Chap 3. Test Models and Strategies

3.2 State-based Testing

1. Introduction
2. State Transition Diagram
3. Transition Test Strategy

1. Introduction

-State-based testing relies on the construction of a finite-state machine (FSM) or state-transition diagram that describes the dynamic behavior of the system under testing.

-However, conventional FSM models are not ready for test case generation, because in general they are ambiguously defined.

- Effective test data generation requires making the FSM model test-ready by defining precisely the underlying meaning.

-The FSM model can be derived from the program specification, or extracted from the code through reverse-engineering.

-State-based models can be used both for unit and integration testing, and for functional as well as structural testing.

-Many state-based test strategies have been proposed in the research literature. In this chapter, we’ll illustrate state-based testing by studying the **transition-based testing strategy**.

-The transition-based test strategy **rigorously** defines the **semantics** of the basic features of a state diagram (e.g., state, event etc.) as invariants or predicates, and analyze these predicates using **domain matrix** in order to generate suitable test cases.

-The advantage of the transition test strategy compared with classical domain analysis is that it emcompasses implicitly a test oracle; so it removes the need to define explicitly expected outputs, which can be extremely difficult in many cases.

2. State-Transition Diagram

-Use cases and scenarios provide a way to describe system behavior, that is the interactions between objects in the system.

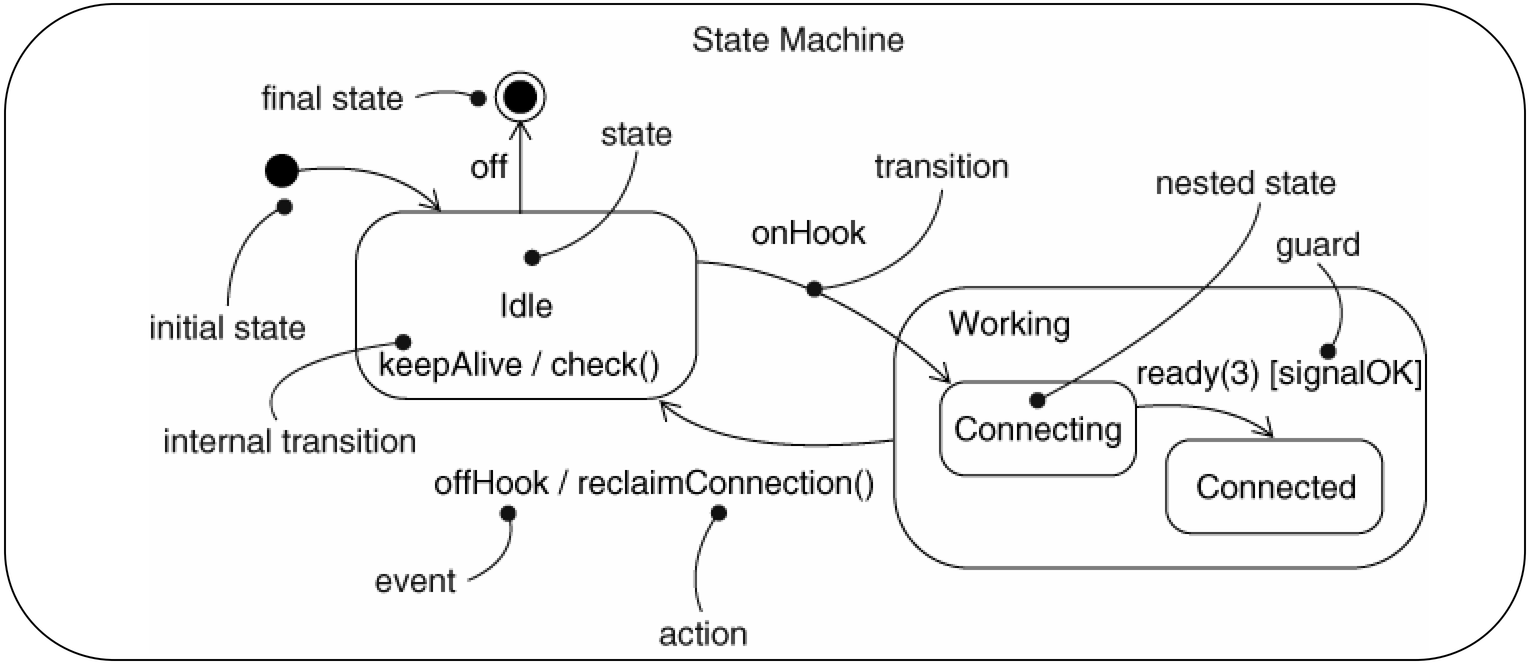
-UML state diagram is used to model the dynamic behavior of any modeling element, such as a class, a use case or an entire system.

- It allows the modeling of the behavior inside a single object.
- It shows the sequence of states of an object during its lifetime.
- It ***shows the events or messages*** that cause a ***transition*** from ***one state to another***, and the actions that result from a state change.
- It is ***created only for classes with significant dynamic behavior***, like control classes.
- So it is appropriate for developing test cases at the class level.

-The main building blocks of a state transition diagram are ***states*** and ***transitions***.

☞ **State**:

- a situation during the life of an object when it ***satisfies some condition, performs some action***, or ***waits for an event***
- found by examining the attributes and links defined for the object
- represented as a rectangle with rounded corners



☞ **Transition**:

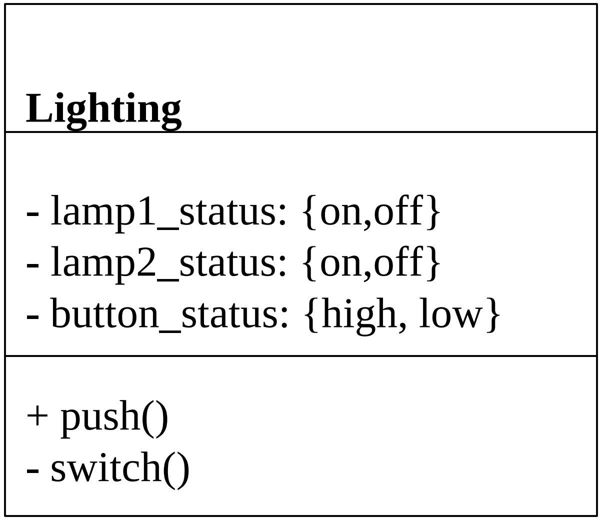
- represent a change from an originating state to a successor state (that may be the same as the originating state).
- may have an action and/or a guard condition associated with it, and may also trigger an event.

Example: Lighting System

Specify a lighting system with two lamps and one button functioning as follows:

1. When the light is off, pushing the button causes the first lamp to go on.
2. Pushing the button again causes the second lamp to go on and the first to go off.
3. Pushing the button yet again causes both lamps to go on, and pushing it once more switches both lamps off.

Class Diagram

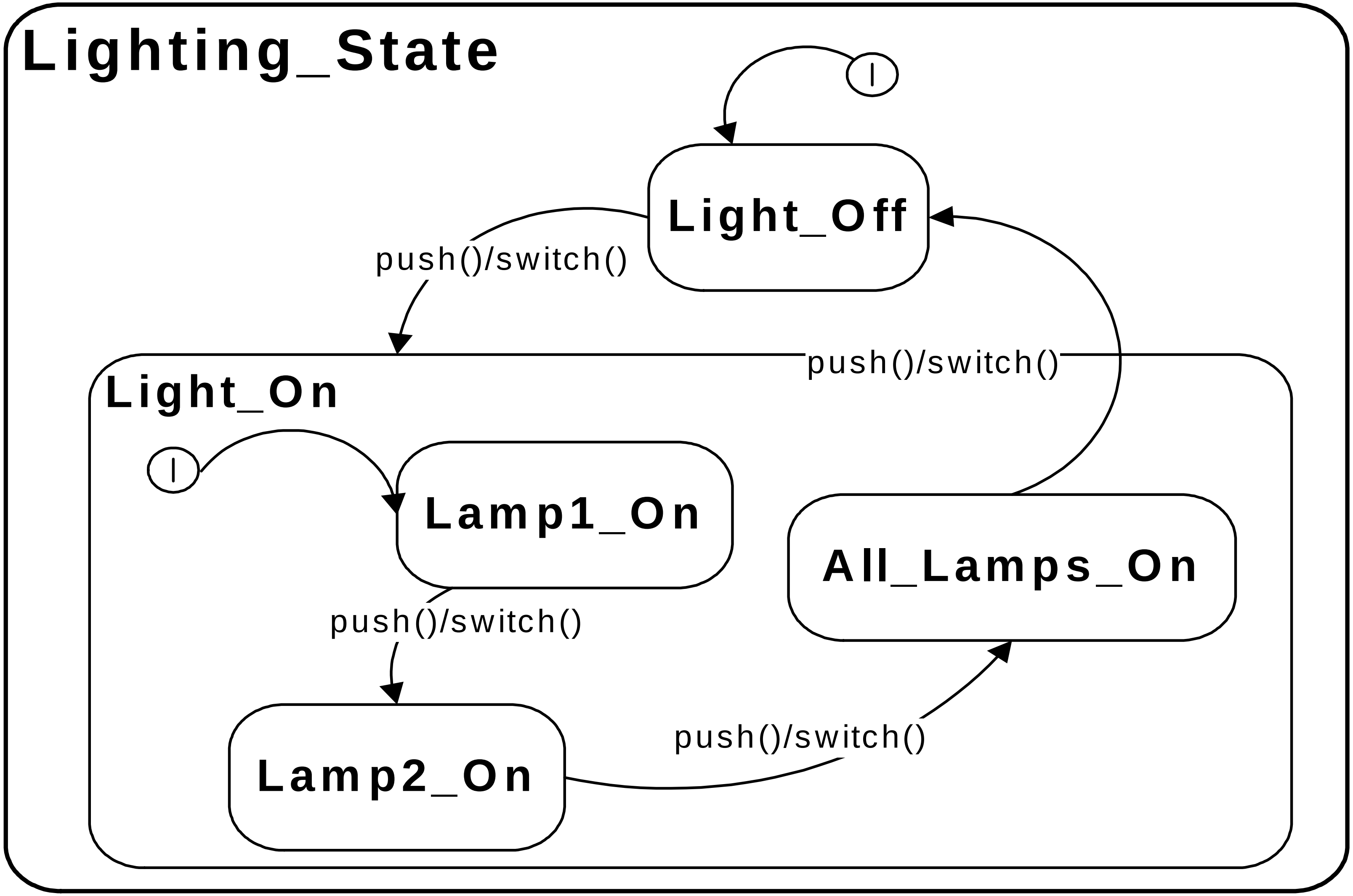


Lighting

- lamp1_status: {on,off}
- lamp2_status: {on,off}
- button_status: {high, low}

+ push()
- switch()

State Diagram



Possible Implementation

```
public class Lighting {
                private static boolean on = true;
                private static  boolean off = false;
                private static boolean high = true;
                private static boolean low = false;
                private boolean lamp1Status, lamp2Status, buttonStatus;
                public Lighting() {lamp1Status=false; lamp2Status=false; buttonStatus=true;}

                public Lighting(boolean l1,boolean l2, boolean b) {
                                lamp1Status=l1; lamp2Status=l2;
                                buttonStatus=b;
                }
                public void push() {
                                if ((lamp1Status==false) & (lamp2Status==false)) lamp1Status = true;
                                elseif ((lamp1Status == true) & (lamp2Status==false))
                                                {
                                                                lamp1Status = false;
                                                                lamp2Status = true;
                                                }
                                elseif ((lamp1Status == false) & (lamp2Status==true)) lamp1Status=true;
                                else  {
                                                                lamp1Status = false;
                                                                lamp2Status = false;
                                                }
                                switch();
                }
                private void  switch () {buttonStatus = ! buttonStatus;}
}
```

3. Transition Test Strategy

-A (UML) state diagram describes the sequence of states through which an object evolves during its lifetime, as well as the sequence of messages it sends and/or receives.

- The messages exchanged during that lifetime correspond to method calls, i.e., events, and are associated to transitions between states.

- Hence, the actual execution of a method is closely related to the firing of a corresponding transition in the statechart specifying the class behavior.
- Since a method can be executed under various circumstances, several transitions can be associated with a method, each corresponding to a particular circumstance.

-The transition test strategy identifies the collection of transitions associated with a method and uses these transitions as basis for test case generation.

- It requires the definition of precise semantics for the state model.

Semantics Definition

-Possible semantics for state model may consist of defining predicate expressions for *basic states*, *actions*, and *guard conditions*.

- ***Guard condition***: the predicate corresponds to the expression of the condition.
- ***Basic state***: the predicate represents an abstraction of corresponding situation.
- ***Action:*** the predicate corresponds to an abstraction of the output or result of the action

-State predicates may be function of only state variables (i.e., attributes or instance variables), while conditions and actions predicates may be function of both state variables and method parameters.

-Example: Semantics Definition for Lighting System

State Predicates

Pred(Light_Off) = (lamp1_status = off) $\wedge$ (lamp2_status = off)

Pred(Lamp1_On) = (lamp1_status = on) $\wedge$ (lamp2_status = off)

Pred(Lamp2_On) = (lamp1_status = off) $\wedge$ (lamp2_status = on)

Pred(All_Lamps_On) = (lamp1_status = on) $\wedge$ (lamp2_status = on)

Action Predicates

Pred(switch) = (button_status’ = $\neg$ button_status)

Condition Predicates

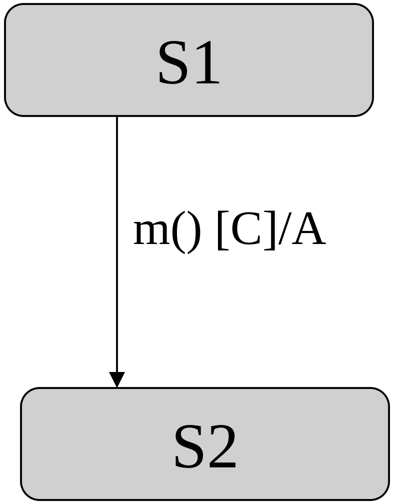
There is no guard condition associated with the transitions (in this example), so no condition predicate.

Definition of Method Pre/Post-Conditions

-The activation of a transition involves two predicates.

- The first predicate defining the enabling condition for the transition is associated with the source state and the guard condition.
- The second predicate defines the outcome of firing the transition, and as such it is related to the target state and the resulting action.

-This pair of predicates can be used to define partial pre-post condition pair for the transition.



pre_m= pred(S1) Ù ***pred(C)***

post_m=pred(S2) Ù ***pred(A)***

- The global pre/post-condition for a method corresponds to the combination of the pre/post condition pairs associated with all the transitions involving that method.

Example: Pre/Postconditions Pairs for Lighting System

- Four transitions are associated with method push(), which correspond to 4 (partial) pre/postconditions pairs.

t1: Light_Off→Lamp1_On; t2: Lamp1_On→Lamp2_On;
t3: Lamp2_On→All_Lamps_On; t4: All_Lamps_On→Light_Off

- Partial Pre/postcondition pair for t1:

pre_push1 = pred(Light_Off) $\wedge$ pred (C1) = (lamp1_status = off) $\wedge$ (lamp2_status = off)
post_push1= pred(Lamp1_On) $\wedge$ pred (switch)
= (lamp1_status’ = on) $\wedge$ (lamp2_status’ = off) $\wedge$ (button_status’ = $\neg$ button_status)

Test Data Generation

- The pre/post-conditions should be converted into executable test assertions by refining them.
  - The preconditions are broken into disjunctive normal forms (DNF), by eliminating the disjunction ($\vee$) operator from their expressions.

  - Example:

    Decomposition of $(a \wedge b) \vee (p \wedge q)$ → two disjoint cases: $(a \wedge b)$ and $(p \wedge q)$.

- Each DNF expression is analyzed separately using the domain testing technique in order to generate the test cases, which can be collected using a domain matrix.

Example: Test data generation for lighting system

- **Partial Pre/postcondition pair for t1:**

pre_push1 = pred(Light_Off) ∧ pred (C1) = (lamp1_status = off) ∧ (lamp2_status = off)
post_push1= pred(Lamp1_On) ∧ pred (switch)
= (lamp1_status’ = on) ∧ (lamp2_status’ = off) ∧ (button_status’ = ¬ button_status)

- **Conversion of the precondition pre_push1 into DNF:**

-No conversion is required because the predicate is already in a DNF format consisting of two conjuncts: (conj1: lamp1_status = off) and (conj2: lamp2_status = off)

- **Refinement into executable test assertion:** consists of bringing the predicate expressions into adequacy with the implementation language used (e.g., Java, C etc.).

-We need to derive adequate Java assertions that can be used to test the target implementation given previously.

- In this case, we simply need to harmonize the **variables** and **operators** involved in the expressions to fit the target program. Doing so lead to the following Java assertions:

conj1 = (lamp1Status **== false**); conj2 = (lamp2Status **== false**)
post_push1= (lamp1Status’ **==true**) **&** (lamp2Status’ **==false**)
& (buttonStatus’ **== !** buttonStatus)

•Test Case Generation Using Domain Matrix:

Variable	**Condition**	**Type**	**Test cases**			
			1	**2**	**3**	**4**
lamp1Status	lamp1Status =false	On	False			
		Off		True		
	Typical	In			False	False
lamp2Status	lamp2Status =false	On			False	
		Off				True
	Typical	In	False	False		
buttonStatus	Typical	In	False	True	True	False
Expected Result			True	Undefined	True	Undefined

-**True** for the expected result means that IUT should accept the test input and produce correct results, which here corresponds to the outcome of the evaluation of the postcondition.

-When the precondition is false, the postcondition can be anything (true or false); so the test case is considered **undefined** in this particular case.

-Finally, for this partial pre/postcondition pair, we obtain only 2 valid test cases: 1 & 3.

Test Execution

-Test execution can be done by generating appropriate test harness and/or using a test execution tool such as JUnit or Rational Tester.

-Basic steps for test execution may include the following:

1. Creating a fresh object, and setting its source state using the test values generated.
2. Then, the method under testing is executed, and the outcome of the target state is observed. Object state can be set and observed by using mutator and accessor methods (e.g. set/get methods).
3. Evaluate the execution outcome and report the result.
4. Class testing is conducted by testing all the methods involved in the class.

Example: Test execution for the lighting system

- **Consider test case 1 from the domain matrix:**

$$tc_1 = \left(input : \left| \begin{array}{l} lamp1Status = false \\ lamp2Status = false \\ buttonStatus = false \end{array} \right|, output : \left| post = true \right| \right)$$

·Test case execution code:

```
public boolean execute() {
        boolean l1, l2, b; //instance variables for class under testing
        boolean result = false; // test result

        // Create a   fresh object, and set its source state using the test values generated#tc1
        Lighting l = new Lighting(false,false,false);

        //Execute the method under testing and observe the outcome
        l.push();
        l1 = l.getLamp1Status();
        l2 = l.getLamp2Status();
        b = l.getButtonStatus();

        //Evaluate the outcome of the execution using the postcondition
        if( (l1 ==true)  && (l2=false) && (b == true)) then result = true;

        return result;
}
```

```
             …
//Alternative code
l.push();
Assert (l1 ==true)  &&
        (l2=false) &&
        (b == true));
             …
}
```

Issue:

-Compilation: can you observe the state of object l (i.e., code getXXX…)?