SOFTWARE TESTING METHODOLOGY

UNIT-VII

OVERVIEW

• A state graph and its associated state table are useful models for describing software (program) behavior. The finite state machine can be used as a functional (behavioral) testing tool as well as a tool for designing a testable program.

state graph

• A state graph is a graphical representation of the program (its FSM) in terms of states, transitions, inputs and outputs (erroneous or normal). It has one start state and usually, an end/destination/exit state.

• Note => In the exam you may draw only 3 state graph for simplicity.

• State graph in the above example is used to model the behavior of the program that recognizes a string occurrence at the input. It can be used to design, implement and the testing of the program

A Property of a state graph

• State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).

• State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

State table
Very big state graphs are difficult to follow as the diagrams get complicated and links get entwined. It is more convenient to represent the state graph as a table called state table or state transition table.

Each row represents the transitions from the originating state. There is one column for each input symbol (erroneous input or normal input). The entry in the table represents the new state to which the system transits to on this transition and the output it prints on the target printer device or on the output side.

A Property of a state graph

- State graphs are not dependent on time or temporal behavior or the program. (Temporal behavior is represented by some time sequence diagrams etc..) The system changes state only when an event (with an input sequence occurs or an epsilon symbol representing no event appears at the input of a transition).

- State graphs (FSM) are implemented as state tables which are represented in software with definite data structures and associated operations.

Software implementation of state table

- There are four tables that are needed.

1. A table or a process that encodes the input values into a compact list (INPUT_CODE_TABLE)

2. A table that specifies the next state for every combination of state and input code. (TRANSITION_TABLE)

3. A table or case statement that specifies the output (or output code) associated with every state-input combination (OUTPUT_TABLE)
4. A table that stores the present state of each device or process or component or system that uses the same state table. (DEVICE_TABLE)

Software implementation of state table

The process of usage of state table is as follows:

1. the present state is fetched from the memory (from DEVICE_TABLE).

2. the present input value (symbol) is fetched from the environment. It is encoded if it is non-numerical by using the INPUT_CODE_TABLE.

3. The present state and the input code are concatenated to give a pointer (row,column) into a cell of the TRANSITION_TABLE.

4. The OUTPUT_TABLE contains a pointer to the routine to be executed when that state-input combination occurs.

5. The same pointer value is used to fetch the new state value, which is then stored in DEVICE_TABLE

Principles of state testing

As it is not possible to test every path thru a state graph, use the notion of coverage. We assume that the graph is strongly connected.

1. It is not useful or practical to plan an entire grand tour of the states for testing initially as it does not work out due to possibility of bugs.

2. During the maintenance phase only few transitions and states need to be tested which are affected.

3 For very long test input symbol sequences it is difficult to test the system.
Uses/Advantages of state testing

- State testing can find bugs which are not possible to be found with other types of testing. Normally most of systems can be modeled as state graphs.

- It can find if the specifications are complete and ambiguous. This is seen clearly if the state table is filled with multiple entries in some cells or some cells are empty. It can also tell if some default transitions or transitions on erroneous inputs are missing.

- State testing can identify the system’s seemingly impossible states and checks if there are transitions from these states to other states are defined in the specifications or not. That is, the error recovery processes are defined for such impossible states.

- State testing can simplify the design of the program / system by identifying some equivalent states and then merging these states. Also, state testing using FSM can allow design/test design in a hierarchical manner if the state tables are so designed.

- The state testing can identify if the system reaches a dead state / unreachable states and allow one to correct the program specifications and make the system complete, robust and consistent.

- The bugs in the functional behavior can be caught earlier and will be less expensive if state testing is done earlier than the structural (white box) testing.
DisAdvantages of state testing

• Temporal behavior is not tested.

• There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.

• State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.

• Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.

• We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.

• Temporal behavior is not tested.

• There could be encoding errors in inputs, outputs, states, input-state combinations, identifying the number of states and merger of equivalent states. All these errors are not always easy to detect and correct.

• State transition testing does not guarantee the complete testing of the program. How much of testing with how many combinations of input symbol sequences constitutes sufficient number of tests is not clear/known. It is not practical to test thru every path in the state graph.

• Functional behavior is tested and structural bugs are not tested for. There could be difficulty if those bugs are found and mixed up with behavioral bugs.

• We assume that the state graph is strongly connected that is every node is connected to every other node thru a path.
Application areas for state testing using FSM model

Any program that processes input as a sequence of events/symbols and produces output such as detection of specified input symbol combinations, sequential format verification, parsing, etc. (compilers & translators).

- Communication Protocols: processing depends on current state of the protocol stack, OS, network environment and the message received
- concurrent systems,
- system failures and the corresponding recovery systems,
- distributed data bases,
- device drivers – processing depends on the state of the device &

- operation requested by the user or system
  - multi-tasking systems,
  - human computer interactive systems,
  - resource management systems – processing depends on availability

- levels and states of resources
  - Processing of hierarchical pop-up menus on windows based software systems – letting the user navigate thru menus
  - the web based application software, embedded systems and other systems also use this model for design and testing.
Good-state graphs and Bad state graph

- The principles of judging whether a state graph is good or bad are:
  - the total number of states is equal to the product of possibilities of factors that make up the state. (i.e., number of permutations of all values of all attributes/properties of the system/component)
  - For every state and every input there is exactly one transition specified to exactly one, possibly the same, state.
  - For every transition there is one output action specified. That output could be trivial (epsilon), but at least one output does some thing sensible.
  - For every state there is a sequence of inputs that drives the system to the starting (same) state.
  - A good state graph has at least two input symbols. With one symbol only a limited number of useful graphs are possible.
  - Bad state graphs contain states not reachable. It is not possible to reach every state from every other state. It is not possible to reach start state from itself.