Software Engineering Testing

Testing State Behavior of Software

Testing State Behavior of Software

- Over the last 25 years, many suggestions have been made for creating Finite State Machines (FSM) and how to test software based on the FSM
- FSM has filled entire textbooks, and authors have gone into great depth and effort to define:
  - what exactly goes into a state
  - what can go onto edges
  - and what causes transitions
- FSMs are essentially graphs, and the graph testing criteria already defined can be used to test software that is based on the FSM
- Logic coverage testing criteria can also be used to test software that is based on the FSM

Testing State Behavior of Software (cont.)

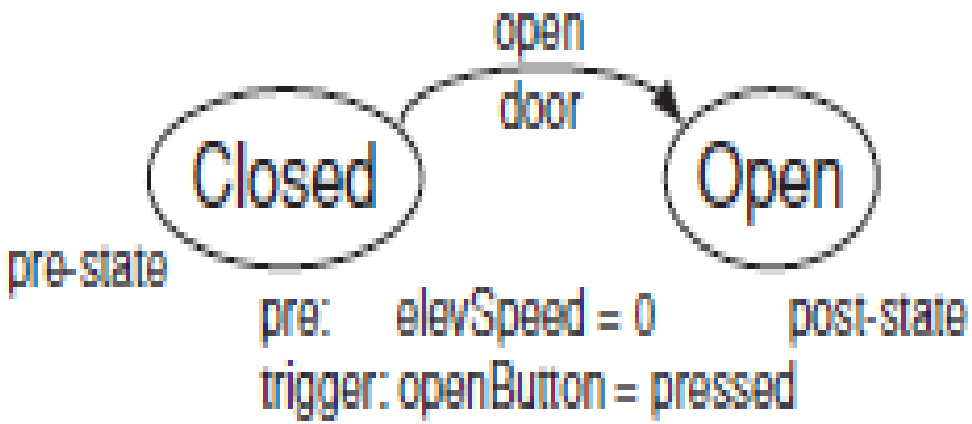
- Advantages of basing tests on FSMs:
  - huge numbers of practical software applications are based on a FSM model or can be modeled as FSMs
  - Virtually all embedded software fits in this category, including software in remote controls, household appliances, watches, cars, cell phones, airplane flight guidance ….
  - Indeed, most software can be modeled with FSMs
- On the other hand, Word processors, for example, contain so many commands and states that modeling them as FSMs is probably impractical

FSM – A quick recollection

- A finite state machine is a graph whose nodes represent states in the execution behavior of the software and edges represent transitions among the states
- A state is defined by specific values for a set of variables (Note that these variables are defined at the design modeling level and may not necessarily correspond to variables in the software)
- A transition is thought of as occurring in zero time and usually represents a change to the values of one or more variables
- When the variables change, the software is considered to move from the transition's *pre-state (predecessor) to its post-state (successor).*

A simple example

- A simple transition that opens an elevator door. If the elevator button is pressed (the triggering event), the door opens only if the elevator is not moving (the precondition, *elevSpeed = 0).*



Graph Coverage of FSM

- Node coverage requires that each state in the FSM be visited at least once and is called *state coverage*
- Edge coverage is applied by requiring that each transition in the FSM be visited at least once, which is called *transition coverage*
- The edge-pair coverage criterion was originally defined for FSMs and is also called *transition-pair* and *two-trip*
- The data flow coverage criteria are a bit more troublesome for FSMs. In most formulations of FSMs, nodes are not allowed to have defs or uses of variables
- All-Defs and All-Uses criteria can only be applied meaningfully to variables involved in guards

Logic Coverage of FSM

- Each transition has a pre-state and a post-state
- FSMs usually model behavior of the software and can be more or less formal and precise, depending on the needs and inclinations of the developers
- The most common way to apply logic coverage criteria to FSMs is to use logical expressions from the transitions as predicates
- In the Elevator example, the trigger and thus the predicate is *openButton = pressed*
- *Tests are created by* applying the criteria discussed (last week) to these predicates.

A more detailed Example

- Consider the example in Figure 3.5 (Page 134 onwards of Ammann book)
- This FSM models the behavior of the memory seat in a car (Lexus 2003 ES300)
- The memory seat has two configurations for two separate drivers and
  - controls the side mirrors (sideMirrors),
  - the vertical height of the seat (seatBottom),
  - the horizontal distance of the seat from the steering wheel (seatBack),
  - and the lumbar support (lumbar)
- The intent is to remember the configurations so that the drivers can conveniently switch configurations with the press of a button
- Each state in the figure has a number for efficient reference

Question 1:

- For the Memory Seat finite state machine (Figure 3.5), complete the test sets for the following coverage criteria by satisfying the predicates, ensuring reachability, and computing the expected output.
  - Predicate Coverage
  - Correlated Active Clause Coverage (CACC)

Question 2:

- Redraw Figure 3.5 to have fewer transitions, but more clauses. Specifically, nodes 1, 2, 4, and 5 each has four transitions to node 3. Rewrite these transitions to have only one transition from each of nodes 1, 2, 4, and 5 to node 3, and the clauses are connected by ORs. Then derive tests to satisfy Predicate Coverage and CACC for the resulting predicates. How do these tests compare with the tests derived from the original graph?

Question 3:

- Consider the following deterministic finite state machine:

Current State	Condition	Next State
Idle	$a \vee b$	Active
Active	$a \wedge b$	Idle
Active	$\neg b$	WindDown
WindDown	a	Idle

(a) Draw the finite state machine.

(b) Find CACC tests for each transition from the Active state.

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008