Chapter 6

Test Generation from Finite State Models

The purpose of this chapter is to introduce techniques for the generation of test data from finite state models of software designs. A fault model and three test generation techniques are presented. The test generation techniques presented are: the W-method, the Unique Input/Output method and the Wp method.

6.1. Software design and testing

Development of most software systems includes a design phase. In this phase the requirements serve as the basis for a design of the application to be developed. The design itself can be represented using one or more notations. Finite state machines, state charts, and Petri nets are three design notations that are used in this chapter.

A simplified software development process is shown in Figure 6.1. Software analysis and design is a step that ideally follows requirements gathering. The end result of the design step is the *design* artifact that expresses a high level application architecture and interactions amongst low level application components. The design is usually expressed using a variety of notations such as those embodied in the UML design language. For example, UML state charts are used to represent the design of the real-time portion of the application and UML sequence diagrams are used to show the interactions between various application components.

The design is often subjected to test prior to its input to the coding step. Simulation tools are used to check if the state transitions depicted in the state charts do conform to the application requirements. Once the design has passed the test, it serves as an input to the coding step. Once the individual modules of the application have been coded, successfully tested, and debugged, they are integrated into an application and another test step is executed. This test step is known by various names such as *system test* and *design verification test*. In any case, the test cases against which the application is run can be derived from the a variety of sources,

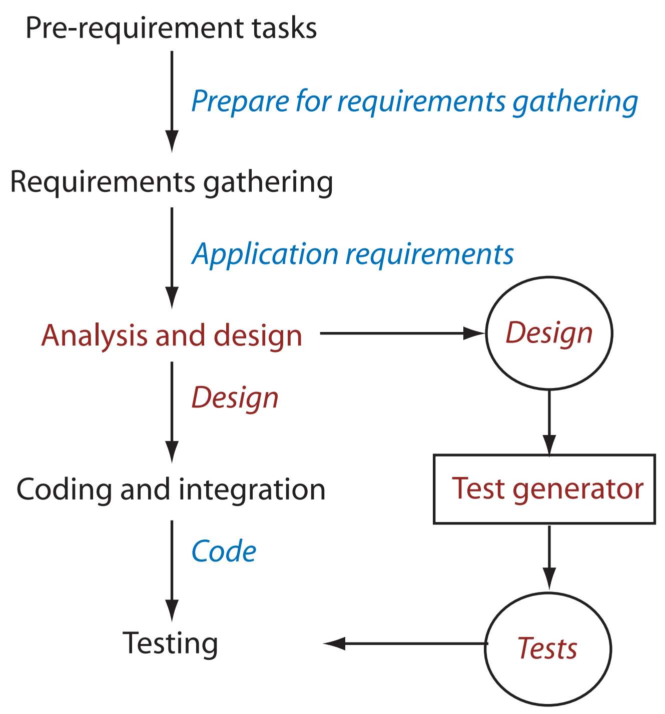

Figure 6.1: Design and test generation in a software development process. A *design* is an artifact generated in the Analysis and Design phase. This artifact becomes an input to the Test Generator algorithm that generates tests for input to the code during testing.

design being one of the sources.

In this chapter we will show how a design can be serve as source of tests that are used to test the application itself. As shown in Figure 6.1, the design generated at the end of the analysis and design phase serves as an input to a *test generation* procedure. This test generation procedure generates a number of tests that serve as inputs to the code in the test phase. Note that though Figure 6.1 seems to imply that the test generation procedure is applied to the entire design, this is not necessarily true; tests can be generated using portions of the design.

We introduce several test generation procedures to derive tests from finite state machines, and state charts. The finite state machine offers a simple way to model state-based behavior of applications. The state chart is a rich extension of the finite state machine and needs to be handled differently when used as an input to a test generation algorithm. The Petri net is a useful formalism to express concurrency and timing and leads to yet another set of algorithms for generating tests. All test generation methods described in this chapter can be automated though only some have been integrated into commercial test tools.

There exist several algorithms that take a finite state machine and some attributes of its implementation as inputs to generate tests. Note that the finite state machine serves as a source for test generation and is not the item under test. It is the implementation of the finite state machine that is under test. Such an implementation is also known as *Implementation Under Test* and abbreviated as IUT. For example, an FSM may represent a model of a communication protocol while an IUT is its implementation. The tests generated by the algorithms we introduce in this chapter are input to the IUT to determine if the IUT behavior conforms to the requirements.

In this chapter we shall introduce the following methods: the W-method, the transition tour method, the distinguishing sequence method, the unique input/output method, and the partial-W method. In Section 6.9 we shall compare the various methods introduced. Before we proceed to describe the test generation procedures, we introduce a fault model for FSMs, the characterization set of an FSM and a procedure to generate this set. The characterization set is useful in understanding and implementing the test generation methods introduced subsequently.

6.2. Finite state machines

Many devices used in daily life contain embedded computers. For example, an automobile has several embedded computers to perform various tasks, engine control being one example. Another example is a computer inside a toy for processing inputs and generating audible and visual responses. Such devices are also known as *embedded systems*. An embedded system can be as simple as a child's musical keyboard or as complex as the flight controller in an aircraft. In any case, an embedded system contains one or more computers for processing inputs.

An embedded computer often receives inputs from its environment and responds with appropriate actions. While doing so, it moves from one state to another. The response of an embedded system to its inputs depends on its current state. It is this behavior of an embedded system in response to inputs that is often modeled by a *finite state machine*. Relatively simpler systems, such as protocols, are modeled using finite state machines. More complex systems, such as the engine controller of an aircraft, are modeled using *statecharts* which can be considered as a generalization of finite state machines. In this section we focus on finite state machines; statecharts are described in Section 7.1. The next example illustrates a simple model using a finite state machine.

EXAMPLE 6.1. Consider a traditional table lamp that has a three-way rotary switch. When the switch is turned to one of its positions, the lamp is in the OFF state. Moving the switch clockwise to the next position moves the lamp to an ON-DIM state. Moving the switch further clockwise moves the lamp to the ON-BRIGHT state. Finally, moving the switch clockwise one more time brings the lamp back to the OFF state. The switch serves as the input device that controls the state of the lamp. The change of lamp state in response to the switch position is often illustrated using a *state* diagram as in Figure 6.2(a).

Our lamp has three states OFF, ON_DIM, and ON_BRIGHT, and one input. Note that the lamp switch has three distinct positions though from a lamp user's perspective the switch can only be turned clockwise to its next position. Thus "turning the switch one notch clockwise" is the only input. Suppose that the switch also can be moved counterclockwise. In this latter case, the lamp has two inputs one corresponding to the clockwise motion and the other to the counterclockwise motion. The number of distinct states of the lamp remains three but the state diagram is now as in Figure 6.2(b). ■

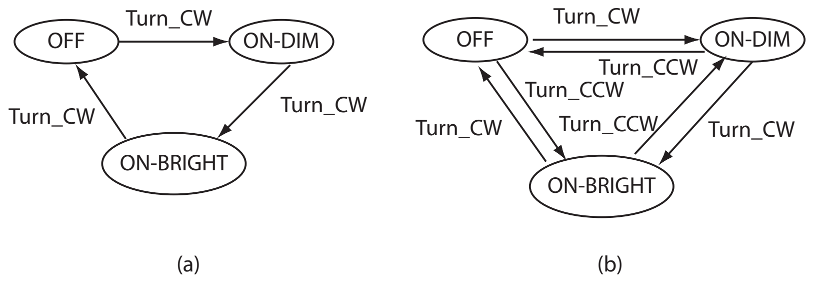

Figure 6.2: Change of lamp state in response to the movement of a switch. (a) Switch can only move one notch clockwise (CW). (b) Switch can move one notch clockwise and counterclockwise (CCW).

In the example above we have modeled the table lamp in terms of its states, inputs and transitions. In most practical embedded systems, the application of an input might cause the system to perform an *action* in addition to performing a state transition. The action might be as trivial as “do nothing” and simply move to another state, or a complex function might be executed. The next example illustrates one such system.

EXAMPLE 6.2. Consider a machine that takes a sequence of one or more unsigned decimal digits as input and converts the sequence to an equivalent integer. For example, if the sequence of digits input is 3, 2, and 1, then the output of the machine is 321. We assume that the end of the input digit sequence is indicated by the asterisk (*) character. We shall refer to this machine as the DIGDEC machine. The state diagram of DIGDEC is shown in Figure 6.3.

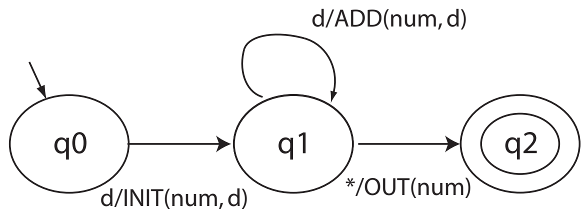

Figure 6.3: State diagram of the DIGDEC machine that converts sequence of decimal digits to an equivalent decimal number.

The DIGDEC machine can be any one of three states. It begins its operation in state q_0. Upon receiving the first digit, denoted by `d` in the figure, it invokes the function INIT to initialize variable `num` to `d`. In addition, the machine moves to state q_1 after performing the INIT operation. In q_1 DIGDEC can receive a digit or an end-of-input character which in our case is the asterisk. If it receives a digit, it updates `num` to $10 * \texttt{num} + d$ and remains in state q_1. Upon receiving an asterisk, the machine executes the OUT operation to output the current value of `num` and moves to state q_2. Notice the double circle around state q_2. Often such a double circle is used to indicate a *final* state of an FSM. ■

Historically, FSMs that do not associate any action with a transition are known as *Moore* machines. In Moore machines, actions depend on the current state. In contrast, FSMs that do associate actions with each state transition are known as *Mealy* machines. In this book we are concerned with Mealy machines. In either case, an FSM consists of a finite set states, a set of inputs, a start state, and a transition function which can be defined using the state diagram. In addition, a Mealy machine has a finite set of outputs. A formal definition of a Mealy machine follows.

Finite State Machine:

A finite state machine is a six tuple $(X, Y, Q, q_0, \delta, O)$ *where*

- X is a finite set of input symbols also known as the *input alphabet*,
- Y is a finite set of output symbols also known as the *output alphabet*,
- Q is a finite set states,
- $q_i \in Q$ is the initial state,
- $\delta : Q \times X \to Q$ is a next-state or state transition function, and
- $O : Q \times X \to Y$ is an output function.

In some variants of FSM more than one state could be specified as an initial state. Also, sometimes it is convenient to add $F \subseteq Q$ as a set of *final* or *accepting* states while specifying an FSM. The notion of accepting states is useful when an FSM is used as an automaton to recognize a language. Also note that the definition of the transition function δ implies that for any state q_i in Q, there is at most one next state. Such FSMs are also known as *deterministic* FSMs. In this book we are concerned only with deterministic FSMs. The state transition function for non-deterministic FSMs is defined as

$\delta : Q \times X \to 2^Q$

which implies that such an FSM can have more than one possible transition out of a state on the same symbol. Non-deterministic FSMs are usually abbreviated as NFSM or NDFSM or simply as NFA for Non-deterministic Finite Automata. We will use variants of NFSMs in Chapter 8 in the context of algorithms for generating test cases to test for the timing constraints in real-time systems.

The state transition and the output functions are extended to strings as follows. Let q_i and q_j be two, possibly same, states in Q and $s = a_1a_2 \ldots a_n$ a string over the input alphabet X of length $n \geq 0$. $\delta(q_i, s) = q_j$ if $\delta(q_i, a_1) = q_k$ and $\delta(q_k, a_2a_3 \ldots a_n) = q_j$. The output function is extended similarly. Thus $O(q_i, s) = O(q_i, a_1).O(\delta(q_i, a_1), a_2, \ldots, a_n)$. Also, $\delta(q_i, \epsilon) = q_i$ and $O(q_i, \epsilon) = \epsilon$.

Table 6.1 contains a formal description of the state and output functions of FSMs given in Figures 6.2 and 6.3. There is no output function for the table lamp. Note that the state transition function δ and the output function O can be defined by a *state diagram* as in Figures 6.2 and 6.3. Thus, for example, from Figure 6.2(a), we can derive δ(OFF, Turn_CW)=ON_DIM. As an example of the output function, we get O = INIT(num, 0) from Figure 6.3.

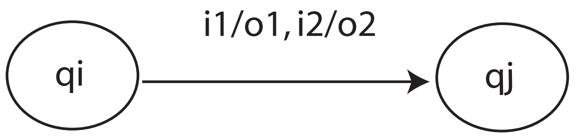

Figure 6.4: Multiple labels for an edge connecting two states in an FSM.

A state diagram is a directed graph that contains nodes representing states and edges representing state transitions and output functions. Each node is labeled with the state it represents. Each directed edge in a state diagram connects two states.

Each edge is labeled i/o where i denotes an input symbol that belongs to the input alphabet X and o denotes an output symbol that belongs to the output alphabet Y. i is also known as the *input portion* of the edge and o its *output portion*. Both i and o might be abbreviations. As an example, the edge connecting states q_0 and q_1 in Figure 6.3 is labeled $d/INIT(num, d)$ where d is an abbreviation for any digit from 0 to 9 and INIT(num,d) is an acttion.

Multiple labels can also be assigned to an edge. For example the label $i_1/o_1, i_2, /o_2$ associated with an edge that connects two states q_i and q_j implies that the FSM can move from q_i to q_j upon receiving either i_1 or i_2. Further, the FSM outputs o_1 if it receives input i_1 and outputs o_2 if it receives o_2. The transition and the output functions can also be defined in tabular form as explained in Section 6.2.2.

6.2.1. Excitation using an input sequence

In most practical situations an implementation of an FSM is excited using a sequence of input symbols drawn from its input alphabet. For example, suppose that the table lamp whose state diagram is in Figure 6.2(b) is in state ON-BRIGHT and receives the input sequence r where

r=Turn_CCW Turn_CCW Turn_CW

Using the transition function in Figure 6.2(b) it is easy to verify that the sequence s will move the lamp from state ON_BRIGHT to state ON_DIM. This state transition can also be expressed as a sequence of transitions given below.

δ(ON_BRIGHT, Turn_CCW)= ON_DIM

δ(ON_DIM, Turn_CCW)= OFF

δ(OFF, Turn_CW)= ON_DIM

For brevity, we will use the notation $T(q_k, z) = q_j$ to indicate that an input sequence z of length one or more moves an FSM from state q_k to state q_j. Using this notation we can write δ(ON_BRIGHT,r)=ON_DIM for the state diagram in Figure 6.2(b).

We will use a similar abbreviation for the output function O. For example, when excited by the input sequence 1001*, the DIGDEC machine ends up in state q_2 after executing the following sequence of actions and state transitions:

$O(q_0, 1)$=INIT(num,1), $\delta(q_0, 1) = q_1$

$O(q_1, 0)$=ADD(num,0), $\delta(q_1, 0) = q_1$

$O(q_1, 0)$=ADD(num,0), $\delta(q_1, 0) = q_1$

$O(q_1, 1)$=ADD(num,1), $\delta(q_1, 1) = q_1$

$O(q_1, *)$=OUT(num), $\delta(q_1, *) = q_2$

Once again, for brevity, we will use the notation $O(q, r)$ to indicate the action sequence executed by the FSM on input r when starting in state q. We also assume that a machine remains in its current state when excited with an empty sequence. Thus $\delta(q, \epsilon) = q$. Using the abbreviation for state transitions we can express the above action and transition sequence as follows

$O(q_0, 1001*)$=INIT(num, 1) ADD(num,0) ADD(num,0) ADD(num, 1) OUT (num)

$\delta(q_0, 1001*) = q_2$

Table 6.1: Formal description of three FSMs from Figures 6.2 and 6.3.

	Figure 6.2 (a)	**Figure 6.2 (b)**	**Figure 6.3**
X	{Turn-CW}	{Turn-CW, Turn-CCW}	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *}
Y	None	None	{INIT, ADD, OUT}
Q	{OFF, ON-DIM, ON-BRIGHT}	{OFF, ON-DIM, ON-BRIGHT}	$\{q_0, q_1, q_2\}$
q_0	{OFF}	{OFF}	q_0
F	None	None	q_2
δ	See Figure 6.2(a)	See Figure 6.2(b)	See Figure 6.3
O	Not applicable	Not applicable	See Figure 6.3

6.2.2. Tabular representation

A table is often used as an alternative to the state diagram to represent the state transition function δ and the output function O. The table consists of two sub-tables that consist of one or more columns each. The leftmost subtable is the *output* or the *action* sub-table. The rows are labeled by the states of the FSM. The rightmost sub-table is the *next state* sub-table. The output sub-table, that corresponds to the output function O, is labeled by the elements of the input alphabet X. The next state sub-table, that corresponds to the transition function δ, is also labeled by elements of the input alphabet. The entries in the output sub-table indicate the actions taken by the FSM for a single input received in a given state. The entries in the next state sub-table indicate the state to which the FSM moves for a given input and in a given current state. The next example illustrates how the output and transition functions can be represented in a table.

EXAMPLE 6.3. The table given below shows how to represent functions δ and O for the

DIGDEC machine. In this table the column labeled "Action", defines the output function by listing the actions taken by the DIGDEC machine for various inputs. Sometimes this column is also labeled as "Output" to indicate that the FSM generates an output symbol when it takes a transition. The column labeled "Next State" defines the transition function by listing the next state of the DIGDEC machine in response to an input. Empty entries in the table should be considered as undefined. However, in certain practical situations, the empty entries correspond to an "error" action. Note that in the table below we have used the abbreviation `d` for the ten digits, 0 through 9. ■

Current state	**Action**		**Next state**	
	`d`	`*`	`d`	`*`
q_0	INIT (num, d)		q_1	
q_1	ADD (num, d)	OUT (num)	q_1	q_2
q_2				

6.2.3. Properties of FSM

FSMs can be characterized by one or more of several properties. Some of the properties found useful in test generation, are given below. We shall use these properties later in while explaining algorithms for test generation.

Completely specified FSM: An FSM M is said to be *completely specified* if from each state in M there exists a transition for each input symbol. The machine described in Example 6.1 with state diagram in Figure 6.2(a) is completely specified as there is only one input symbol and each state has a transition corresponding to this input symbol. Similarly, the machine with state diagram shown in Figure 6.2(b) is also completely specified as each state has transitions corresponding to each of the two input symbols. The DIGDEC machine in Example 6.2 is not completely specified as state q_0 does not have any transition corresponding to an asterisk and state q_2 has no outgoing transitions.

Strongly connected: An FSM M is considered *strongly connected* if for each pair of states (q_i, q_j) there exists an input sequence that takes M from state q_i to q_j. It is easy to verify that the machines in Example 6.1 are strongly connected. The DIGIDEC machine in Example 6.2 is not strongly connected as it is impossible to move from state q_2 to q_0, from q_2 to q_1, and from q_1 to q_0. In a strongly connected FSM, given some state $q_i \neq q_0$, one can find an input sequence $s \in X^*$ that takes the machine from its initial state to q_i. We therefore say that in a strongly connected FSM, every state is *reachable* from the initial state.

V-equivalence: Let $M_1 = (X, Y, Q_1, m_0^1, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m_0^2, T_2, O_2)$ be two FSMs. Let V denote a set of non-empty strings over the input alphabet X, i.e. $V \subseteq X^+$. Let q_i and q_j, $i \neq j$, be the states of machines M_1 and M_2, respectively. Statesq_i and q_j are considered V-equivalent if $O_1(u, s) = O_2(v, s)$ for all $s \in V$. Stated differently, states q_i and q_j are considered V-*equivalent* if M_1 and M_2 , when excited in states q_i and q_j, respectively, yield identical output sequences. States q_i and q_j are said to be *equivalent* if $O_1(q_i, r) = O_2(q_j, r)$ for any set V. If q_i and q_j are not equivalent then they are said to be *distinguishable*. This definition of equivalence also applies to states within a machine. Thus machines M_1 and M_2 could be the same machine.

Machine equivalence: Machines M_1 and M_2 are said to be equivalent if (a) for each state σ in $M1$ there exists a state σ' in $M2$ such that σ and σ' are equivalent and (b) for each state σ in $M2$ there exists a state σ' in $M1$ such that σ and σ' are equivalent. Machines that are not equivalent are considered distinguishable. If $M1$ and $M2$ are strongly connected then they are equivalent if their respective initial states, m_0^1 and m_0^2, are equivalent. We write $M_1 = M_2$ if machines M_1 and M_2 are equivalent and $M_1 \neq M_2$ when they are distinguishable. Similarly, we write $q_i = q_j$ when statesq_i and q_j are equivalent and $q_i \neq q_j$ if they are distinguishable.

k-equivalence: Let $M_1 = (X, Y, Q_1, m_0^1, T_1, O_1)$ and $M_2 = (X, Y, Q_2, m_0^2, T_2, O_2)$ be two FSMs. States $q_i \in Q_1$ and $q_j \in Q_2$ are considered *k-equivalent* if when excited by any input of length k, yield identical output sequences. States that are not *k-equivalent* are considered *k-distinguishable*. Once again, M_1 and M_2 may be the same machines implying that k-distinguishability applies to any pair of states of an FSM. It is also easy to see that if two states are k-distinguishable for any $k > 0$ then they are also distinguishable for any $n \geq k$. If M_1 and M_2 are not *k-distinguishable* then they are said to be *k-equivalent*.

Minimal machine: An FSM M is considered *minimal* if the number of states in M is less than or equal to any other FSM equivalent to M.

EXAMPLE 6.4. Consider the DIGDEC machine in Figure 6.3. This machine is not completely specified. However, it is often the case that certain erroneous transitions are not indicated in the state diagram. A modified version of the DIGDEC machine appears in Figure 6.5. Here we have labeled explicitly the error transitions by the output function ERROR( ). ■

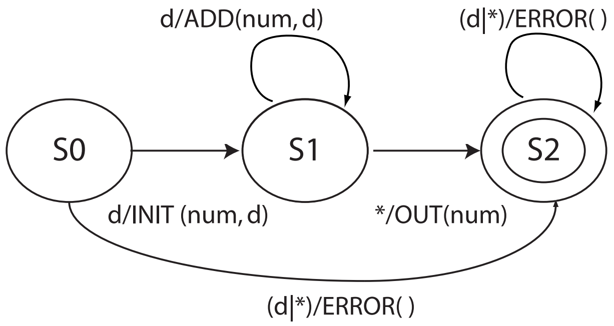

Figure 6.5: State diagram of a completely specified machine for converting a sequence a one or more decimal digits to their decimal number equivalent.

6.3. Conformance testing

The term *conformance testing* is used during the testing of communication protocols. An implementation of a communication protocol is said to conform to its specification if the implemen-