - Develop a better understanding of SUT
  - Find many bugs while exploring SUT
- Compare versions of the product

Choosing Models to Fit Needs

- Highly dependent on application type
  - HTML processing component in a browser
  - Calculator mathematical expressions
  - State rich systems: GUI & Phone systems
  - Parallel systems
- Different models for different system aspects
- Combinations of models are more useful
- Try to reuse models from design & requirements analysis

Tool Support

- Try the following tools:
  - **UPPAAL** (used for embedded systems)
    - Free to use for academics like you
    - Has support for model-based testing of Java code
  - **Spec Explorer** (model programs in C#) Free to use
    - Is a Visual Studio Power Tool.
    - 100 testers use it on a daily basis at Microsoft (Campbell 2005, FM)
  - **CompleteTest** (you used it in the lab for automatically generating tests for FBDs)
    - Is used by Bombardier engineers for unit testing (**www.completeTest.org**)
  - Coq Model Checker (used for verifying OS kernels)

Thank you

Eduard Paul Enoiu

@eduardpaulenoiu; **www.testinghabits.org**