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,

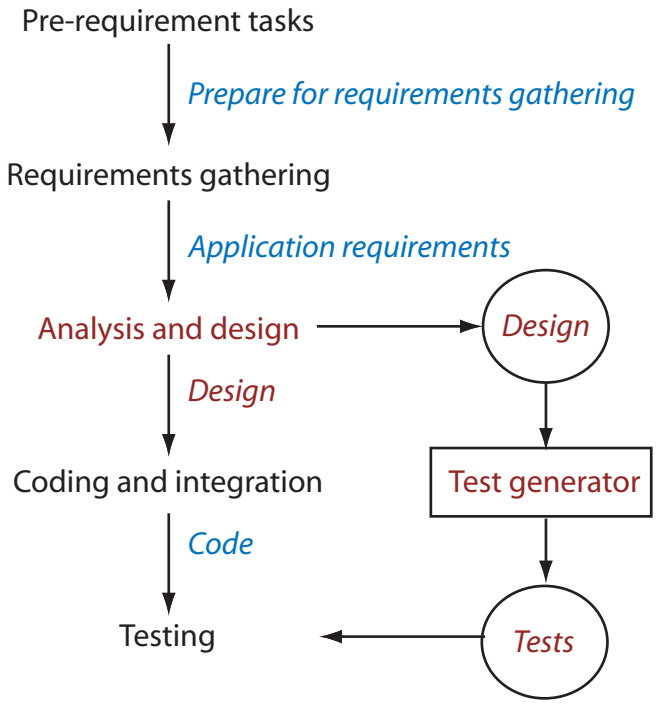


Figure 6.1: Design and test generation in a software development process. A *design* is an artifact generated in the Analysis and Design phase. This artifact becomes an input to the Test Generator algorithm that generates tests for input to the code during testing.

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, state charts, and Petri nets. 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 6.8 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. 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 implementation passes a collection of tests derived from the specification. In this chapter we introduce techniques for generating tests to test an implementation for conformance testing of a protocol modeled as finite state machine. Note that the term conformance testing applies equally well to the testing of any implementation to its specification, regardless of whether or not the implementation is that of a communication protocol.

Communication protocols are used in a variety of situations. For example, common use of protocols is in public data networks. These networks use access protocols such as the X.25 which is a protocol standard for wide area network communications. The Alternating Bit Protocol (ABP) is another example of a protocol used in the connection-less transmission of messages from a transmitter to a receiver. Musical instruments such as synthesizers and electronic pianos use the MIDI (Musical Instrument Digital Interface) protocol to communicate amongst themselves and a computer. These are just two examples of a large variety of communication protocols that exist and new ones are often under construction.

A protocol can be specified in a variety of ways. For example, it could be specified informally in textual form. It could also be specified more formally using a finite state machine, a visual notation such as a state chart, and using languages such as LOTOS (Language of Temporal Ordering Specification), Estelle, and SDL. In any case, the specification serves as the source for implementers and also for automatic code generators. As shown in Figure 6.2, an implementation of a protocol consists of a control portion and a data portion. The control portion captures the transitions in the protocol from one state to another. It is often modeled as a finite state machine. The data portion maintains information needed for the correct behavior. This includes counters and other variables that hold data. The data portion is modeled as a collection of program modules or segments.

Testing an implementation of a protocol involves testing both the control and data portions. The implementation under test is often referred to as IUT. In this chapter we are concerned with testing the control portion of an IUT. Techniques for testing the data portion of the IUT are described in other chapters. Testing the control portion of an IUT requires the generation of test cases. As shown in Figure 6.3, the IUT is tested against the generated tests. Each test

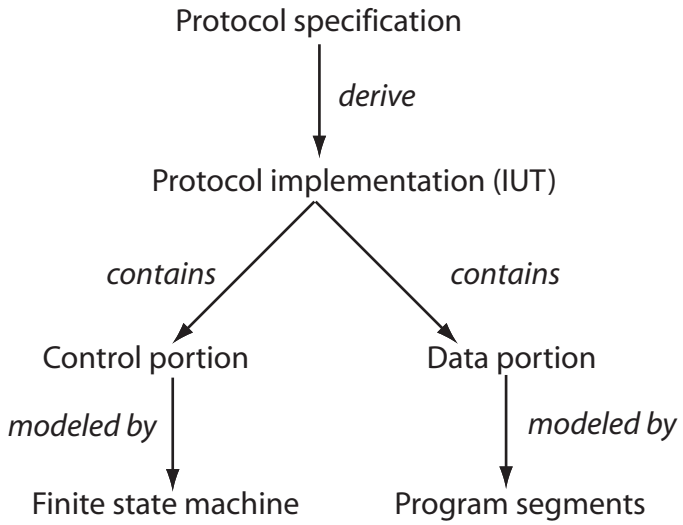


Figure 6.2: Relationship amongst protocol specification, implementation, and models.

is a sequence of symbols that are input to the IUT. If the IUT behaves in accordance with the specification, as determined by an oracle, then its control portion is considered equivalent to the specification. Non-conformance usually implies that the IUT has an error that needs to be fixed. Such tests of an IUT are generally effective in revealing certain types of implementation errors discussed in Section 6.3.

A significant portion of this chapter is devoted to describing techniques for generating tests to test the control portion of an IUT corresponding to a formally specified design. The techniques described for testing IUTs modeled as FSMs are applicable to protocols and other requirements that can be modeled as FSMs. Most complex software systems are generally modeled using state charts, and not FSMs. However, some of the test generation techniques described in this chapter, in particular the UIO method, are useful in generating tests from state charts.

6.2.1. Reset inputs

The test methods described in this chapter often rely on the ability of a tester to reset the IUT so that it returns to its start state. Thus, given a set of test cases $T = \{t_1, t_2, \ldots, t_n\}$, a test proceeds as follows:

1. Bring the IUT to its start state. Repeat the steps below each test input in T.
2. Select the next test case from T and apply it to the IUT. Observe the behavior of the IUT and compare it against the expected behavior as shown in Figure 6.2. The IUT is said to have failed if it generates an output different from the expected output.
3. Bring the IUT back to its start state by applying the reset input and repeat the step above but with the next test input.

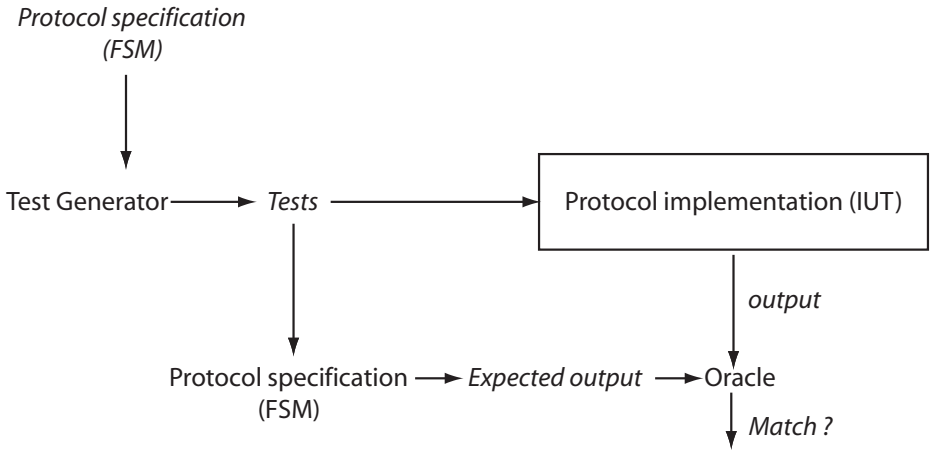


Figure 6.3: A simplified procedure for testing a protocol implementation against an FSM model. Italicized items represent data input to some procedure. Note that the model itself serves as data input to the test generator and as a procedure for deriving the expected output.

It is usually assumed that the application of the reset input generates a `null` output. Thus for the purpose of testing an IUT against its control specification FSM $M = (X, Y, Q, q_1, \delta, O)$, the input and output alphabets are augmented as follows:

$X = X \bigcup \{Re\}$
$Y = Y \bigcup \{\texttt{null}\}$

While testing a software implementation against its design, a reset input might require executing the implementation from the beginning, i.e. restarting the implementation, manually or automatically via, for example, a script. However, in situations where the implementation causes side effects, such as writing to a file or sending a message over a network, bringing the implementation to its start state might be a non-trivial task if its behavior depends on the state of the environment. While testing continuously running programs, such as network servers, the application of a reset input does imply bringing the server to its initial state but not necessarily shutting it down and restarting.

The transitions corresponding to the reset input are generally not shown in the state diagram. In a sense these are hidden transitions that are allowed during the execution of the implementation. When necessary, these transitions could be shown explicitly as in Figure 6.4.

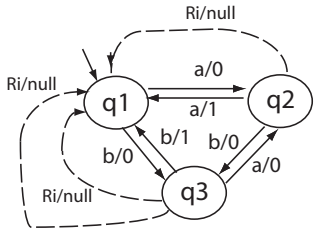


Figure 6.4: Transitions corresponding to reset inputs.

EXAMPLE 6.1. Consider the test of an application that resides inside a microwave oven. Suppose that one test, say t_1, corresponds to testing the "set clock time" functions. Another test, say t_2 tests for heating ability of the oven under "low power" setting. Finally the third test t_3 tests the ability of the oven under "high power" setting. In its start state, the oven has electrical power applied and is ready to receive commands from its keypad. We assume that the clock time setting is not a component of the start state. This implies that the current time on the clock, e.g. 1:15pm or 2:09am, does not have any affect on the state of the oven.

It seems obvious that the three tests mentioned above can be executed in any sequence. Further, once a test is completed and the oven application, and the oven, does not fail, the oven will be in its start state and hence no explicit reset input is required prior to executing the next test.

However, the scenario mentioned above might change after the application of , for example, t_2, if there is an error in the application under test or in the hardware that receives control commands from the application under test. For example, the application of t_2 is expected to start the ovens heating process. However, due to some hardware or software error, this might not happen and the oven application might enter a loop waiting to receive a "task completed" confirmation signal from the hardware. In this case, tests t_3 or t_1 cannot be applied unless the oven control software and hardware are reset to their start state.

It is due to situations such as the one described in the previous example, that require the availability of a reset input to bring the IUT under test to its start state. Bringing the IUT to its start state gets it ready to receive the next test input.

6.2.2. The testing problem

There are several ways to express the design of a system or a subsystem. Finite state machines (FSM), state charts, Petri Nets are some of the formalisms to express various aspects of a design. Protocols, for example, are often expressed as FSMs. Software designs are expressed using state charts. Petri nets are often used to express are used to express aspects of designs that relate to concurrency and timing. The design is used as a source for the generation of tests that are used subsequently to test the IUT for conformance to the design.

Let M be a formal representation of a design. As above, this could be an FSM, a state chart, or a Petri net. Other formalisms are possible too. Let R denote the requirements that led to M. R may be for a communication protocol, an embedded system such as the automobile engine and a heart pacemaker. Often, both R and $M1$ are used as the basis for the generation of tests and to determine the expected output as shown in Figure 6.3. In this chapter we are concerned with the generation of tests using M.

The testing problem is to determine if the IUT is *equivalent* to M. For the finite state machine representation, equivalence was defined earlier in Chapter 1. For other representations equivalence is defined in terms of the input/output behavior of the IUT and M. As mentioned earlier, the protocol or design of interest might be implemented in hardware, software, or a combination of the two. Testing the IUT for equivalence against M requires executing it against a sequence of test inputs derived from M and comparing the observed behavior with the expected behavior

as shown in Figure 6.3. Generation of tests from a formal representation of the design and their evaluation in terms of their fault detection ability is the key subject of this chapter.

6.3. A fault model

Given a set of requirements R, one often constructs a design from the requirements. A finite state machine is one possible representation of the design. Let M_d be a design intended to meet the requirements in R. Sometimes M_d is referred to as a *specification* that guides the implementation. M_d is implemented in hardware or software. Let M_i denote an implementation that is intended to meet the requirements in R and has been derived using M_d. Note that in practice $_i$ is unlikely to be an exact analog of M_d. In embedded real-time systems and communications protocols, it is often the case that M_i is a computer program that uses variables and operations not modeled in M_d. Thus one could consider M_d as a finite state model of M_i.

The problem in testing is to determine whether or not M_i conforms to R. To do so one tests M_i using a variety of inputs and checks for the correctness of the behavior of M_i with respect to R. The design M_d can be useful in generating a set T of tests for M_i. Tests so generated are also known as *black-box* tests because they have been generated with reference to M_d and not M_i. Given T, one tests M_i against each test $t \in T$ and compares the behavior of M_i withe the expected behavior given by exciting M_d in the initial state with test t.

In an ideal situation, one would like to ensure that any error in M_i is revealed by testing it against some $t \in T$ derived from M_d. One reason why this is not feasible is that the number of possible implementations of a design M_d is infinite. This gives rise to the possibility of a large variety of errors one could introduce in M_i. In the face of this reality, a *fault model* has been proposed. This fault model defines a small set of possible fault types that can occur in M_i. Given a fault model, the goal is to generate a test set T from a design M_d. Any fault in M_i, of the type in the fault model is guaranteed to be revealed when tested against T.

A widely used fault model for FSMs is shown in Figure 6.5. This figure illustrates four fault categories.

- *Operation error*: Any error in the output generated upon a transition is an operation error. This is illustrated by machines M and M_1 in Figure 6.5 where FSM M_1 generates an output 0, instead of a 1, when symbol `a` is input in state q_0. More formally, an operation error implies that for some state q_i in M and some input symbol s, $O(q_i, x)$ is incorrect.

- *Transfer error*: Any error in the transition from one state to the next is a transition error. This is illustrated by machine M_2 in Figure 6.5 where FSM M_2 moves to state q_1, instead of moving to q_0 fromq_0 when symbol `a` is input. More formally, a transfer error implies that for some state q_i in M and some input symbol s, $\delta(q_i, x)$ is incorrect.

- *Extra state error*: An extra state may be introduced in the implementation. In Figure 6.5, machine M_3 has q_2 as the extra state when compared with the state set of machine M. An extra state may or may not be an error. For example, in Figure 6.6, machine M represents the correct design. Machines M_1 and M_2 have one extra state each. However, M_1 is an erroneous design but M_2 is not. This is because M_2 is actually equivalent to M even though it has an extra state.

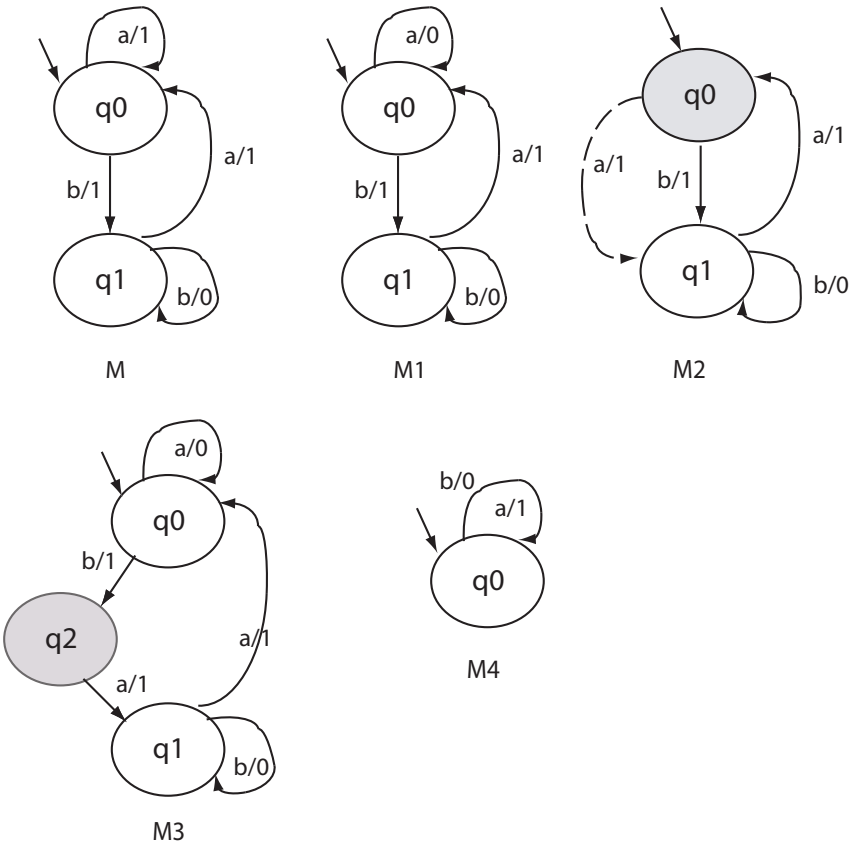


Figure 6.5: Machine M represents the correct design. Machines M_1, M_2 , M_3, and M_4 each contain an error.

- *Missing state error*: A missing state is another type of error. In Figure 6.5, machine M_4 has q_1 missing when compared with machine M_4. Given that the machine representing the design is minimal and complete, a missing state implies an error in the IUT.

The above fault model is generally used to evaluate a method for generating tests from a given FSM. The faults indicated above are also known collectively as *sequencing faults* or *sequencing errors*. It may be noted that a given implementation could have more than one error of the same type. Other possibilities, such as two errors one each of a different type, also exist.

6.3.1. Mutants of FSMs

As mentioned earlier, given a design M_d, one could construct many correct and incorrect implementations. Let $\mathcal{I}(M_d)$ denote the set of all possible implementations of M_d. In order to make $\mathcal{I}$ finite, we assume that any implementation in $\mathcal{I}(M_d)$ differs from M_d in known ways. One way to distinguish an implementation from its specification is through the use of *mutants*. A mutant of M_d is an FSM obtained by introducing one or more errors zero or more times. We assume that the errors introduced belong to the fault model described earlier. Using this method we see in Figure 6.5 four mutants M_1, M_2, M_3, and M_4 of machine M. More complex mutants can

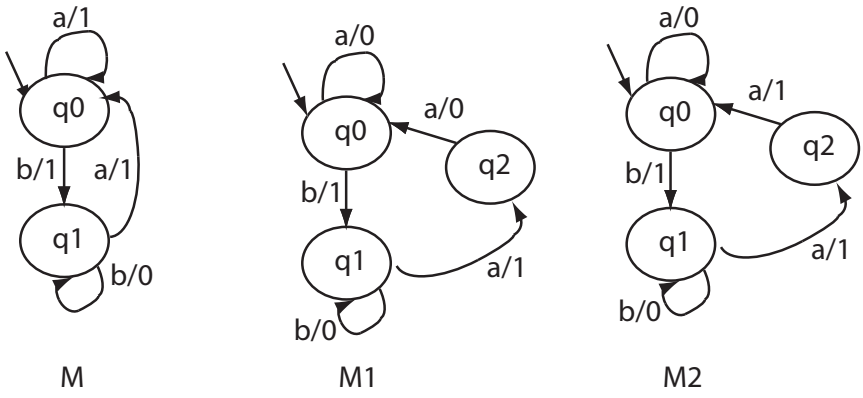


Figure 6.6: Machine M represents the correct design. Machines M_1 and M_2 each have an extra state. However, M_1 is erroneous and M_2 is equivalent to M.

also be obtained by introducing one or more errors in a machine. Figure 6.7 shows two such mutants of machine M of Figure 6.5.

Note that some mutant may be equivalent to M_d implying that the output behaviors of M_d and the mutant are identical on *all possible* inputs. Given a test set T, we say that a mutant is *distinguished* from M_d by a test $t \in T$ if the output sequence generated by the mutant is different from that generated by M_d when excited with t in their respective initial states.In this case we also say that Tv distinguishes the mutant.

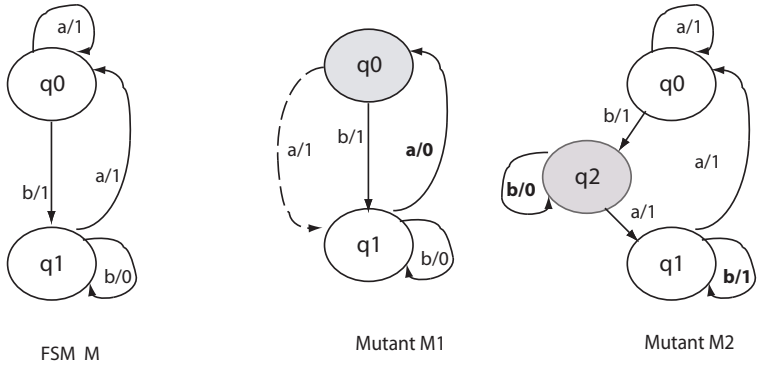


Figure 6.7: Two mutants of machine M. Mutant M_1 is obtained by introducing into M one operation error in state q_1 and one transfer error in state q_0 for input a. Mutant M_2 has been obtained by introducing an extra state q_2 in machine M and an operation error in state q_1 on input b.

Using the idea of mutation, one can construct a finite set of possible implementations of a given specification M_d. Of course, this requires that some constraints are placed on the process of mutation, i.e. on how the errors are introduced. WE caution the reader in that the technique of testing software using program mutation is a little different from the use of mutation

described here. We discuss this difference in Chapter 16. The next example illustrates how to obtain one set of possible implementations.

EXAMPLE 6.2. Let M shown in Figure 6.7 be an FSM that denotes the correct design. Suppose that we are allowed to introduce only one error at a time in M. This generates four mutants obtained by introducing an operation error. Considering that each state has two transitions, we get four mutants by introducing the transfer error.

Introducing an additional state to create a mutant is more complex. First we assume that only one extra state can be added to the FSM. However, there are several ways in which this state can interact with the existing states in the machine. This extra state can have two transitions that can be specified in 36 different ways depending on the tail state of the transition and the associated output function. We delegate to Exercise 6.7 the task of creating all the mutants of M by introducing an extra state.

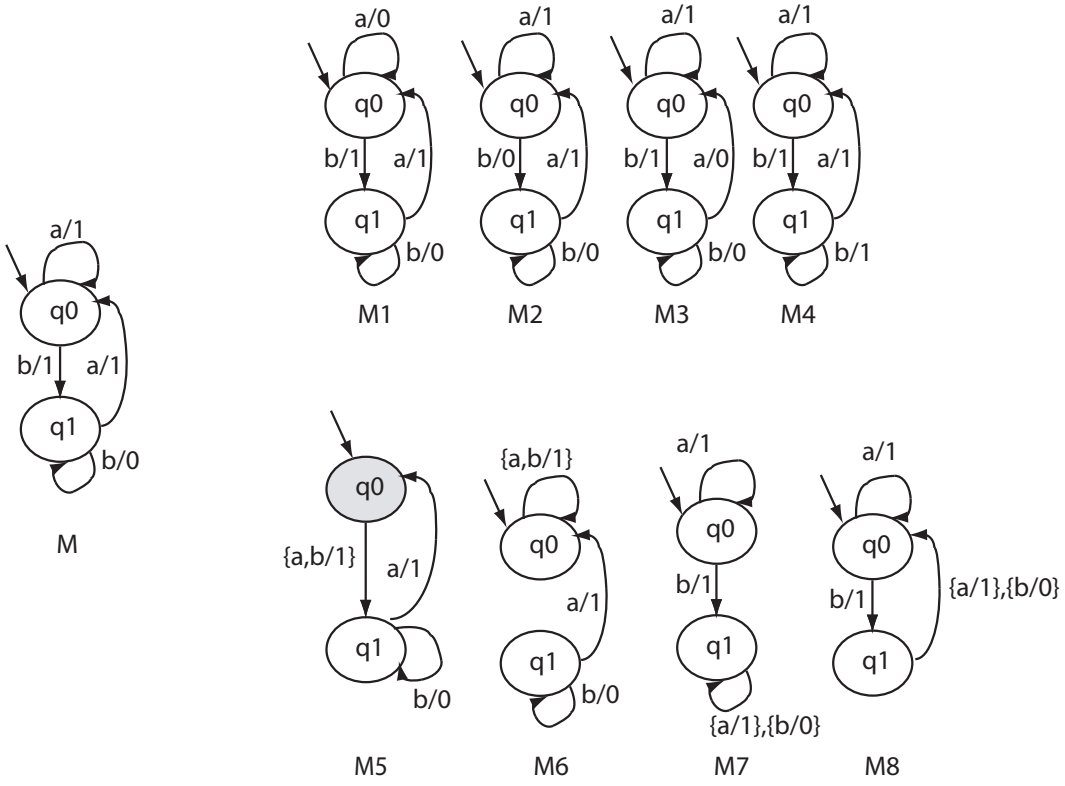


Figure 6.8: Eight first order mutants of M generated by introducing operation and transfer errors. Mutants M_1, M_2,M_3, and M_4 are generated by introducing operation errors in M. Mutants M_5, M_6, M_7, and M_8 are generated by introducing transfer errors in M.

Removing a state can be done in only two ways: remove q_0 or remove q_1. This generates two mutants. In general this could be more complex when the number of states is greater than three. Removing a state will require the redirection of transitions and hence the number of mutants generated will be more than the number of states in the correct design.

Any test input distinguishes M from M_1. Mutant M_2 is distinguished from M by the input sequence `ab`. Note that when excited with `ab`, M outputs the string 11 whereas M_2 outputs the string 10. Input `ab` also distinguishes M_5 from M. The complete set of distinguishing inputs can be found using the W-method described in Section 6.5.

6.3.2. Fault coverage

One way to determine the "goodness" of a test set is to compute how many errors it reveals in a given implementation M_i. A test set that can reveal all errors in M_i is considered superior to one that fails to reveal one or more errors. Methods for the generation of test sets are often evaluated based on their fault coverage. The fault coverage of a test set is measured as a fraction between 0 and 1 and with respect to a given design specification. We give a formal definition of fault coverage in the following where the fault model presented earlier is assumed to generate the mutants.

N_t: Total number of first order mutants of the machine M used for generating tests. This is also the same as $\mid \mathcal{I}(M) \mid$.

N_e: Number of mutants that are equivalent to M.

N_f: Number of mutants that are distinguished by test set T generated using some test generation method. Recall that these are the faulty mutants as no input in T is able to distinguish these mutants.

N_l: Number of mutants that are not distinguished by T.

The fault coverage of a test suite T with respect to a design M, and an implementation set $\mathcal{I}(M)$, is denoted by $FC(T, M)$ and computed as follows.

$$
\begin{aligned}
FC(T, M) &= \frac{\text{Number of mutants not distinguished by } T}{\text{Number of mutants that are not equivalent to } M} \\
&= \frac{N_t - N_e - N_f}{N_t - N_e}
\end{aligned}
$$

In Section 6.8 we will see how FC can be used to evaluate the goodness of various methods to generate tests from FSMs. Next we introduce the characterization set, and its computation, useful in various methods for generating tests from FSMs.

6.4. Characterization set

Most methods for generating tests from finite state machines make use of an important set known as the *characterization set*. This set is usually denoted by W and is sometimes referred to as the W-set. In this section we explain how one can derive a W-set given the description of an FSM. Let us examine the definition of the W-set.

Let $M = (X, Y, Q, q_1, \delta, O)$ be an FSM that is minimal and complete. A characterization set of M, denoted as W, is a finite set of input sequences that distinguish the behavior of any pair of states in M. Thus if q_i and q_j are states in Q, then there exists a string s in W such that $O(s, q_i) \neq O(s, q_j)$, where s belongs to X^+.

EXAMPLE 6.3. Consider an FSM $M = (X, Y, Q, q_1, \delta, O)$ where $X = \{a, b\}$, $Y = \{0, 1\}$, and $Q = \{q_1, q_2, q_3, q_4, q_5\}$, q_1 is the initial state. The state transition function δ and the output function O are shown in Figure 6.9. The set W for this FSM is given below.

$W = \{a, aa, aaa, baaa\}$

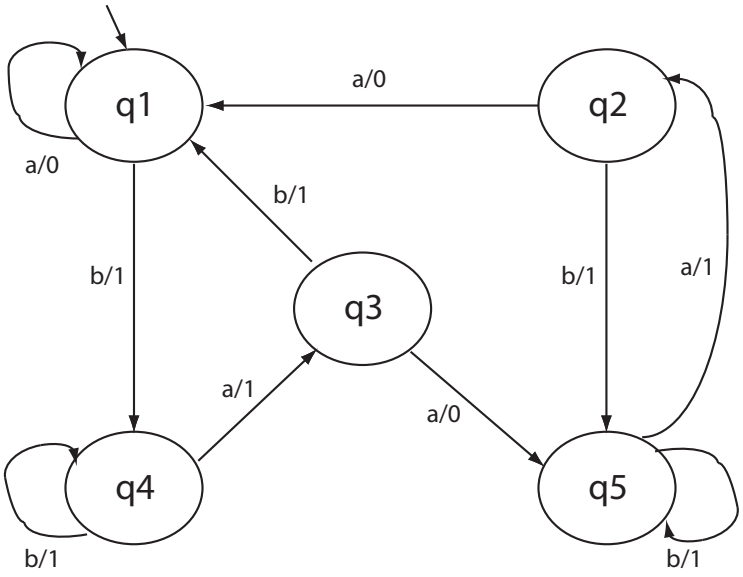


Figure 6.9: The transition and output functions of a simple FSM.

Let us do a sample check to determine if indeed W is the characterization set for M. Consider the string `baaa`. It is easy to verify from Figure 6.9 that when M is started in state q_1 and excited with `baaa`, the output sequence generated is 1101. However, when M is started in state q_2 and excited with input `baaa`, the output sequence generated is 1100. Thus we see that $O(\texttt{baaa}, q_1) \neq O(\texttt{baaa}, q_2)$ implying that the sequence `baaa` is a distinguishing sequence for states q_1 and q_2. You may now go ahead and perform similar checks for the remaining pairs of states.

The algorithm to construct a characterization set for an FSM M consists of two main steps. The first step is to construct a sequence of k-equivalence partitions $P_1, P_2, \ldots, P_m$ where $m > 0$. This iterative step converges in at most n steps where n is the number of states in M. In the second step these k-equivalence partitions are traversed, in reverse order, to obtain the distinguishing sequences for every pair of states. These two steps are explained in detail in the following two subsections.

6.4.1. Construction of the k-equivalence partitions

Recall from Chapter 1 that given $M = (X, Y, Q, q_1, \delta, O)$, two states $q_i \in Q$ and $q_j \in Q$ are considered *k-equivalent* if there does not exist an $s \in X^k$ such that $O(s, q_i) \neq O(s, q_j)$. The notion of *k-equivalence* leads to the notion of *k-equivalence partitions*.

Given an FSM $M = (X, Y, Q, q_1, \delta, O)$, a *k-equivalence partition* of Q, denoted by P_k, is a collection of n finite sets of states denoted as $\Sigma_{k\,1}, \Sigma_{k\,2}, \ldots, \Sigma_{k\,n}$ such that

- $\bigcup_{i=1}^{n} \Sigma_{k\,i} = Q$,
- States in $\Sigma_{k\,j}$, for $1 \leq j \leq n$, are *k*-equivalent,
- If $q_l \in \Sigma_{k\,i}$ and $q_m \in \Sigma_{k\,j}$, for $i \neq j$, then q_l and q_m must be *k*-distinguishable.

We now illustrate the computation of *k*-equivalence partitions by computing them for the FSM of Example 6.3 using a rather long example.

EXAMPLE 6.4. We begin by computing the 1-equivalence partition, i.e. P_1, for the FSM shown in Figure 6.9. To do so, we write the transition and output functions for this FSM in a tabular form as shown below.

Current state	**Output**		**Next state**	
	a	b	a	b
q_1	0	1	q_1	q_4
q_2	0	1	q_1	q_5
q_3	0	1	q_5	q_1
q_4	1	1	q_3	q_4
q_5	1	1	q_2	q_5

State transition and output table for M.

The next step in constructing P_1 is to regroup the states so that all states that are identical in their Output entries belong to the same group. We indicate the separation amongst two groups by a horizontal line as shown in the table below. Note from this table that states q_1, q_2, and q_3 belong to one group as they share identical output entries. Similarly, states q_4 and q_5 belong to another group. Notice also that we have added a Σ column to indicate group names. You may decide on any naming scheme for groups. In this example we have labeled the groups as 1 and 2.

Σ	**Current state**	**Output**		**Next state**	
		a	b	a	b
1	q_1	0	1	q_1	q_4
	q_2	0	1	q_1	q_5
	q_3	0	1	q_5	q_1
2	q_4	1	1	q_3	q_4
	q_5	1	1	q_2	q_5

State transition and output table for M with grouping indicated.

We have now completed the construction of P_1. The groups separated by the horizontal line

constitute a 1-equivalence partition. We have labeled these groups as 1 and 2. Thus we get the 1-equivalence partition as

$P_1 = \{1, 2\}$

Group $1 = \Sigma_{1\,1} = \{q_1, q_2, q_3\}$

Group $2 = \Sigma_{1\,2} = \{q_4, q_5\}$

In preparation to begin the construction of the 2-equivalence partition, we construct the P_1 table as follows. First we copy the "Next State" sub-table. Next we rename each Next State entry by appending a second subscript which indicates the group to which that state belongs. For example, the next state entry under column "a" in the first row is q_1. As q_1 belongs to group 1, we relabel this as q_{11}. Other next state entries are relabeled in a similar way. This gives us the P_1 table as shown below.

Σ	**Current state**	**Next state**	
		a	b
1	q_1	$q_{1\,1}$	$q_{4\,2}$
	q_2	$q_{1\,1}$	$q_{5\,2}$
	q_3	$q_{5\,2}$	$q_{1\,1}$
2	q_4	$q_{3\,1}$	$q_{4\,2}$
	q_5	$q_{2\,1}$	$q_{5\,2}$

P_1 table.

From the P_1 table given above, we construct the P_2 table as follows. First regroup all rows with identical *second* subscripts in its row entries under the Next State column. Note that the subscript we are referring to is the group label. Thus, for example, in $q_{4\,2}$ the subscript we refer to is 2 and not 4 2. This is because q_4 is the name of the state in machine M and 2 is the group label under the Σ column. As an example of regrouping in the P_1 table, the rows corresponding to the current states q_1 and q_2 have next states with identical subscripts and hence we group them together. This regrouping leads to additional groups. We now relabel the groups and update the subscripts associated with the next state entries. Using this grouping scheme, we get the following P_2 table.

Σ	**Current state**	**Next state**	
		a	b
1	q_1	$q_{1\,1}$	$q_{4\,3}$
	q_2	$q_{1\,1}$	$q_{5\,3}$
2	q_3	$q_{5\,3}$	$q_{1\,1}$
3	q_4	$q_{3\,2}$	$q_{4\,3}$
	q_5	$q_{2\,1}$	$q_{5\,3}$

P_2 table.

Notice that we have three groups in the P_2 table. We regroup the entries in the P_2 table using the scheme described earlier for generating the P_2 table. This regrouping and relabeling, gives us the following P_3 table.

Σ	**Current state**	**Next state**	
		a	b
1	q_1	$q_{1\,1}$	$q_{4\,3}$
	q_2	$q_{1\,1}$	$q_{5\,4}$
2	q_3	$q_{5\,4}$	$q_{1\,1}$
3	q_4	$q_{3\,2}$	$q_{4\,3}$
4	q_5	$q_{2\,1}$	$q_{5\,4}$

P_3 table.

Further regrouping and relabeling of P_3 table gives us the P_4 table given below.

Σ	**Current state**	**Next state**	
		a	b
1	q_1	$q_{1\,1}$	$q_{4\,4}$
2	q_2	$q_{1\,1}$	$q_{5\,5}$
3	q_3	$q_{5\,5}$	$q_{1\,1}$
4	q_4	$q_{3\,3}$	$q_{4\,4}$
5	q_5	$q_{2\,2}$	$q_{5\,5}$

P_4 table.

Note that no further partitioning is possible using the scheme described earlier. We have completed the construction of *k*-equivalence partitions, $k = 1, 2, 3, 4$, for machine M.

Notice that the process of deriving the P_k tables converged to P_4 and no further partitions are possible. See Exercise 6.3 that asks you to show that indeed the process will always converge. It is also interesting to note that each group in the P_4 table consists of exactly one state. This implies that all states in M are distinguishable. In other words, no two states in M are equivalent.

Next we summarize the algorithm to construct a characterization set from the k-equivalence partitions and illustrate it using an example.

6.4.2. Deriving the characterization set

Having constructed the k-equivalence partitions, i.e. the P_k tables, we are now ready to derive the W-set for machine M. Recall that for each pair of states q_i and q_j in M, there exists at least one string in W that distinguishes q_i from q_j. Thus the method to derive W proceeds by determining a distinguishing sequence for each pair of states in M. First we sketch the method, referred to as the W procedure, and then illustrate it by a continuation of the previous example. In the procedure below, $G(q_i, x)$ denotes the label of the group to which the machine moves when excited using input x in state q_i. For example, in the table for P_3 on page 187, $G(q_2, b) = 4$ and $G(q_5, a) = 1$.

The W-procedure
(A Procedure to derive W from a set of partition tables.)
Begin W procedure.

1. Let $M = (X, Y, Q, q_1, \delta, O)$ be the FSM for which $P = \{P_1, P_2, \ldots, P_n\}$ is the set of k-equivalence partition tables for $k = 1, 2, \ldots, n$. Initialize $W = \varnothing$.

2. Repeat the steps (a) through (d) for each pair of states (q_i, q_j), $i \neq j$, in M.

   a. Find a $r, 1 \leq r < n$ such that the states in pair (q_i, q_j) belong to the same group in P_r but not in P_{r+1}. In other words, P_r is the *last* of the P tables in which (q_i, q_j) belong to the same group. If such an r is found then move to Step (b), otherwise we find an $\eta \in X$ such that $O(q_i, \eta) \neq O(q_j, \eta)$, set $W = W \cup \{\eta\}$, and continue with the next available pair of states.

   The length of the minimal distinguishing sequence for (q_i, q_j) is $r + 1$. We denote this sequence as $z = x_0 x_1 \ldots x_r$ where $x_i \in X$ for $0 \leq i \leq r$.

   b. Initialize $z = \epsilon$. Let $p_1 = q_i$ and $p_2 = q_j$ be the *current* pair of states. Execute steps (i) through (iii) for $m = r, r-1, r-2, \ldots, 1$.

   (i) Find an input symbol η in P_m such that $G(p_1, \eta) \neq G(p_2, \eta)$. In case there is more than one symbol that satisfy the condition in this step, then select one arbitrarily.

   (ii) Set $z = z.\eta$.

   (iii) Set $p_1 = \delta(p_1, \eta)$ and $p_2 = \delta(p_2, \eta)$.

   c. Find an $\eta \in X$ such that $O(p_1, \eta) \neq O(p_2, \eta)$. Set $z = z.\eta$.

   d. The distinguishing sequence for the pair (q_i, q_j) is the sequence z. Set $W = W \cup \{z\}$.

End of the W procedure.

Upon termination of the W procedure we would have generated distinguishing sequences for all pairs of states in M. Note that the above algorithm is inefficient in that it derives a distinguishing sequence for each pair of states even though two pairs of states might share the same distinguishing sequence. (See Exercise 6.4.) The next example applies the W procedure to obtain W for the FSM in Example 6.4.

EXAMPLE 6.5. As there are several pairs of states in M, we illustrate how to find the distinguishing input sequences for the pairs (q_1, q_2) and (q_3, q_4). Let us begin with the pair (q_1, q_2).

According to Step 2(a) of the W procedure we first determine r. To do so we note from Example 6.4 that the last P-table in which q_1 and q_2 appear in the same group is P_3. Thus $r = 3$.

We now move to Step 2(b) and set $z = \epsilon, p_1 = q_1, p_2 = q_2$. Next, from P_3 we find that $G(p_1, b) \neq G(p_2, b)$ and update z to $z.b = b$. Thus `b` is the *first* symbol in the input string that distinguishes q_1 from q_2. In accordance with Step 2(b)(ii), we reset $p_1 = \delta(p_1, b)$ and $p_2 = \delta(p_2, b)$. This gives us $p_1 = q_4$ and $p_2 = q_5$.

Going back to Step 2(b)(i), we find from the P_2 table that $G(p_1, a) \neq G(p_2, a)$. We update z to $z.a = ba$ and reset p_1 and p_2 to $p_1 = \delta(p_1, a)$ and $p_2 = \delta(p_2, a)$. We now have $p_1 = q_3$ and $p_2 = q_2$. Hence `a` is the second symbol in the string that distinguishes states q_1 and q_2.

Once again we return to Step 2(b)(i). We now focus our attention on the P_1 table and find that states $G(p_1, a) \neq G(p_2, a)$ and also $G(p_1, b) \neq G(p_2, b)$. We arbitrarily select `a` as the third

Table 6.1: Distinguishing sequences for all pairs of states in the FSM of Example 6.4.

S_i	S_j	x	$o(S_i, x)$	$o(S_j, x)$
1	2	baaa	1	0
1	3	aa	0	1
1	4	a	0	1
1	5	a	0	1
2	3	aa	0	1
2	4	a	0	1
2	5	a	0	1
3	4	a	0	1
3	5	a	0	1
4	5	aaa	1	0

symbol of the string that will distinguish states q_1 and q_2. According to Step 2(b)(ii) we update z, p_1, and p_2 as $z = z.a = baa$, $p_1 = \delta(p_1, a)$ and $p_2 = \delta(p_2, a)$. We now have $p_1 = q_1$ and $p_2 = q_5$.

Finally we arrive at Step 2(c) and focus on the original state transition table for M. From this table we find that states q_1 and q_5 are distinguished by the symbol a. This is the last symbol in the string to distinguish states q_1 and q_2. We set $z = z.a = baaa$. We have now discovered baaa as the input sequence that distinguishes states q_1 and q_2. This sequence is added to W. Note that in Example 6.3 we have already verified that indeed baaa is a distinguishing sequence for states q_1 and q_2.

Next we select the pair (q_3, q_4) and resume at Step 2(a). We note that these two states are in a different group in table P_1 but in the same group in M. Thus we get $r = 0$. As $O(q_3, a) \neq O(q_4, a)$, the distinguishing sequence for the pair (q_3, q_4) is the input sequence a.

We leave it to you to derive distinguishing sequences for the remaining pairs of states. A complete set of distinguishing sequences for FSM M is given in Table 6.1. The leftmost two columns labeled S_i and S_j contain pairs of states to be distinguished. The column labeled x contains the distinguishing sequence for the pairs of states to its left. The rightmost two columns in this table show the *last* symbol output when input x is applied to state S_i and S_j, respectively. For example, as we have seen earlier $O(q_1, \texttt{baaa}) = 1101$. Thus, in Table 6.1, we only show the 1 in the corresponding output column. Notice that each pair in the last two columns is different, i.e. $o(S_i, x) \neq o(S_j, x)$ in each row. From this table we obtain $W = \{a, aa, aaa, baaa\}$.

6.4.3. Identification sets

Consider an FSM $M = (X, Y, Q, q_0, \delta, O)$ with symbols having their usual meanings and $\mid Q \mid = n$. Assume that M is completely specified and minimal. We know that a characterization set W for M is a set of strings such that for any pair of states q_i and q_j in M there exists a string s in W such that $O(q_i, s) \neq (q_j, s)$.

Analogous to the characterization set for M, we associate an *identification* set with each

state of M. An identification set for state $q_i \in Q$ is denoted by W_i and has the following properties: (a) $W_i \subseteq W$, $1 \leq i \leq n$, (b) $O(q_i, s) \neq O(q_j, s)$, for some j, $1 \leq j \leq n$, $j \neq i$, $s \in W_i$, and (c) no subset of W_i satisfies property (b). The next example illustrates the derivation of the identification sets given W.

EXAMPLE 6.6. Consider the machine given in Example 6.5 and its characterization set W shown in Table 6.1. From this table we notice that state q_1 is distinguished from the remaining states by the strings `baaa`, `aa`, and `a`. Thus $W_1 = \{baaa, aa, a\}$. Similarly, $W_2 = \{baaa, aa, a\}$, $W_3 = \{a, aa\}$, $W_4 = \{a, aaa\}$, and $W_5 = \{a, aaa\}$.

While the characterization sets are used in the W-method, the W_i sets are used in the Wp-method. The W- and the Wp- methods are used for generating tests from an FSM. We are now well equipped to describe various methods for test generation given an FSM. We begin with a description of the W-method.

6.5. The W-method

The W-method is used for constructing a test set from a given FSM M. The test set so constructed is a finite set of sequences that can be input to a program whose control structure is modeled by M. Alternately, the tests can also be input to a design to test its correctness with respect to some specification.

The implementation modeled by M is also known as the Implementation Under Test and abbreviated as IUT. Note that most software systems cannot be modeled accurately using a finite state machine. However, the global control structure of a software system can be modeled by an FSM. This also implies that the tests generated using the W-method, or any other method based exclusively on a finite state model of an implementation, will likely reveal only certain types of faults. Later in Section 6.8 we will discuss what kinds of faults are revealed by the tests generated using the W-method.

6.5.1. Assumptions

The W-method makes the following assumptions for it to work effectively.

1. M is completely specified, minimal, connected, and deterministic.
2. M starts in a fixed initial state.
3. M can be reset accurately to the initial state. A `null` output is generated during the reset operation.
4. M and IUT have the same input alphabet.

Later in this section we shall examine the impact of violating the above assumptions. Given an FSM $M = (X, Y, Q, q_0, \delta, O)$, the W-method consists of the following sequence of steps.

Step 1 Estimate the maximum number of states in the correct design.

Step 2 Construct the characterization set W for the given machine M.
Step 3 Construct the testing tree for M and from it determine the transition cover set P.
Step 4 Construct set Z.
Step 5 $P.Z$ is the desired test set.

We already know how to construct W for a given FSM. In the remainder of this section we explain each of the remaining three steps mentioned above.

6.5.2. Maximum number of states

We do not have access to the correct design or the correct implementation of the given specification. Hence there is a need to estimate the maximum number of states in the correct design. In the worst case, i.e. when no information about the correct design or implementation is available, one might assume that the maximum number of states is the same as the number of states in M. The impact of an incorrect estimate is discussed after we have examined the W-method.

6.5.3. Computation of the transition cover set

A transition cover set, denoted as P, is defined as follows. Let q_i and q_j, $i \neq j$, be two states in M. P consists of sequences sx such that $\delta(q_0, s) = q_i$ and $\delta(q_i, x) = q_j$. The empty sequence ϵ is also in P. We can construct P using the "testing tree" of M. A testing tree for an FSM is constructed as follows.

1. State q_0, the initial state, is the root of the testing tree.
2. Suppose that the testing tree has been constructed until level k. The $(k+1)^{th}$ level is built as follows.
   - Select a node n at level k. If n appears at any level from 1 through k, then n is a leaf node and is not expanded any further. If n is not a leaf node then we expand it by adding a branch from node n to a new node m if $\delta(n, x) = m$ for $x \in X$. This branch is labeled as x. This step is repeated for all nodes at level k.

The following example illustrates construction of a testing tree for the machine in Example 6.4 whose transition and output functions are shown in Figure 6.9.

EXAMPLE 6.7. The testing tree is initialized with the initial state q_1 as the root node. This is level 1 of the tree. Next we note that $\delta(q_1, a) = q_1$ and $\delta(q_1, b) = q_4$. Hence we create two nodes at the next level and label them as q_1 and q_4. The branches from q_1 to q_1 and q_4 are labeled, respectively, `a` and `b`. As q_1 is the only node at level 1, we now proceed to expand the tree from level 2.

At level 2 we first consider the node labeled q_1. However, another node labeled q_1 already appears at level 1, hence this node becomes a leaf node and is not expanded any further. Next we examine the node labeled q_4. We note that $\delta(q_4, a) = q_3$ and $\delta(q_4, b) = q_4$. We therefore create two new nodes at level 3 and label these as q_4 and q_3 and label the corresponding branches as `b` and `a`, respectively.

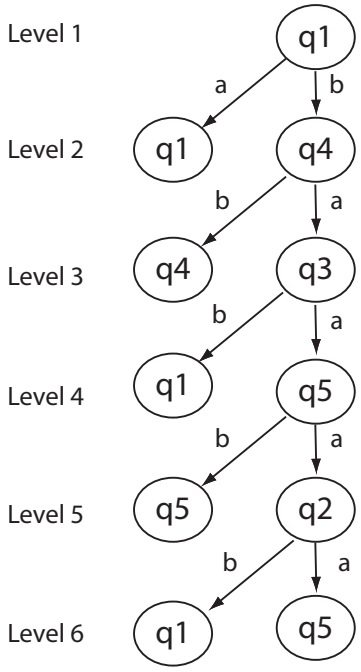


Figure 6.10: Testing tree for the machine with transition and output functions shown in Figure 6.9 on page 184.

We proceed in a similar manner down one level in each step. The method converges when at level 6 we have the nodes labeled q_1 and q_5 both of which appear at some level up the tree. We thus arrive at the testing tree for M as shown in Figure 6.10.

Once a testing tree has been constructed, we obtain the transition cover set P by concatenating labels of all *partial paths* along the tree. A partial path is a path starting from the root of the testing tree and terminating in any node of the tree. Traversing all partial paths in the tree shown in Figure 6.10, we obtain the following transition cover set.

$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$

It is important to understand the function of the elements of P. As the name *transition cover set* implies, exciting an FSM in q_0, the initial state, with an element of P, forces the FSM into some state. After the FSM has been excited with all elements of P, each time starting in the initial state, the FSM has reached every state. Thus exciting an FSM with elements of P ensures that all states are reached, and all transitions have been traversed at least once. For example, when the FSM M of Example 6.4 i is excited by the input sequence $baab$ in state q_1, it traverses the branches (q_1, q_4), (q_4, q_3), (q_3, q_5), and (q_5, q_5), in that order. The empty input sequences, ϵ, does not traverse any branch but is useful in constructing the desired test sequence as is explained next.

6.5.4. Constructing Z

Given the input alphabet X and the characterization set W, it is straightforward to construct Z. Suppose that the number of states estimated to be in the IUT is m and the number of states in the design specification is n, $m \geq n$. Given this information we compute Z as:

$$Z = (X^0.W) \cup (X.W) \cup (X^2.W) \ldots \cup (X^{m-1-n}.W) \cup (X^{m-n}.W)$$

It is easy to see that $Z = X.W$ for $m = n$, i.e. when the number of states in the IUT is the same as that in the specification. For $m < n$ we use $Z = X.W$. Recall from Chapter 1 that $X^0 = \{\epsilon\}, X^1 = X, X^2 = X.X$, and so on, where . denotes string concatenation. For convenience, we shall use the shorthand notation $X[p]$ to denote the following set union

$$\{\epsilon\} \cup X^1 \cup X^2 \ldots \cup X^{p-1} \cup X^p.$$

We can now rewrite Z as $Z = X[m-n].W$, for $m \geq n$ where m is the number of states in the IUT and n the number of states in the design specification.

6.5.5. Deriving a test set

Having constructed P and Z we can easily obtain a test set T as $P.Z$. The next example shows how to construct a test set from the FSM of Example 6.4.

EXAMPLE 6.8. For the sake of simplicity let us assume that the number of states in the correct design or IUT is the same as in the given design shown in Figure 6.9. Thus we have $m = n = 5$. This leads to the following Z:

$$Z = X^0.W = \{a, aa, aaa, baaa\}$$

Catenating P with Z we obtain the desired test set.

$$\begin{aligned} T = P.Z &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}.\{a, aa, aaa, baaa\} \\ &= \{a, aa, aaa, baaa, \\ &\quad aa, aaa, aaaa, abaaa, \\ &\quad ba, baa, baaa, bbaaa, bba, \\ &\quad bba, bbaa, bbaaa, bbbaaa, \\ &\quad baa, baaa, baaaa, babaaa, \\ &\quad baba, babaa, babaaa, babbaaa, \\ &\quad baaa, baaaa, baaaaa, baabaaa, \\ &\quad baaba, baabaa, baabaaa, baabbaaa, \\ &\quad baaaa, baaaaa, baaaaaa, baaabaaa \\ &\quad baaaba, baaabaa, baaabaaa, baaabbaaa \\ &\quad baaaaa, baaaaaa, baaaaaaa, baaaabaaa\} \end{aligned}$$

If we assume that the IUT has one extra state, i.e. $m = 6$, then we obtain Z and the test set $P.Z$ as follows.

$Z = X^0.W \cup X^1.W = \{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa\}$

$T = P.Z = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}.$
$\{a, aa, aaa, baaa, aa, aaa, aaaa, abaaa, ba, baa, baaa, bbaaa\}$

6.5.6. Testing using the W-method

To test the given IUT M_i against its specification M, we do the following for each test input.

1. Find the expected response $M(t)$ to a given test input t. This is done by examining the specification. Alternately, if a tool is available, and the specification is executable, one could determine the expected response automatically.
2. Obtain the response $M_i(t)$ of the IUT, when excited with t in the initial state.
3. If $M(t) = M_i(t)$ then no flaw has been detected so far in the IUT. $M(t) \neq M_i(t)$ implies the possibility of a flaw in the IUT under test, given a correct design.

Notice that a mismatch between the expected and the actual response does not necessarily imply an error in the IUT. However, if we assume that (a) the specification is error free, (b) $M(t)$, $M_i(t)$ have been determined without any error, and (c) the comparison between $M(t)$ and $M_i(t)$ is correct, then indeed $M(t) \neq M_i(t)$ implies an error in the design or the IUT under test.

EXAMPLE 6.9. Let us now apply the W-method to test the design shown in Figure 6.9. We assume that the specification is also given in terms of an FSM which we refer to as the "correct design." Note that this might not happen in practice but we make this assumption for illustration.

We consider two possible implementations under test. The correct design is shown in Figure 6.11(a) and is referred to as M. We denote the two erroneous designs, corresponding to two IUTs under test, as M_1 and M_2, respectively, shown in Figure 6.11(b) and (c).

Notice that M_1 has one transfer error with respect to M. The error occurs in state q_2 where the state transition on input a should be $\delta_1(q_2, a) = q_1$ and not $\delta_1(q_2, a) = q_3$. M_2 has two errors with respect to M. There is a transfer error in state q_2 as mentioned earlier. In addition there is an operation error in state q_5 where the output function on input b should be $O_2(q_5, b) = 1$ and not $O_2(q_5, b) = 0$.

To test M_1 against M, we apply each input t from the set $P.Z$ derived in Example 6.8 and compare $M(t)$ with $M_1(t)$. However, for the purpose of this example, let us first select $t = ba$. Tracing gives us $M(t) = 11$ and $M_1(t) = 11$. Thus IUT M_1 behaves correctly when excited by the input sequence ba. Next we select $t = baaaaaa$ as a test input. Tracing the response of $M(t)$ and $M_1(t)$ when excited by t, we obtain $M(t) = 1101000$ and $M_1(t) = 1101001$. Thus the input sequence $baaaaaa$ has revealed the transfer error in M_1.

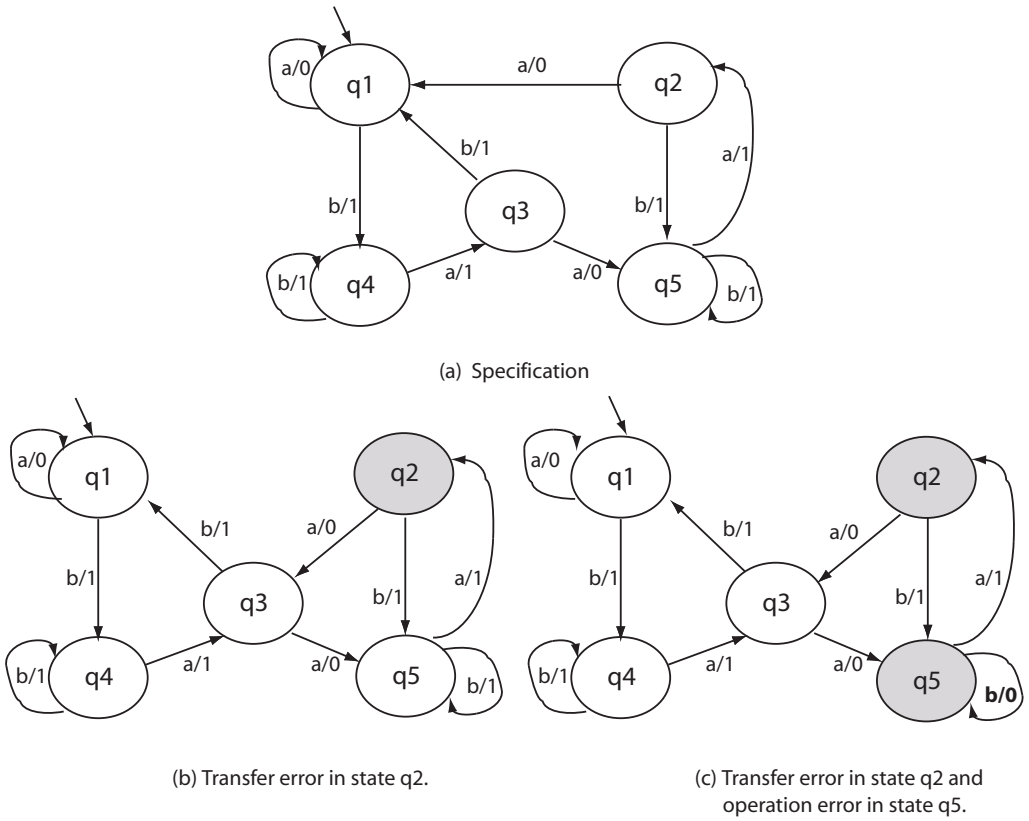


Figure 6.11: The transition and output functions of FSMs of the design under test, denoted as M in the text and copied from Figure 6.9 and two incorrect designs in (b) and (c) denoted as M_1 and M_2 in the text.

Next let us test M_2 with respect to specification M. We select the input sequence $t = baaba$ for this test. Tracing the two designs we obtain $M(t) = 11011$ whereas $M_2(t) = 11001$. As the two traces are different, input sequence $baaba$ reveals the operation error. We have already shown that $x = baaaaaa$ reveals the transfer error. Thus the two input sequences $baaba$ and $baaaaaa$ in $P.Z$ reveal the errors in the two IUTs under test.

Note that for the sake of brevity we have used only two test inputs. However, in practice, one needs to apply test inputs to the IUT in some sequence until the IUT fails or the IUT has performed successfully on all tests.

EXAMPLE 6.10. Suppose we estimate that the IUT corresponding to the machine in Figure 6.9 has one extra state, i.e. $m = 6$. Let us also assume that the IUT, denoted as M_1, is as shown in Figure 6.12(a) and indeed has six states. We test M_1 against $t = baaba$ and obtain $M_1(t) = 11001$. The correct output obtained by tracing M against t is 11011. As $M(t) \neq M_1(t)$, test input t has revealed the "extra state" error in S_1.

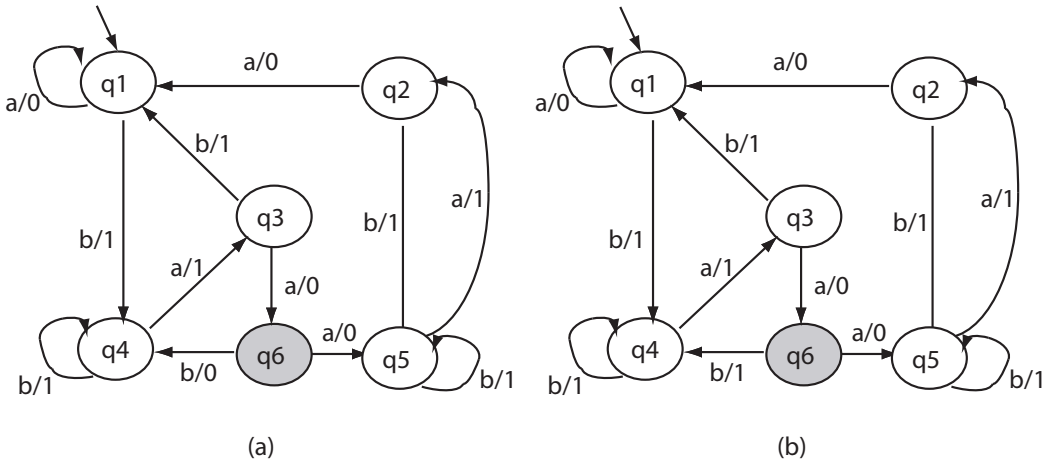


Figure 6.12: Two implementations each of the design in Figure 6.9 each having an extra state. Note that the output functions of the extra state q_6 are different in (a) and (b)

However, test $t = baaba$ does not reveal the extra state error in machine M_2 as $M_2(t) = 11011 = M(t)$. Consider now test $t = baaa$. We get $M(t) = 1101$ and $M_2(t) = 1100$. Thus the input $baaa$ reveals the extra state error in machine M_2.

6.5.7. The error detection process

Let us now examine carefully how the test sequences generated by the W-method detect operartion and transfer errors. Recall that the test set generated in the W-method is the set $P.W$, given that the number of states in the IUT is the same as that in the specification. Thus each test case t is of the form $r.s$ where r belongs to the transition cover set P and s to the characterization set W. We consider the application of t to the IUT as a two phase process. In the first phase input r moves the IUT from its initial state q_0 to state q_i. In the second phase, the remaining input s moves the IUT to its final state q_j or $q_{j'}$. These two phases are shown in Figure 6.13.

When the IUT is started in its initial state, say q_0, and excited with test t, it consumes the string r and arrives at some state q_i as shown in Figure 6.13. The output generated so far by the IUT is u where $u = O(q_0, r)$. Continuing further in state q_i, the IUT consumes symbols in string s and arrives at state q_j. The output generated by the IUT while moving from state q_i to q_j is v where $v = O(q_i, s)$. If there is any operation error along the transitions traversed between states q_0 and q_i, then the output string u would be different from the output generated by the specification machine M when excited with r in its initial state. If there is any operation error along the transitions while moving from q_i to q_j then the output wv would be different from that generated by M.

The detection of a transfer error is more involved. Suppose that there is a transfer error in state q_i and $s = as'$. Thus instead of moving to state q_k, where $q_k = \delta(q_i, a)$, the IUT moves to state $q_{k'}$ where $q_{k'} = \delta(q_i, a)$. Eventually the IUT ends up in state $q_{j'}$ where $q_{j'} = \delta(q_{k'}, s')$. Our

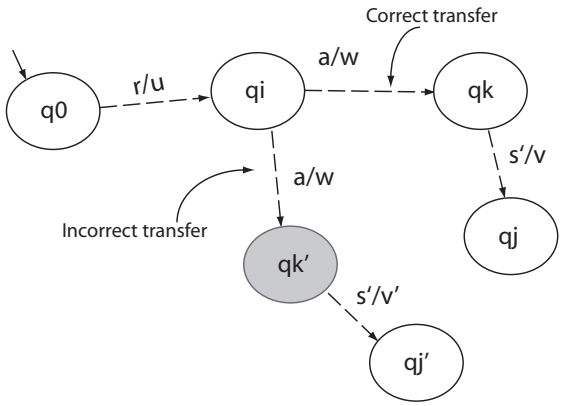


Figure 6.13: Detecting errors using tests generated by the W-method.

assumption is that s is a distinguishing sequence in W. If s is the distinguishing sequence for states q_i and $q_{j'}$ then $wv' \neq wv$. If s is not a distinguishing sequence for q_i and $q_{j'}$ then there must exist some other input as'' in W such that $wv'' \neq wv$ where $v'' = O(q_{k'}, s'')$.

6.6. The UIO-sequence method

The W-method uses the characterization set W of distinguishing sequences as the basis to generate a test set for an IUT. The tests so generated are effective in revealing operation and transfer faults. However, the number of tests generated using the W-method is usually large. Several alternative methods have been proposed that generate fewer test cases. In addition these methods are either as effective or nearly as effective as the W-method in their fault detection capability. We now introduce two such methods. A method for test generation based on *unique input/output sequences* is presented in this section. Another method known as the partial W method, or simply the Wp method, is presented in Section 6.7.

6.6.1. Assumptions

The UIO method generates tests from an FSM representation of the design. All the assumptions stated in Section 6.5.1 apply to the FSM that is input to the UIO test generation procedure. In addition, it is assumed that the IUT has the same number of states as the FSM that specifies the corresponding design. Thus any errors in the IUT are transfer and operations errors only as shown in Figure 6.8.

6.6.2. UIO sequences

A unique input/output sequence, abbreviated as UIO sequence, is a sequence of input and output pairs that distinguishes a state of an FSM from the remaining states. Consider FSM $M = (X, Y, Q, q_0, \delta, O)$. A UIO sequence of length k for some state $s \in Q$, is denoted as $UIO(s)$ and looks like the following sequence

$UIO(s) = i_1/o_1.i_2/o_2 \ldots i_{(k-1)}/o_{(k-1)}.i_k/o_k.$

In the sequence given above, each pair a/b consists of an input symbol a that belongs to the input alphabet X and an output symbol b that belongs to the output alphabet Y. As before, the dot symbol (.) denotes string concatenation. We refer to the *input* portion of a $UIO(s)$ as $in(UIO(s))$ and to its *output* portion as $out(UIO(s))$. Using this notation for the input and output portions of $UIO(s)$, we can rewrite $UIO(s)$ as

$in(UIO(s)) = i_1.i_2.\ldots.i_{(k-1)}.i_k$, and

$out(UIO(s)) = o_1.o_2.\ldots.o_{(k-1)}.o_k$.

When the sequence of input symbols that belong to $in(UIO(s))$ is applied to M in state s, the machine moves to some state t and generates the output sequence $out(UIO(s))$. This can be stated more precisely as follows

$\delta(s, in(UIO(s)) = out(UIO(s))$

A formal definition of unique input/output sequences follows.

Given an FSM $M = (X, Y, Q, q_0, \delta, O)$, the unique input/output sequence for state $s \in Q$, denoted as $UIO(s)$, is a sequence of one or more edge labels such that the following condition holds.

$\delta(s, in(UIO(s)) \neq \delta(t, in(UIO(s))$, for all $t \in Q, t \neq s$.

The next example offers UIO sequences for a few FSMs. The example also shows that there may not exist any UIO sequence for one or more states in an FSM.

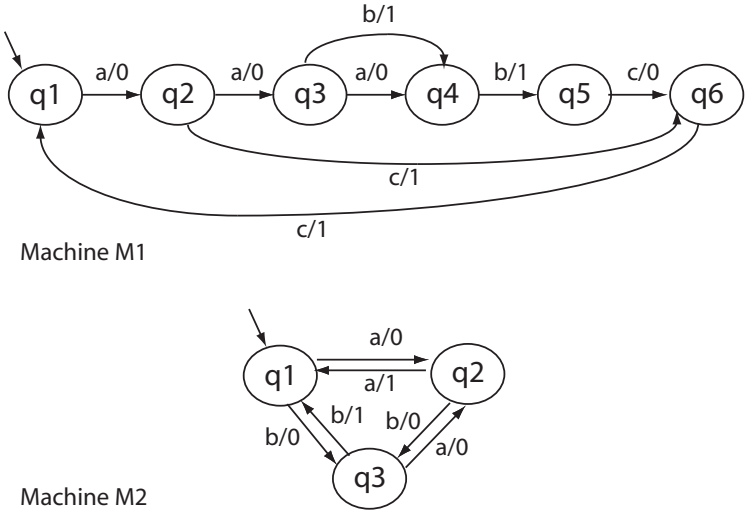


Figure 6.14: Two FSMs used in Example 6.11. There is a UIO sequence for each state in Machine M1. There is no UIO sequence for state q_1 in Machine M2.

EXAMPLE 6.11. Consider machine M1 shown in Figure 6.14. The UIO sequence for each of the six states in M1 is given below.

State (s)	UIO (s)
q_1	a/0.c/1
q_2	c/1.c/1
q_3	b/1.b/1
q_4	b/1.c/0
q_5	c/0
q_6	c/1.a/0

Using the definition of UIO sequences, it is easy to check whether or not a $UIO(s)$ for some state s is indeed a unique input/output sequence. However, before we perform such a check, we assume that the machine generates an empty string as the output if there does not exist an outgoing edge from a state on some input. For example, in Machine M1 there is no outgoing edge from state q_2 for input c. Thus M1 generates an empty string when it encounters a c in state q_2. This behavior is explained in Section 6.6.3.

Let us now perform two sample checks. From the table above we have $UIO(q_1) = a/0.c/1$. Thus $in(UIO(q_1) = a.c$. Applying the sequence $a.c$ to state q_2 produces the output sequence 0 which is different from 01 generated when $a.c$ is applied to M1 in state q_1. Similarly, applying $a.c$ to Machine M1 in state q_5 generates the output pattern 0 which is, as before, different from that generated by state/1. You may now perform all other checks to ensure that indeed the UIOs given above are correct.

EXAMPLE 6.12. Now consider Machine M2 in Figure 6.14. The UIO sequences for all states, except state q_1, are listed below. Notice that there does not exist any UIO sequence for state q_1.

State (s)	UIO (s)
q_1	None
q_2	a/1
q_3	b/1

6.6.3. Core and non-core behavior

An FSM must satisfy certain constraints before it is used for constructing UIO sequences. First there is the "connected assumption" which implies that every state in the FSM is reachable from its initial state. The second assumption is that the machine can be applied a *reset* input in any state that brings it to its start state. As has been explained earlier, a `null` output is generated upon the application of a reset input.

The third assumption is known as the *completeness* assumption. According to this assumption, an FSM remains in its current state upon the receipt of any input for which the state transition function δ is not specified. Such an input is known as a *non-core* input. In this situation the machine generates a `null` output. The completeness assumption implies that each state contains a self loop that generates a `null` output upon the receipt of an unspecified in-

put. The fourth assumption is the FSM must be minimal. A machine that depicts only the core behavior of an FSM is referred to as its core-FSM. The following example illustrates the core behavior.

EXAMPLE 6.13. Consider the FSM shown in Figure 6.6.3(a, similar to the FSM shown earlier in Figure 6.9 and contains self-loops in states q_1, q_4, and q_5. Note that states q_1, q_4, and q_5 have no transitions corresponding to inputs a, b, and b, respectively. Thus this machine does no satisfy the completeness assumption.

As shown in Figure 6.15(b), additional edges are added to this machine in states q_1, q_4, and q_5. These edges represent transitions that generate `null` output. Such transitions are also known as errorneous transitions. Figure 6.15(a) shows the core behavior corresponding to the machine in Figure 6.15(b).

While determining the UIO sequences for a machine, only the core behavior is considered. The UIO sequences for all states of the machine shown in Figure 6.15(a) are given below.

State(s)	UIO (s)
q_1	b/1.a/1/.b/1.b/1
q_2	a/0.b/1
q_3	b/1.b/1
q_4	a/1.b/1.b/1
q_5	a/1.a/0.b/1

Note that the core behavior exhibited in Figure 6.15 is different from that in the original design in Figure 6.9. Self loops that generate a `null` output, are generally not shown in a state diagram that depicts the core behavior. The impact of removing self-loops, that generate a non-null output, on the fault detection capability of the UIO method will be discussed in Section 6.6.8. In Figure 6.15, the set of core edges is $\{(q_1, q_4), (q_2, q_1), (q_2, q_5), (q_3, q_1), (q_3, q_5), (q_4, q_3), (q_5, q_2), (q_1, q_1), (q_2, q_2), (q_3, q_3), (q_4, q_4), (q_5, q_5)\}$.

During a test one wishes to determine if the IUT behaves in accordance with its specification. An IUT is said to conform *strongly* to its specification if it generates the same output as its specification for all inputs. An IUT is said to conform *weakly* to its specification if it generates the same output as that generated by the corresponding core-FSM.

EXAMPLE 6.14. Consider an IUT that is to be tested against its specification M as in Figure 6.9. Suppose that the IUT is started in state q_1 and the one symbol sequence a is input to it and the IUT outputs a `null` sequence, i.e. the IUT does not generate any output. In this case the IUT does not conform strongly to M. However, on input a, this is exactly the behavior of the core-FSM. Thus if the IUT behaves exactly as the core-FSM shown in Figure 6.15, then it conforms weakly to M.

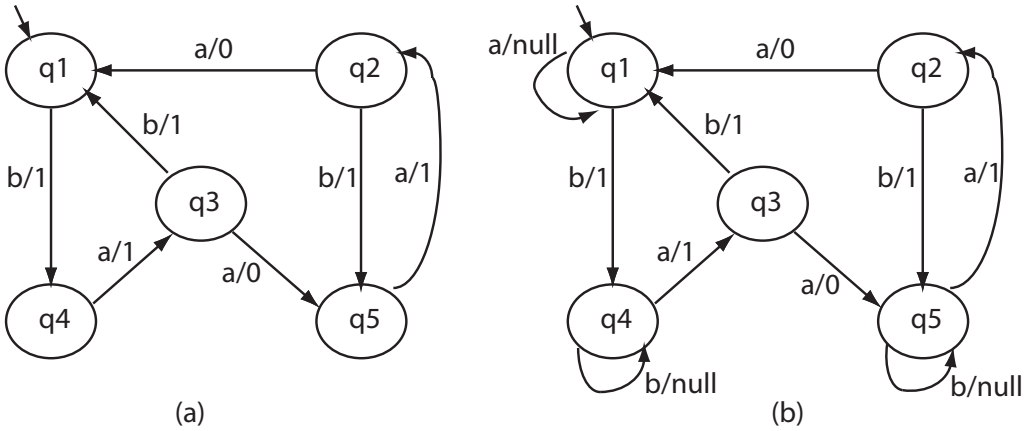


Figure 6.15: (a) An incomplete FSM. (b) Addition of `null` transitions to satisfy the completeness assumption.

6.6.4. Generation of UIO sequences

The UIO sequences are essential to the UIO-sequence method for generating tests from a given FSM. In this section we present an algorithm for generating the UIO sequences. As mentioned earlier, a UIO sequence might not exist for one or more states in an FSM. In such a situation, a signature is used instead. An algorithm for generating UIO sequences for all states of a given FSM follows.

`Procedure for generating UIO sequences`

Input: (a) An FSM $M = (X, Y, Q, q_0, \delta, O)$ where $\mid Q \mid = n$.

(b) State $s \in Q$.

Output: $UIO[s]$ contains a unique input/output sequence for state s; $UIO(s)$ is empty if no such sequence exists.

Procedure: `gen-UIO(s)`

/* $Set(l)$ denotes the set of all edges with label *l*.

$label(e)$ denotes the label of edge e.

A label is of the kind a/b where $a \in X$ and $b \in Y$.

$head(e)$ and $tail(e)$ denote, respectively, the head and tail states of edge e. */

Step 1 For each distinct edge label el in the FSM, compute $Set(el)$.

Step 2 Compute the set *Oedges(s)* of all outgoing edges for state s. Let $NE = \mid Oedges \mid$.

Step 3 For $1 \leq i \leq NE$ and each edge $e_i \in Oedges$, compute $Oled[i]$, $Opattern[i]$, and $Oend[i]$ as follows:

3.1 $Oled[i] = Set(label(e_i)) - \{e_i\}$.

3.2 $Opattern[i] = label(i)$.

3.3 $Oend[i] = tail(i)$.

Step 4 Apply algorithm *gen-1-uio(s)* to determine if $UIO[s]$ consists of only one label.

Step 5 If simple UIO sequence found then return $UIO[s]$ and terminate this algorithm, otherwise proceed to the next step.

Step 6 Apply the *gen-long-uio(s)* procedure to generate longer $UIO[s]$.

Step 7 If longer UIO sequence found then return $UIO[s]$ and terminate this algorithm, else return with an empty $UIO[s]$

`End of Procedure gen-uio.`

`Procedure gen-1-uio`

Input: State $s \in Q$.

Output: $UIO[s]$ of length 1 if it exists, an empty string otherwise.

Procedure: `gen-1-UIO(State s):`

Step 1 If $Oled[i] = \varnothing$ for $1 \leq i \leq NE$ then return $UIO[s] = label(e)$ otherwise return $UIO[s] =$ "".

`End of Procedure gen-1-uio`

`Procedure for generating longer UIO sequences`

Input: (a) $Oedges$, $Opattern$, $Oend$, and $Oled$ as computed in *gen-1-uio*.

(b) State $s \in Q$.

Output: $UIO[s]$ contains a unique input/output sequence for state s. $UIO(s)$ is empty if no such sequence exists.

Procedure: `gen-long-uio (State s)`

Step 1 Let L denote the length of the UIO sequence being examined. Set $L = 1$. Let Ond denote the number of outgoing edges from some state under consideration.

Step 2 Repeat steps below while $L < 2n^2$ or the procedure is terminated prematurely.

2.1 Set $L = L + 1$ and $k = 0$. Counter k denotes the number of distinct patterns examined as candidates for $UIO(s)$. The following steps attempt to find a $UIO[s]$ of size L.

2.2 Repeat steps below for $i = 1$ to NE. Index i is used to select an element of $Oend$ to be examined next. Note that $Oend[i]$ is the tail state of the pattern in $Opattern[i]$.

2.2.1 Let $Tedges(t)$ denote the set of outgoing edges from state t. We compute $Tedges(Oend[i])$.

2.2.2 For each edge $te \in Tedges$ execute *gen-L-UIO(te)* until either all edges in $Tedges$ have been considered or a $UIO[s]$ is found.

2.3 Prepare for the next iteration. Set $NE = k$ which will serve as the new maximum value of the next iteration over index i. For $1 \leq j \leq k$, set $Opattern[j] = Pattern[j]$, $Oend[j] = End[j]$, and $Oled[j] = Led[j]$. If the loop termination condition is not satisfied then go back to Step 2.1 and search for UIO of the next higher length, else return to *gen-uio* indicating a failure to find any UIO for state s.

`End of Procedure gen-long-uio`

`Procedure for generating UIO sequences of Length` $L > 1$.

Input: Edge te from procedure *gen-long-UIO* .

Output: $UIO[s]$ contains a unique input/output sequence of length L for state s of length$L$, $UIO(s)$ is empty if no such sequence exists.

Procedure: `gen-L-uio ( edge te)`

Step 1 $k = k + 1$, $Pattern[k] = Opattarn[i].label(te)$. This could be a possible UIO.

Step 2 $End[k] = tail(te)$ and $Led[k] = \varnothing$.

Step 3 For each pair $oe \in Oled[i]$, where $h = head(oe)$ and $t = tail(oe)$, repeat the next step.

3.1 Execute the next step for each edge $o \in OutEdges(t)$.

3.1.1 If $label(o) = label(te)$ then
$Led[k] = Led[k] \bigcup \{(head(oe), tail(o))\}$.

Step 4 If $Led[k] = \varnothing$ then UIO of length L has been found. Set $UIO[s] = Pattern[k]$ and terminate this and all procedures up until the main procedure for finding UIO for a state. If $Led[k] \neq \varnothing$ then no UIO of length L has been found corresponding to edge te. Return to the caller to make further attempts.

`End of Procedure gen-L-uio`

Explanation of *gen-uio*

The idea underlying the generation of a UIO sequence for state s can be explained as follows. Let M be the FSM under consideration and s a state in M for which a UIO is to be found. Let $E(s) = e_1 e_2 \ldots e_k$ be a sequence of edges in M such that $s = head(e_1), tail(e_i) = head(e_{i+1}), 1 \leq i < k$.

For $E(s)$, we define a string of edge labels as $label(E(s)) = l_1.l_2.\ldots.l_{k-1}.l_k$, where $l_i = label(e_i), 1 \leq i \leq k$, is the label of edge e_i. For a given integer $l > 0$, we find if there is a sequence $E(s)$ of edges starting from s such that $labels(E(s)) \neq labels(E(t)), s \neq t$ for all states t in M. If there is such an $E(s)$ then $label(E(s))$ is a UIO for state s. The uniqueness check is performed on $E(s)$ starting with $l = 1$ and continuing for $l = 2, 3, ...$ until a UIO is found or $l = 2n^2$, where n is the number of states in M. Note that there can be multiple UIOs for a given state though the algorithm generates only one UIO, if it exists.

We explain the *gen-uio* algorithm with reference to Figure 6.16. *gen-uio* for state s begins with the computation of some sets used in the subsequent steps. First, the algorithm computes $Set(el)$ for all distinct labels el in the FSM. $Set(el)$ is the set of all edges that have el as their label.

Next, the algorithm computes the set $Oedges(s)$ which is the set outgoing edges from state q_s. For each edge e in $Oedges(s)$, the set $Oled[e]$ is computed. $Oled[e]$ contains all edges in $Set(label(e))$ except for the edge e. The tail state of reach edge in $Oedges$ is saved in $Oend[e]$. The use of $Oled$ will become clear later in Example 6.19.

EXAMPLE 6.15. To find $UIO(q_1)$ for machine M1 in Figure 6.14, Set, $Oedges$ and $Oled$ are computed as follows.

Distinct labels in M1= $\{a/0, b/1, c/0, c/1\}$
$Set(a/0) = \{(1,2),(2,3),(3,4)_{a/0}\}$
$Set(b/1) = \{(3,4),(4,5)\}$
$Set(c/0) = \{(5,6)\}$
$Set(c/1) = \{(2,6),(6,1)\}$

$Oedges(q_)) = \{(1,2)\}$, $NE = 1$
$Oled[1] = \{(2,3),(3,4)\}$
$Oend[1] = q_2$

Next *gen-1-uio* is called in an attempt to generate a UIO of length 1. For each edge $e_i \in Oedges$, *gen-1-uio* initializes $Opattern[i]$ to $label(e_i)$ and $Oend[i]$ to $tail(e_i)$. $Opattern$ and $Oend$ are used subsequently by *gen-uio* in case a UIO of length 1 is not found. *gen-1-uio* now checks if the label of any of the outgoing edges from s is unique. This is done by checking if $Oled[i]$ is empty for any $e_i \in Oedges(s)$. Upon return from *gen-1-uio*, *gen-uio* terminates if a UIO of length 1 was found, otherwise it invokes *gen-long-uio* in an attempt to find a UIO of length greater than 1. *gen-longer-uio* attempts to find UIO of length $L > 1$ until it finds one, or $L = 2n^2$.

EXAMPLE 6.16. As an example of a UIO of length 1, consider state q_5 in Machine M1 in Figure 6.14. The set of outgoing edges for state q_5 is {(5, 6)}. The label of {(5, 6)} is $c/0$. Thus from *gen-uio* and *gen-1-uio* we get $Oedges(q_5) = \{(5,6)\}$, $Oled[1] = \varnothing$, $Oen[1] = 6$, and $Opattern[1] = c/0$. As $Oled[1] = \varnothing$, state q_5 has a UIO of length 1 which is $Opattern[1]$, i.e. $c/0$.

EXAMPLE 6.17. As an example of a state for which a UIO of length 1 does not exist, consider state q_3 in Machine $M2$ shown in Figure 6.14. For this state we get the following sets:

$Oedges[q_3] = \{(3,4)_{a/0},(3,4)_{b/1}\}$, $NE = 2$
$Oled[1] = \{(1,2),(2,3)\}$, $Oled[2] = \{(4,5)\}$

$Opattern[1] = a/0$, $Opattern[2] = b/1$
$Oend[1] = q_4$, $Oend[2] = q_4$.

As no element of $Oled$ is empty, *gen-1-uio* concludes that state q_3 has no UIO of length 1 and returns to *gen-uio*.

In the event *gen-1-uio* fails, procedure *gen-long-uio* is invoked. The task of *gen-long-uio* is to check if a $UIO(s)$ of length two or more exist. To do so, it collaborates with its descendent *gen-L-uio*. An incremental approach is used where UIOs of increasing length, starting at two, are examined until either a UIO is found or one is unlikely to be found, i.e. a maximum length pattern has been examined.

To check if there is UIO of length L, *gen-long-uio* computes the set $Tedges(t)$ for each edge $e \in Oedges(s)$, where $t = tail(e)$. For $L = 2$, $Tedges(t)$ will be a set of edges outgoing from the tail state t of one of the edges of state s. However, in general, $Tedges(t)$ will be the set of edges going out of a tail state of some edge outgoing from state s', where s' is a successor of s. After initializing $Tedges$, *gen-L-uio* is invoked iteratively for each element of $Tedges(t)$. The task of *gen-L-uio* is to find whether or not there exists a UIO of length L.

EXAMPLE 6.18. There is no $UIO(q_1)$ of length one. Hence *gen-long-uio* is invoked. It begins by attempting to find a UIO of length two, i.e. $L = 2$. From *gen-uio* we have $Oedges(q_1) = \{(1, 2)\}$. For the lone edge in $Oedges$, we get $t = tail((1, 2) = q_2$. Hence $Tedges(q_2)$, which is the set of edges out of state q_2, is computed as follows:

$Tedges(q_2) = \{(2, 3), (2, 6)\}$

gen-L-uio is now invoked first with edge (2, 3) as input to determine if there is a UIO of length two. If this attempt fails, then *gen-L-uio* is invoked with edge (2, 6) as input.

To determine if there exists a UIO of length L corresponding to an edge in $Tedges$, *gen-L-uio* initializes $Pattern[k]$ by catenating the label of edge te to $Opattern[k]$. Recall that $Opattern[k]$ is of length $(L - 1)$ and has been rejected as a possible $UIO(s)$. $Pattern[k]$ is of length L and is a candidate for $UIO(s)$. Next, the tail of te is saved in $End[k]$. Here k serves as a running counter of the number of patterns of length L examined.

Next, for each element of $Oled[i]$, *gen-L-uio* attempts to determine if indeed $Pattern[k]$ is a $UIO(s)$. To understand this procedure, suppose that oe is an edge in $Oled[i]$. Recall that edge oe has the same label as the edge e_i and that e_i is not in $Oled[i]$. Let t be the tail of state of oe and $OutEdges(t)$ the set of all outgoing edges from state t. Then $Led[k]$ is the set of all pairs $(head(oe), tail(o))$ such that $label(te) = label(o)$ for all $o \in Outedges(t)$.

Note that an element of $Led[k]$ is not an edge in the FSM. If $Led[k]$ is empty after having completed the nested loops in Step 3, then $Pattern[k]$ is indeed a $UIO(s)$. In this case the *gen-uio* procedure is terminated and $Pattern[te]$ is the desired UIO. If $Led[k]$ is not empty then *gen-L-uio* returns to the caller for further attempts at finding $UIO(s)$.

EXAMPLE 6.19. Continuing Example 6.18, suppose that *gen-L-uio* is invoked with $i = 1$ and $te = (2, 3)$. The goal of *gen-L-uio* is to determine if there is a path of length L starting from state $head(te)$ that has its label same as $Opattern[i].label(te)$.

$Pattern[i]$ is set to $a/0.a/0$ because $Opattern[i] = a/0$ and $label((2, 3)) = a/0$. $End[i]$ is set to $tail(2, 3)$ which is q_3. The outer loop in Step 3 examines each element oe of $Oled[1]$. Let $oe = (2, 3)$ for which we get $h = q_2$ and $t = q_3$. The set $OutEdges(q_3) = \{(3, 4)_{a/0}, (3, 4)_{b/1})\}$. Step 3.1 now iterates over the two elements of $OutEedges(q_3)$ and updates $Led[i]$. At the end of this loop we have $Led[i] = \{(2, 4)\}$ because $label((3, 4)_{a/0}) = label((2, 3))$.

Continuing with the iteration over $Oled[1]$, oe is set to $(3,4)_{a/0}$ for which $h = q_2$ and $t = q_4$. The set $OutEdges(q_4) = \{(4,5)\}$. Iterating over $OutEdges(q_4)$ does not alter $Led[i]$ because $label((4,5)) \neq label((3,4)_{a/0})$.

The outer loop in Step 3 is now terminated. In the next step, i.e. Step 4, $Pattern[i]$ is rejected and *gen-L-uio* returns to *gen-long-uio*.

Upon return from *gen-L-uio*, *gen-long-uio* once again invokes it with $i = 1$ and $te = (2,6)$. $Led[2]$ is determined during this call. To do so, $Pattern[2]$ is initialized to $a/0.c/1$ because $Opattern[i] = a/0$ and $label(2,6) = c/1$, and $End[2]$ to q_6.

Once again the procedure iterates over elements oe of $Oled$. For $oe = (2,3)$, we have as before, $OutEdges(q_3) = \{(3,4)_{a/0}, (3,4)_{b/1})\}$. Step 3.1 now iterates over the two elements of $OutEdges(q_3)$. In Step 1, none of the checks is successful as $label((2,6))$ is not equal to $label((3,4)_{a/0})$, or to $label((4,5))$. The outer loop in Step 3 is now terminated. In the next step, i.e. Step 4, $Pattern[2]$ is accepted as $UIO(q_1)$.

It important to note that $Led[i]$ does not contain edges. Instead, it contains one or more pairs of states, (s_1, s_2) such that a path in the FSM from state s_1 to state s_2 has the same label as $Pattern[i]$. Thus, at the end of the loop in Step 3 of *gen-L-uio*, an empty $Led[i]$ implies that there is no path of length L from $head(te)$ to $tail(tail(te))$ with a label same as $Pattern[i]$.

A call to *gen-L-uio* either terminates the *gen-uio* algorithm abruptly indicating that $UIO(s)$ has been found, or returns normally indicating that $UIO(s)$ is not found, and any remaining iterations should now be carried out. Upon a normal return from *gen-L-uio*, the execution of *gen-long-uio* resumes at Step 2.3. In this step, the existing $Pattern$, Led, and End data is transferred to $Opattern$, $Oend$ and $Oled$, respectively. This is done in preparation for the next iteration to determine if there exists a UIO of length $(L+1)$. In case a higher length sequence remains to be examined, the execution resumes from Step 2.2, else the *gen-long-uio* terminates without having determined a $UIO(s)$.

EXAMPLE 6.20. Consider Machine M2 in Figure 6.14. We invoke *gen-uio(q_1)* to find a UIO for state q_1. As required in Steps 1 and 2, we compute Set for each distinct label and the set of outgoing edges $Oedges(q_1)$.

Distinct labels in M2= $\{a/0, a/1, b/0, b/1\}$

$Set(a/0) = \{(1,2),(3,2)\}$
$Set(a/1) = \{(2,1)\}$
$Set(b/0) = \{(1,3),(2,3)\}$
$Set(b/1) = \{(3,1)\}$

$Oedges(q_1) = \{(1,2),(1,3)\}$
$NE = 2$

Next, as directed in Step 3, we compute $Oled$ and $Oend$ for each edge in $Oedges(q_1)$. The edges in $Oedges$ are numbered seqeuntially so that edge (1, 2) is numebred 1 and edge (1, 3) is numbered 2.

$Oled[1] = \{(3,2)\}$

$Oled[2] = \{(2,3)\}$

$Oend[1] = q_2$
$Oend[2] = q_3$

$Opattern[1] = a/0$
$Opattern[2] = b/0$

gen-1-uio(q_1) is now invoked. As $Oled[1]$ and $Oled[2]$ are non-empty, *gen-1-uio* fails to find a UIO of length one and returns. We now move to Step 6 where procedure *gen-long-uio* is invoked. It begins by attempting to check if a UIO of length two exists. Towards this goal, the set $Tedges$ is computed for each edge in $Oedges$ at Step 2.1 is computed. First, for $i = 1$, we have $Oend[1] = (3,2)$ whose tail is state q_2. Thus we obtain $Tedges(q_2) = \{((2,3),(2,1)\}$.

The loop for iterating over the elements of $Tedges$ begins at Step 2.2.2. Let $te = (2,3)$. *gen-L-uio* is invoked with te as input and the value of index i as 1. Inside *gen-L-uio*, k is incremented to 1 indicating that the first pattern of length two is to be examined. $Pattern[1]$ is set to $a/0.b/0$, $End[1]$ to 3, and $Led[1]$ initialized to the empty set.

The loop for iterating over the two elements of $Oled$ begins at Step 3. Let $oe = (3,2)$ for which $h = 3$, $t = 2$, and $OutEdges(2) = \{(2,1),(2,3)\}$. We now iterate over the elements of $Oedges$ as indicated in Step 3.1. Let $o = (2,1)$. As labels of (2, 1) and (2, 3) do not match, $Led[1]$ remains unchanged. Next, $o = (2,3)$. This time we compare labels of o and te, which are the same edges. Hence the two labels are the same. As directed in Step 3.1.1, we set $Led[1] = (head(oe), tail(o)) = (3,3)$. This terminates the iteration over $Outedges$. This also terminates the iteration over $Oled$ as it contains only one element.

In Step 4, we find that $Led[1]$ is not empty and therefore reject $Pattern[1]$ as a UIO for state q_1. Notice that $Led[1]$ contains a pair (3, 3) which implies that there is a path from state q_3 to q_3 with the label identical to that in $Pattern[1]$. A quick look at the state diagram of M2 in Figure 6.14 reveals that indeed the path $q_3 \rightarrow q_2 \rightarrow q_3$ has the label $a/0.b/0$ which is the same as in $Pattern[1]$.

Contol now returns to Step 2.2.2 in *gen-long-uio*. The next iteration over $Tedges(q_2)$ is initiated with $te = (2,1)$. *gen-L-uio* is invoked once again but this time with index $i = 1$ and $te = (2,1)$. The next pattern of length two to be examined is $Pattern[2] = a/0.a/1$. We now have $End[2] = 1$ and $Led[2] = \varnothing$. Iteration over elements of $Oled[1]$ begins. Once again let $oe = (3,2)$ for which $h = 3$, $t = 2$, and $OutEdges(2) = \{(2,1),(2,3)\}$. Iterating over the two elements of $OutEdges$ we find that that the label of te matches that of edge (2, 1) and not of edge (2, 3). Hence we get $Led[2] = (head(oe), tail(2,1)) = (3,1)$ implying that $Pattern[2]$ is the same as the label of the path from state q_3 to state q_1. The loop over $Oled$ also terminates and control returns to *gen-long-uio*.

In *gen-long-uio* the iteration over $Tedges$ now terminates as we have examined both edges in $Tedges$. This also completes the iteration for $i = 1$ which corresponds to the first outgoing edge of state q_1. We now repeat Step 2.2.2 for $i = 2$. In this iteration the second outgoing edge of state q_1, i.e. edge (1, 3), is considered. Without going into the fine details of each step, we leave it to you to verify that at the end of the iteration for $i = 2$, we get the following.

$Pattern[3] = b/0.a/0$

$Pattern[4] = b/0.b/1$
$End[3] = 2$
$End[4] = 1$
$Led[3] = (2, 2)$
$Led[4] = (2, 1)$

This completes the iteration set up in Step 2.2. We now move to Step 3. This step leads to the following values of $Opattern$, $Oled$, and $Oend$.

$Opattern[1] = Pattern[1] = a/0.b/0$
$Opattern[2] = Pattern[2] = a/0.a/1$
$Opattern[3] = Pattern[3] = b/0.a/0$

$Oend[1] = End[1] = 3$
$Oend[2] = End[2] = 1$
$Oend[3] = End[3] = 2$
$Oend[4] = End[4] = 1$

$Oled[1] = Led[1] = (3, 3)$
$Oled[2] = Led[2] = (3, 1)$ $Oled[3] = Led[3] = (2, 2)$ $Oled[4] = Led[4] = (2, 1)$

The while-loop is now continued with the new values of $Opattern$, $Oend$ and $Oled$. During this iteration, $L = 3$ and hence patterns of length three will be examined as candidates for $UIO(q_1)$. The patterns to be examined next will have at least one of the patterns in $Opattern$ as their prefix. For example, one of the patterns examined for $i = 1$ is $a/0/.b/0.a/0$.

The *gen-L-uio* procedure is invoked first with $e = (1, 2)$ and $te = (2, 3)$ in Step 2.1.2. It begins with $Pattern[(2, 3) = a/0.b/0$ and $End[(2, 3)] = 3$.

EXAMPLE 6.21. This example illustrates the workings of Procedure *gen-UIO(s)* by tracing through the algorithm for Machine M1 in Figure 6.14. The complete set of UIO sequences for this machine is given in Example 6.11 on page 198. In this example we sequence through *gen-uio(s)* for $s = q_1$. The trace follows.

---

gen-uio *Input*: State q_1

Step 1 Find the set of edges for each distinct label in M1. There are four distinct labels in M1, namely $a/0, b/1, c/0$, and $c/1$. The set of edges for each of these four labels is given below.
$Set(a/0) = \{(1, 2), (2, 3), (3, 4)\}$
$Set(b/1) = \{(3, 4), (4, 5)\}$
$Set(c/0) = \{(5, 6)\}$
$Set(c/1) = \{(2, 6), (6, 1)\}$

Step 2 It is easily seen from the state transition function of M1 that the set of outgoing edges from state q_1 is {(1,2)}. Thus we obtain $Oedges(q_1) = \{(1, 2)\}$ and $NE = 1$.

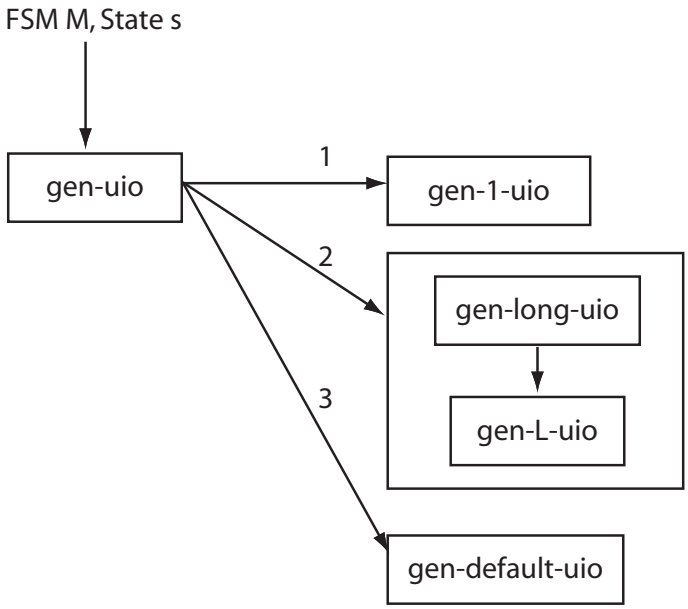


Figure 6.16: Flow of control across various procedures in *gen-uio*.

Step 3 For each edge in $Oedges$, we compute $Oled$, $Opattern$, and $Oend$.
$Oled[1] = \{(1,2),(2,3),(3,4)_{a/0}\} - \{(1,2)\} = \{(2,3),(3,4)_{a/0}\}$.
$Opattern[1] = label((1,2)) = a/0$, and
$Oend[1] = tail((1,2)) = 2$.

Step 4 Procedure *gen-1-UIO* is invoked with state q_1 as input. In this step an attempt is made to determine if there exists a UIO sequence of length 1 for state q_1.

---

gen-1-UIO *Input*: State q_1

Step 1 Now check if any element of $Oled$ contains only one edge. From the computation done earlier, note that there is only one element in $Oled$, and that this element, $Oled[1]$, contains two edges. Hence there is no $UIO[q_1]$ with only a single label. The *gen-1-UIO* procedure is now terminated and the control returns to *gen-UIO*.

---

gen-uio *Input*: State q_1

Step 5 Determine if a simple UIO sequence was found. As it has not been found, move to the next step.

Step 6 Procedure *gen-long-UIO* is invoked in an attempt to generate a longer $UIO[q_1]$.

---

gen-long-UIO *Input*: State q_1

Step 1 $L = 1$.

Step 2 Start a loop to determine if a $UIO[q_1]$ of length greater than 1 exists. This loop terminates when a UIO is found or when $L = 2n^2$, which for a machine with $n = 6$ states translates to $L = 72$. Currently L is less than 72 and hence continue with the next step in this procedure.

Step 2.1 $L = 2$ and $k = 0$.

Step 2.2 Start another loop to iterate over the edges in $Oedges$. Set $i = 1$ and $e_i = (1, 2)$.

Step 2.2.1 $t = tail((1, 2)) = 2$. $Tedges(2) = \{(2, 3), (2, 6)\}$.

Step 2.2.2 Yet another loop begins at this point. Loop over the elements of $Tedges$. First set $te = (2, 3)$ and invoke *gen-L-UIO*.

---

gen-L-UIO *Input*: $te = (2, 3)$

Step 1 $k = 1$, $Pattern[1] = Opattern[1].label((2, 3)) = a/0.a/0$.

Step 2 $End[1] = 3$, $Led[1] = \varnothing$.

Step 3 Now iterate over elements of $Oled[1] = \{(2, 3), (3, 4)_{a/0}\}$. First select $oe = (2, 3)$ for which $h = 2$ and $t = 3$.

Step 3.1 Another loop is set up to iterate over elements of $OutEdges(q_3) = \{(3, 4)_{a/0}, (3, 4)_{b/1}\}$. Select $o = (3, 4)_{a/0}$.

Step 3.1.1 As $label((3, 4)_{a/0}) = label((2, 3))$, set $Led[1] = \{(2, 4)\}$.
Next select $o = (3, 4)_{b/1}$ and execute Step 3.1.1.

Step 3.1.1 As $label((2, 3)) \neq label((3, 4)_{b/1})$, no change is made to $Led[1]$. The iteration over $Oedges$ terminates.
Next, continue from Step 3.1 with $oe = (3, 4)_{a/0}$ for which $h = 3$ and $t = 4$.

Step 3.1 Another loop is set up to iterate over elements of $OutEdges(4) = \{(4, 5)\}$, select $o = (4, 5)$.

Step 3.1.1 As $label((4, 5)) \neq label((3, 4)_{a/0})$ no change is made to $Led[1]$.
The iteration over $Oedges$ terminates. Also, the iteration over $Oled[1]$ terminates.

Step 3.4 $Led[1]$ is not empty which implies that this attempt to find a $UIO[q_1]$ has failed. Note that in this attempt the algorithm checked if $a/0.a/0$ is a valid $UIO[q_1]$. Now return to the caller.

---

gen-long-UIO *Input*: State q_1

Step 2.2.2 Select another element of $Tedges$ and invoke *gen-L-UIO(e, te)* . For this iteration $te = (2, 6)$.

---

gen-L-UIO *Input*: $te = (2, 6)$

Step 1 $k = 2$, $Pattern[2] = Opattern[1].label((2, 6)) = a/0.c/1$.

Step 2 $End[2] = 6$, $Led[2] = \varnothing$.

Step 3 Iterate over elements of $Oled[1] = \{(2, 3), (3, 4)_{a/0}\}$. First select $oe = (2, 3)$ for which $h = 2$ and $t = 3$.

Step 3.1 Another loop is set up to iterate over elements of $OutEdges(3) = \{(3, 4)_{a/0}, (3, 4)_{b/1}\}$, select $o = (3, 4)_{a/0}$.

Step 3.1.1 As $label((3, 4)_{a/0}) \neq label((2, 6))$, do not change $Led[2]$.
Next select $o = (3, 4)_{b/1}$ and once again execute Step 3.1.1.

Step 3.1.1 As $label((2, 6)) \neq label((3, 4)_{b/1})$, do not change $Led[2]$. The iteration over $Oedges$ terminates.

Next continue from Step 3.1 with $oe = (3,4)_{a/0}$ for which $h = 3$ and $t = 4$.

Step 3.1 Another loop is set up to iterate over elements of $OutEdges(4) = \{(4,5)\}$, select $o = (4,5)$.

Step 3.1.1 As $label((4,5) \neq label((3,4)_{a/0})$ no change is made to $Led[2]$.

The iteration over $Oedges$ terminates. Also, the iteration over $Oled[(1,2)]$ terminates.

Step 3.4 $Led[2]$ is empty. Hence $UIO[q_1] = Pattern[2] = a/0.c/1$. A UIO of length 2 for state q_1 is found and hence the algorithm terminates.

6.6.5. Distinguishing signatures

As mentioned earlier, the *gen-uio* procedure might return an empty $UIO(s)$ indicating that it failed to find a UIO for state s. In this case we compute a signature that distinguishes s from other states one by one. We use $Sig(s)$ to denote a signature of state s. Before we show the computation of such a signature, we need some definitions. Let $W(q_i, q_j), i \neq j$ be a sequence of edge labels that distinguishes states q_i and q_j. Note that $W(q_i, q_j)$ is similar to the distinguishing sequence W for q_i and q_j except that we are now using edge labels of the kind a/b, where a is an input symbol and b an output symbol, instead of using only the input symbols.

EXAMPLE 6.22. A quick inspection of M2 in Figure 6.14, reveals the following distinguishing sequences.

$W(q_1, q_2) = a/0$
$W(q_1, q_3) = b/0$
$W(q_2, q_3) = b/0$

To check if indeed the above are correct distinguishing sequences, consider states q_1 and q_2. From the state transition function of M2, we get $\delta(q_1, a) = 0 \neq \delta(q_2, a)$. Similarly, $\delta(q_1, b) = 0 \neq \delta(q_3, b)$, and $\delta(q_2, a) = 0 \neq \delta(q_2, a)$. For a machine more complex than M2, one can find $W(q_i, q_j)$ for all pairs of states q_i and q_j using the algorithm in Section 6.4 and using edge labels in the sequence instead of the input symbols.

We use $P_i(j)$ to denote a sequence of edge labels along the shortest path from state q_j to q_i. $P_i(j)$ is known as a *transfer sequence* for state q_j to move the machine to state q_i. When the inputs along the edge labels of $P_i(j)$ are applied to a machine in state q_j, the machine moves to state q_i. For $i = j$, the null sequence is the transfer sequence. As we see later, $P_i(j)$ is used to derive a signature when the *gen-uio* algorithm fails.

EXAMPLE 6.23. For M2, we have the following transfer sequences.

$P_1(q_2) = a/1$
$P_1(q_3) = b/1$
$P_2(q_1) = a/0$

$P_2(q_3) = a/0$
$P_3(q_1) = b/0$
$P_3(q_2) = b/0$

For M1 in Figure 6.14, we get the following subset of transfer sequences (you may derive the others by inspecting the transition function of $M1$).

$P_1(q_5) = c/0.c/1$
$P_5(q_2) = a/0.a/0.b/1$ or $P_5(2) = a/0.b/1.b/1$
$P_6(q_1) = a/0.c/1$

A transfer sequence $P_i(j)$ can be found by finding the shortest path from state q_j to q_i and catenating in order the labels of the edges along this path.

To understand how the signature is computed, suppose that *gen-uio* fails to find a UIO for some state $q_i \in Q$ where Q is the set of n states in machine M under consideration. The signature for s consists of two parts. The first part of the sequence is $W(q_i, q_1)$ which distinguishes q_i from q_1.

Now suppose that the application of sequence $W(q_i, q_1)$ to state q_i takes the machine to some state t_1. The second part of the signature for q_i consists of pairs $P_i(t_k).W(q_i, q_{k+1})$ for $1 \leq k < n$. Notice that the second part can be further split into two sequences.

The first sequence, $P_i(t_k)$, transfers the machine from t_k back to q_i. The second sequence, $W(q_i, q_{k+1})$ applies a sequence that distinguishes q_i from state q_{k+1}. Thus, in essense, the signature makes use of the sequences that distinguish q_i from all other states in the machine and the transfer sequences that move the machine back to state q_i prior to applying another distinguishing sequence. Given that $q_1 \in Q$ is the starting state of M, a compact definition of the signature for state $q_i \in Q$ follows.

$$
\begin{aligned}
Sig(q_i) &= W(q_1, q_2).(P_1(t_1).W(q_1, q_3)).(P_1(t_2).W(q_1, q_4)) \ldots .(P_1(t_n).W(q_1, q_n)), \\
&\quad \text{for } i = 1. \\
&= W(q_i, q_1).(P_i(t_1).W(q_i, q_2)).(P_i(t_2).W(q_i, q_3)) \ldots .(P_i(t_{i-2}).W(q_i, q_{i-1})) \\
&\quad .(P_i(t_i).W(q_i, q_{i+1})) \ldots .(P_i(t_{n-1}).W(q_i, q_n)), \\
&\quad \text{for } i \neq 1.
\end{aligned}
$$

EXAMPLE 6.24. We have seen earlier that *gen-long-uio* fails to generate a UIO sequence for state q_1 in M2 shown in Figure 6.14. Let us therefore apply the method described above for the construction of a signature for state q_1. The desired signature can be found by substituting the appropriate values in the following formula.

$Sig(q_1) = W(q_1, q_2).((P_1(t_1).W(q_1, q_2))$

From Example 6.22 we have the following distinguishing sequences for state q_1.

$W(q_1, q_2) = a/0$
$W(q_1, q_3) = b/0$

The application of $W(q_1, q_2)$ to state q_1 takes M2 to state q_2. Hence we need the transfer sequence $P_1(q_2)$ to bring M2 back to state q_1. From Example 6.23 we get $P_1(q_2) = a/1$. Substituting these values in the formula for $UIO(q_1)$ we obtain the desired signature.

$Sig(q_1) = a/0.a/1.b/0$

Later, while deriving test cases from UIO of different states, we will use signatures for states that do not possess a UIO.

6.6.6. Test generation

Let $M = (X, Y, Q, q_1, \delta, O)$ be an FSM from which we need to generate tests to test an implementation for conformance. Let E denote the set of core edges in M. m is the total number of core edges in M. Recall that edges corresponding to a reset input in each state are included in E. The following procedure is used to construct a total of m tests, each corresponding to the *tour* of an edge.

1. Find the UIO for each state in M.
2. Find the shortest path from the initial state to each of the remaining states. As before, the shortest path from the initial state q_1 to any other state $q_i \in Q$ is denoted by $P_i(q_1)$.
3. Construct an *edge tour* for each edge in M. Let $TE(e)$ denote a subsequence that generates a tour for edge e. $TE(e)$ is constructed as follows

   $TE(e) = P_{head(e)}(1).label(e).UIO(tail(e))$.
4. This step is optional. It is used to combine the M test subsequences generated in the previous step into one test sequence that generates the tour of all edges. This sequence is denoted by TA. It is sometimes referred to as β-sequence. TA is obtained by catenating pairs of reset input and edge tour subsequences as follows

   $TA = \times_{e \in E}((Re/\texttt{null}).TE(e))$.

TA is useful when the IUT under test can be brought automatically to its start state by applying a *Re*. In this case the application of TA will likely shorten the time to test the IUT. While testing an IUT, the application of *Re* might be possible automatically through a script that sends a "kill process" signal to terminate the IUT and, upon the receipt of an acknowledgment that the process has terminated, may restart the IUT for the next test.

The next example illustrates the test generation procedure using the UIO sequences generated from M1 shown in Figure 6.14.

EXAMPLE 6.25. The UIO sequences for each of the six states in M1 are reproduced below for convenience. Also included in the rightmost column are the shortest paths from state q_1 to the state corresponding to state in the leftmost column.

State(s)	UIO (s)	$P_i(q_1)$
q_1	a/0.c/1	`null`
q_2	c/1.c/1	a/0
q_3	b/1.b/1	a/0.a/0
q_4	b/1.c/0	a/0.a/0.a/0
q_5	c/0	a/0.a/0.a/0.b/1
q_6	c/1.a/0	a/0.c/1

We consider only the core edges while developing tests for each edge. For example, the self loop in state q_5 corresponding to inputs a and b, not shown in Figure 6.14, is ignored as it is not part of the core behavior of M1. Also note that edge (q_6, q_1) will be considered twice, one corresponding to the label $c/0$ and the other corresponding to label *Re*/`null`. Using the formula given above, we obtain the following tests for each of the 14 core edges.

Test count	Edge (e)	$TE(e)$
1	q_1, q_2	a/0.c/1.c/1
2	q_1, q_1	*Re*/`null`.*Re*/`null`.a/0.c/1
3	q_2, q_3	a/0.a/0.b/1.b/1
4	q_2, q_6	a/0.c/1.c/1.a/0
5	q_2, q_1	a/0.*Re*/`null`.a/0.c/1
6	q_3, q_4	a/0.a/0.a/0.b/1.c/0
7	q_3, q_4	a/0.a/0.b/1.b/1.c/0
8	q_3, q_1	a/0.a/0.*Re*/`null`.a/0.c/1
9	q_4, q_5	a/0.a/0.a/0.c/0
10	q_4, q_1	a/0.a/0.a/0.*Re*/`null`.a/0.c/1
11	q_5, q_6	a/0.a/0.b/1.c/0.c/1
12	q_5, q_1	a/0.a/0.a/0.b/1.*Re*/`null`.a/0.c/1
13	q_6, q_1	a/0.c/1.c/1.a/0.c/1
14	q_6, q_1	a/0.c/1.*Re*/`null`.a/0.c/1

The 14 tests derived above can be combined into a β-sequence and applied to the IUT. This sequence will exercise only the core edges. Thus, for example, the self loops in state q_5, corresponding to inputs a and b will not be exercised by the tests given above. It is also important to note that each $TE(e)$ subsequence is applied with the IUT in its start state implying that a *Re* input is applied to the IUT prior to exercising it with the input portion of $TE(e)$.

6.6.7. Test optimization

The set of test subsequences $TE(e)$ can often be reduced by doing a simple optimization. For example, if $TE(e_1)$ is a subsequence of test $TE(e_2)$, then $TE(e_1)$ is redundant. This is because the edges toured by $TE(e_1)$ are also toured by $TE(e_2)$, and in the same order. Identification and elimination of subsequences that are fully contained in another subsequence generally leads to a reduction in the size of the test suite. In addition, if two tests are identical then one of them can be removed from further consideration. (See Exercise 6.18.)

EXAMPLE 6.26. In an attempt to reduce the size of the test set derived in Example 6.25, we examine each test and check if it is contained in any of the remaining tests. We find that test 3 is fully contained in test 7, test 1 is contained in test 4, and test 4 is contained in test 13. Thus the reduced set of tests consists of 11 tests: 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14 given in Example 6.25.

The tests derived in Example 6.25 are useful for a weak conformance test. To perform a strong conformance test of the IUT against the specification, we need to derive tests for non-core edges as well. The method for deriving such tests is the same as that given earlier for deriving $TE(e)$ for the tour of edge e except that e now includes non-core edges.

EXAMPLE 6.27. We continue Example 6.25 for M1 and derive additional tests needed for strong conformance. To do so we need to identify the non-core edges. There are 10 non-core edges corresponding to the six states. Figure 6.17 shows the state diagram of M with both core and non-core edges shown. Tests that tour the non-core edges can be generated easily using the formula for TE given earlier. The 10 tests are given below.

Test count	**Edge (e)**	$TE(e)$
1	$(q_1, q_1)_b$/null	b/null.a/0.c/1
2	$(q_1, q_1)_c$/null	c/null.a/0.c/1
3	$(q_2, q_2)_b$/null	a/0.b/null.c/1.c/1
4	$(q_3, q_3)_c$/null	a/0.a/0.c/null.b/1.b/1
5	$(q_4, q_4)_a$/null	a/0.a/0.a/0.a/null.b/1.c/0
6	$(q_4, q_4)_c$/null	a/0.a/0.a/0.c/null.b/1.c/0
7	$(q_5, q_5)_a$/null	a/0.a/0.a/0.b/1.a/null.c/0
8	$(q_5, q_5)_b$/null	a/0.a/0.a/0.b/1.b/null.c/0
9	$(q_6, q_6)_a$/null	a/0.c/1.a/null.c/1.a/0
10	$(q_6, q_6)_b$/null	a/0.c/1.b/null.c/1.a/0

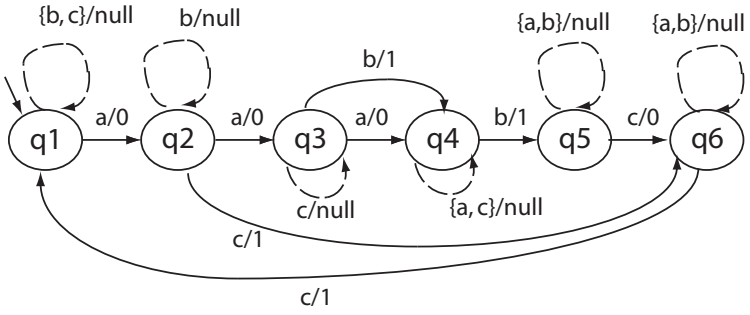


Figure 6.17: Machine M1 from Figure 6.14 with all non-core edges shown as dashed lines.

Note that a test for non-core edge e is similar to the tests for core edges in that the test first moves the machine to $head(e)$. It then traverses the edge itself. Finally it applies $UIO(tail(e))$ starting at state $tail(e)$. Thus in all we have 21 tests to check for strong conformance of an IUT

against M.

6.6.8. Fault detection

Tests generated using the UIO sequences are able to detect all operation and transfer errors. However, combination faults, such as an operation error and a transfer error might not be detectable. The next two examples illustrate the fault detection ability of the UIO method.

EXAMPLE 6.28. Consider the transition diagram in Figure 6.18. Suppose that this diagram represents the state transitions in an IUT to be tested for strong conformance against machine M1 in Figure 6.17. The IUT has two errors. State q_2 has a transfer error due to which $\delta(q_2, c) = q_5$ instead of $\delta(q_2, c) = q_6$. State q_3 has an operation error due to which $O(q_3, b)$ is undefined.

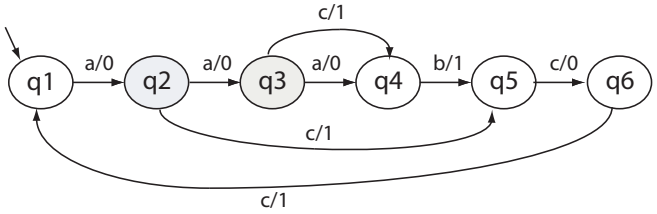


Figure 6.18: State diagram of an IUT containing two faults. State q_2 has a transfer error and q_3 has an operation error. The behavior of this IUT is to tested against that of the machine in Figure 6.17. Non-core edges are not shown.

To test the IUT against its specification as in Figure 6.17, one needs to apply the β sequence to the IUT and observe its behavior. The β sequence is obtained by combining all subsequences derived for weak and strong conformance in the earlier examples. However, as such a β sequence is too long to be presented here, we use an alternate method to show how the faults will be detected.

First, consider the transfer error in state q_2. Let us apply the input portion of test 4, which is $TE(q_2, q_6)$, to the IUT. We assume that the IUT is in its start state, i.e. q_1, prior to the application of the test. From Example 6.25, we obtain the input portion as $acca$. The expected behavior of the IUT for this input is determined by tracing the behavior of the machine in Figure 6.17. Such a trace gives us $O(q_1, acca) = 0110$. However, when the same input is applied to the IUT, we obtain $O(q_1, acca) = 010\texttt{null}$.

As the IUT behaves differently than its specification, $TE(q_2, q_6)$ has ben able to discover the fault. Note that if test 4 has been excluded due to optimization, then test 13 can be used instead. In this case the expected behavior is $O(q_1, accac) = 01101$ whereas the IUT generates $O(q_1, accac) = 010\texttt{null}1$ which also reveals a fault in the IUT.

Next let us consider the operation error in state q_3 along the edge $(q_3, q_4)_{c/1}$. We use test 7. The input portion of this test is $aabbc$. For the specification we have $O(q_1, aabbc) = 00110$. Assuming that this test is applied to the IUT in its start state, we get $O(q_1, aabbc) = 00\texttt{null}\texttt{null}1$ which is different from the expected output and thus test 7 has revealed the operation fault.

(Also see Exercise 6.21).

EXAMPLE 6.29. Consider the specification shown in Figure 6.19(a). We want to check if tests generated using the UIO method will be able to reveal the transfer error in the IUT shown in Figure 6.19(b). UIO sequences and shortest paths from the start state to the remaining two states are given below.

State(s)	**UIO (s)**	$P_i(q_1)$
q_1	a/1	null
q_2	a/0.a/1	a/0
q_3	b/1.a/1	b/1
	(also a/0.a/0)	

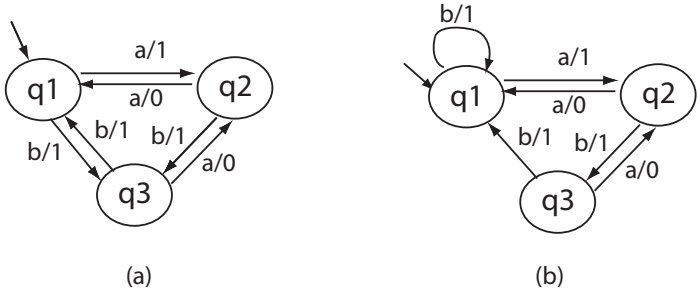


Figure 6.19: State diagram of an FSM for which a test sequence derived using the UIO approach does not reveal the error. (a) Specification FSM. (b) State diagram of a faulty IUT.

Given below are tests for touring each of the nine core edges including three edges that bring each state to state q_1 upon reset. Notice that we have added the *Re*/null transition at the start of each test sequence to indicate explicitly that the machine starts in state q_1 prior to the test being applied.

Test count	**Edge (e)**	$TE(e)$
Edges (e) from each state to state q_1, $label(e) =$ Re/null		
1	q_1, q_1	*Re*/null.*Re*/null.a/1
2	q_2, q_1	*Re*/null.a/1*Re*/null.a/1
3	q_3, q_1	*Re*/null.b/1.*Re*/null.a/1
Edges shown in Figure 6.19(a).		
4	q_1, q_2	*Re*/null.a/1.a/0.a/1
5	q_1, q_3	*Re*/null.b/1.b/1.a/1
6	q_2, q_1	*Re*/null.a/1.a/0.a/1
7	q_2, q_3	*Re*/null.a/1.b/1.b/1.a/1
8	q_3, q_1	*Re*/null.b/1.b/1.a/1
9	q_3, q_2	*Re*/null.b/1.a/0.a/0.a/1

To test the IUT, let us apply the input portion bba of test 5 that tours the edge (q_1, q_3). The expected output is 111. The output generated by the IUT is also 111. Hence the tour of edge (q_1, q_3) fails to reveal the transfer error in state q_1. However, test 9 that tours edge (q_3, q_2) does reveal the error because the IUT generates the output 1101 whereas the expected output is 1001, ignoring the `null` output. Note that tests 4 and 6 are identical and so are tests 5 and 8. Thus an optimal test sequence for this example is the set containing tests 1, 2, 3, 4, 5, 7, and 9.

6.7. The partial W-method

The partial W-method, also known as the *Wp-method*, is similar to the W-method in that tests are generated from a minimal, complete, and connected FSM. However, the size of the test set generated using the Wp-method is often smaller than that generated using the W-method. This reduction is achieved by dividing the test generation process into two phases and making use of the state identification sets W_i, instead of the characterization set W, in the second phase. Furthermore, the fault detection effectiveness of the Wp-method remains the same as that of the W-method. The two-phases used in the Wp method are described later in Section 6.7.1.

First we define a *state cover set* S of an FSM $M = (X, Y, Q, q_0, \delta, O)$. S is a finite non-empty set of sequences where each sequence belongs to X^* and for each state $q_i \in Q$ there exists an $r \in S$ such that $\delta(q_0, r) = q_i$. It is easy to see that the state cover set is a subset of the transition cover set and is not unique. Note that the empty string ϵ belongs to S which *covers* the initial state because, as per the definition of state transitions, $\delta(q_0, \epsilon) = q_0$.

EXAMPLE 6.30. In Example 6.7 we constructed the following transition cover set for machine M shown in Figure 6.9:

$P = \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\}$

The following subset of P forms a state cover set for M.

$S = \{\epsilon, b, ba, baa, baaa\}$

We are now ready to show how tests are generated using the Wp-method. As before, let M denote the FSM representation of the specification against which we are to test an IUT. We assume that M contains $n > 0$ states and the IUT contains M states. The test set T derived using the Wp-method is composed of subsets T_1 and T_2, i.e. $T = T_1 \cup T_2$. Below we show how to compute subsets T_1 and T_2 assuming that M and the IUT have an equal number of states, i.e. $m = n$.

`Procedure for test generation using the Wp-method.`

Step 1 Compute the transition cover set P, the state cover set S, the characterization set W, and the state identification sets W_i for M. Recall that S can be derived from P and the state identification sets can be computed as explained in Example 6.6.

Step 2 $T_1 = S.W$.

Step 3 Let $\mathcal{W}$ be the set of all state identification sets of M, i.e. $\mathcal{W} = \{W_1, W_2, \ldots, W_n\}$.

Step 4 Let $R = \{r_{i_1}, r_{i_2}, \ldots r_{i_k}\}$ denote the set of all sequences that are in the transition cover set P but not in the state cover set S, i.e. $R = P - S$. Furthermore, let $r_{i_j} \in R$ be such that $\delta(q_0, r_{i_j}) = q_{i_j}$.

Step 5 $T_2 = R \bigotimes \mathcal{W} = \bigcup_{j=1}^{k} (\{r_{i_j}\}.W_{i_j})$, where $W_{i_j} \in \mathcal{W}$ is the state identification set for state q_{i_j}.

`End of Procedure for test generation using the Wp-method.`

The $\bigotimes$ operator used in the derivation of T_2 is known as the *partial string concatenation operator*. Having constructed the test set T, which consists of subsets T_1 and T_2, one proceeds to test the IUT as described in the next section.

6.7.1. Testing using the Wp-method for $m = n$

Given a specification machine M and its implementation under test, the Wp-method consists of a two-phase test procedure. In the first phase the IUT is tested using the elements of T_1. In the second phase the IUT is tested against the elements of T_2. Details of the two phases follow.

Phase 1: Testing the IUT against each element of T_1 tests each state for equivalence with the corresponding state in M. Note that a test sequence t in T_1 is of the kind uv where $u \in S$ and $v \in W$. Suppose that $\delta(q_0, u) = q_i$ for some state q_i in M. Thus the application of t first moves M to some state q_i. Continuing with the application of t to M, now in state q_i, generates an output different from that generated if v were applied to M in some other state q_j, i.e. $O(q_i, v) \neq O(q_j, v)$, for $i \neq j$. If the IUT behaves as M against each element of T_1, then the two are equivalent with respect to $S.W$. If not, then there is an error in the IUT.

Phase 2: While testing the IUT using elements of T_1 checks all states in the implementation against those in M, it may miss checking all transitions in the IUT. This is because T_1 is derived using the state cover set S and not the transition cover set P. Elements of T_2 complete the checking of the remaining transitions in the IUT against M.

To understand how, we note that each element t of T_2 is of the kind uv where u is in P but not in S. Thus, the application of t to the M moves it to some state q_i where $\delta(q_0, u) = q_i$. However, the sequence of transitions that M goes through is different from the sequence it goes through while being tested against elements of T_1 because $u \notin S$. Next, M which is now in state q_i, is excited with v. Now v belongs to W_i which is the state identification set of q_i and hence this portion of the test will distinguish q_i from all other states with respect to transitions not traversed while testing against T_1. Thus, any transfer or operation errors in the transitions not tested using T_1, will be discovered using T_2.

EXAMPLE 6.31. We will now show how to generate tests using the Wp-method. For this example, let us reconsider the machine in Figure 6.9to be the specification machine M. First, various sets needed to generate the test set T are reproduced below for convenience.

$$
\begin{aligned}
W &= \{a, aa, aaa, baaa\} \\
P &= \{\epsilon, a, b, bb, ba, bab, baa, baab, baaa, baaab, baaaa\} \\
S &= \{\epsilon, b, ba, baa, baaa\} \\
W_1 &= \{baaa, aa, a\} \\
W_2 &= \{baaa, aa, a\} \\
W_3 &= \{a, aa\} \\
W_4 &= \{a, aaa\} \\
W_5 &= \{a, aaa\}
\end{aligned}
$$

Next, we compute set T_1 using Step 2 .

$$
\begin{aligned}
T_1 = S.W = \quad & \{a, aa, aaa, baaa, \\
& ba, baa, baaa, bbaaa, \\
& baa, baaa, baaa, babaaa \\
& baaa, baaaa, baaaaa, baabaaa, \\
& baaaa, baaaaa, baaaaaa, baaabaaa\}
\end{aligned}
$$

Next, in accordance with Step 4 we obtain R as

$$R = P - S = \{a, bb, bab, baab, baaab, baaaa\}.$$

With reference to Step 4 and Figure 6.9, we obtain the following state transitions for the elements in R:

$$
\begin{aligned}
\delta(q_1, a) &= q_1 \\
\delta(q_1, bb) &= q_4 \\
\delta(q_1, bab) &= q_5 \\
\delta(q_1, baab) &= q_5 \\
\delta(q_1, baaab) &= q_5 \\
\delta(q_1, baaaa) &= q_1
\end{aligned}
$$

From the transitions given above we note that when M is excited by elements a, bb, bab, $baab$, $baaab$, $baaaa$, starting on each occasion in state q_1, it moves to states, q_1, q_4, q_5, q_5, q_5,, and q_1, respectively. Thus, while computing T_2, we need to consider the state identification sets W_1, W_4, and W_5. Using the formula in Step 5, we obtain T_2 as follows.

$$
\begin{aligned}
T_2 = R \bigotimes \mathcal{W} &= (\{a\}.W_1) \cup (\{bb\}.W_4) \cup (\{bab\}.W_5) \cup (\{baab\}.W_5) \cup \\
&\quad (\{baaab\}.W_5) \cup (\{baaaa\}.W_1) \\
&= \{abaaa, aaa, aa\} \cup \{bba, bbaaa\} \cup \{baba, babaaa\} \cup \\
&= \{baaba, baabaaa\} \cup \{baaaba, baaabaaa\} \cup \{baaaabaaa, baaaaaa, baaaaa\} \\
&= \{abaaa, aaa, aa, bba, bbaaa, baba, babaaa, baaba, baabaaa, baaaba, \\
&\quad baaabaaa, baaaabaaa, baaaaaa, baaaaa\}
\end{aligned}
$$

The desired test set $T = T_1 \cup T_2$. Note that T contains a total of 34 tests, with 20 from T_1 and 14 from T_2. This is in contrast to the 44 tests generated using the W-method in Example 6.8 when $m = n$.

The next example illustrates how to use the Wp-method to test an IUT.

EXAMPLE 6.32. Let us assume that we are given the specification M as shown in Figure 6.11(a) and are required to test IUTs that correspond to designs M_1 and M_2 in Figure 6.11(b) and (c), respectively. For each test, we proceed in two phases as required by the Wp-method.

Test M_1, phase 1: We apply each element t of T_1 to the IUT corresponding to M_1 and compare $M_1(t)$ with the expected response $M(t)$. To keep this example short, let us consider test $t = baaaaaa$. We obtain $M_1(t) = 1101001$. On the contrary, the expected response is $M(t) = 1101000$. Thus test input t has revealed the transfer error in state q_2.

Test M_1, phase 2: This phase is not needed to test M_1. In practice, however, one would test M_1 against all elements of T_2. You may verify that none of the elements of T_2 reveal the transfer error in M_1.

Test M_2, phase 1: Test $t = baabaaa$ that belongs to T_1 reveals the error as $M_2(t) = 1100100$ and $M(t) = 110100$.

Test M_2, phase 2: Once again this phase is not needed to reveal the error in M_2.

Note that in both the cases in the example above, we do not need to apply phase 2 of the Wp-method to reveal the error. For an example that does require phase 2 to reveal an implementation error, refer to Exercise 6.12 that shows an FSM and its erroneous implementation and requires the application of phase 2 for the error to be revealed. However, in practice one would not know whether or not phase 1 is sufficient. This would lead to the application of both phases and hence all tests. Note that tests in phase 2 ensure the coverage of all transitions. While tests in phase 1 might cover all transitions, they might do not apply the state identification inputs. Hence, all errors corresponding to the fault model in Section 6.3, are not guaranteed to be revealed by tests in phase 1.

6.7.2. Testing using the Wp-method for $m > n$

The procedure for constructing tests using the Wp-method can be modified easily when the number of states in the IUT is estimated to be larger than that in the specification, i.e. when $m > n$. The modifications required to the procedure described on page 218 are in Step 2 and Step 5.

For $m = n$, we compute $T_1 = S.W$. For $m > n$ we modify this formula so that $T_1 = S.Z$ where $Z = X[m-n].W$ as explained in Section 6.5.4. Recall that T_1 is used in phase 1 of the test procedure. T_1 is different from T derived from the W-method in that it uses the state cover set S and not the transition cover set P. Hence T_1 contains fewer tests than T except when $P = S$.

To compute T_2, we first compute R as in Step 4 on page 218. Recall that R contains only those elements of the transition cover set P that are not in the state cover set S. Let $R = P - S = \{r_{i_1}, r_{i_2}, \ldots r_{i_k}\}$. As before, $r_{i_j} \in R$ moves M from its initial state to state q_{i_j}, i.e. $\delta(q_0, r_{i_j}) = q_{i_j}$. Given R, we derive T_2 as follows.

$$T_2 = R.X[m-n] \bigotimes \mathcal{W} = \bigcup_{j=1}^{k} \{r_{i_j}\}.(\bigcup_{u \in X[m-n]} u.W_l),$$

where $\delta(q_0, r_{i_j}) = q_{i_j}$, $\delta(q_{i_j}, u) = q_l$, and $W_l \in \mathcal{W}$ is the identification set for state q_l.

The basic idea underlying phase 2 is explained in the following. The IUT under test is exercised so that it reaches some state q_i from the initial state q_0. Let u denote the sequence of input symbols that move the IUT from state q_0 to q_i. As the IUT contains $m > n$ states, it is now forced to take an additional $(m - n)$ transitions. Of course, it requires $(m - n)$ input symbols to force the IUT for that many transitions. Let v denote the sequence of symbols that move the IUT from state q_i to state q_j in $(m - n)$ steps. Finally, the IUT is input a sequence of symbols, say w, that belong to the state identification set W_j. Thus one test for the IUT in phase 2 is comprised of the input sequence uvw.

Notice that the expression to compute T_2 consists of two parts. The first part is R and the second part is the partial string concatenation of $X[m - n]$ and $\mathcal{W}$. Thus a test in T_2 can be written as uvw where $u \in R$ is the string that takes the IUT from its initial state to some state q_i, string $v \in X[m - n]$ takes the IUT further to state q_j and string $w \in W_j$ takes it to some state q_l. If there is no error in the IUT, then the output string generated by the IUT upon receiving the input uvw must be the same as the one generated when the design specification M is exercised by the same string.

EXAMPLE 6.33. We will now show how to construct a test set using the Wp-method for machine M in Figure 6.9 given that the corresponding IUT contains an extra state. For this scenario we have $n = 5$ and $m = 6$. Various sets needed to derive T are reproduced here for convenience.

$$
\begin{aligned}
X &= \{a, b\} \\
W &= \{a, aa, aaa, baaa\} \\
P &= \{\epsilon, a, b, bb, ba, baa, bab, baab, baaa, baaab, baaaa\} \\
S &= \{\epsilon, b, ba, baa, baaa\} \\
W_1 &= \{baaa, aa, a\} \\
W_2 &= \{baaa, aa, a\} \\
W_3 &= \{a, aa\} \\
W_4 &= \{a, aaa\} \\
W_5 &= \{a, aaa\}
\end{aligned}
$$

First we derive derive T_1 as $S.X[1].W$. Recall that $X[1]$ denotes the set $\{\epsilon\} \cup X^1$.

$$
\begin{aligned}
T_1 &= S.(\{\epsilon\} \cup X^1).W \\
&= (S.W) \cup (S.X.W) \\
S.W &= \{a, aa, aaa, baaa, \\
&\quad ba, baa, baaa, bbaaa, \\
&\quad baa, baaa, baaaa, babaaa, \\
&\quad baaa, baaaa, baaaaa, baabaaa, \\
&\quad baaaa, baaaaa, baaaaaa, baaabaaa\} \\
S.X &= \{a, b, ba, bb, baa, bab, baaa, baab, baaaa, baaab\}
\end{aligned}
$$

$$
\begin{aligned}
S.X.W \quad = \quad & \{aa, aaa, aaaa, abaaa, \\
& ba, baa, baaa, bbaaa, \\
& baa, baaa, baaaa, babaaa, \\
& bba, bbaa, bbaaa, bbbaaa, \\
& baaa, baaaa, baaaaa, baabaaa, \\
& baba, babaa, babaaa, babbaaa, \\
& baaaa, baaaaa, baaaaaa, baaabaaa, \\
& baaba, baabaa, baabaaa, baabbaaa, \\
& baaaaa, baaaaaa, baaaaaaa, baaaabaaa, \\
& baaaba, baaabaa, baaabaaa, baaabaaa\}
\end{aligned}
$$

T_1 contains a total of 60 tests of which 20 are in $S.W$ and 40 in $S.X.W$. To obtain T_2 we note that $R = P - S = \{a, bb, bab, baab, baaab, baaaa\}$. T_2 can now be computed as follows.

$$
\begin{aligned}
T_2 \quad & = \quad R.X[m-n] \bigotimes \mathcal{W} \\
& = \quad (R \bigotimes \mathcal{W}) \cup (R.X \bigotimes \mathcal{W}) \\
R \bigotimes \mathcal{W} \quad & = \quad (\{a\}.W_1) \cup (\{bb\}.W_4) \cup (\{baab\}.W_5) \cup \{bab\}.W_1) \cup \{baaab\}.W_1 \cup \{baaaa\}.W_5 \\
& = \quad \{abaaa, aaa, aa, bba, bbaaa, baaba, baabaaa\} \\
R.X \bigotimes \mathcal{W} \quad & = \quad (aa.W_1) \cup (ab.W_4) \cup \\
& \quad (bba.W_3 (\cup (bbb.W_4) \cup \\
& \quad (baaba.W_2) \cup (baabb.W_5) \\
& \quad (baba.W_1) \cup (babb.W_4) \\
& \quad (baaaba.W_2) \cup (baaabb.W_5) \\
& \quad (baaaaa.W_1) \cup (baaaab.W_5) \\
& = \quad \{aabaaa, aaaa, aaa, aba, abaaa, bbaa, bbaaa, bbba, \\
& \quad bbbaaa, baababaaa, baabaaa, baabaa, baabba, baabbaaa\}
\end{aligned}
$$

T_2 contains a total of 21 tests. In all the entire test set $T = T_1 \cup T_2$ contains 81 tests. This is in contrast to a total of 81 tests that would be generated by the W-method. (See Exercises 6.13 and 6.14.)

6.8. Automata theoretic versus control-flow based techniques

The test generation techniques described in the this chapter fall under the *automata theoretic* category. There exist other techniques that fall under the *control-flow based* category. Here we compare the fault detection effectiveness of some techniques in the two categories.

Several empirical studies have aimed at assessing the fault detection effectiveness of test sequences generated from FSMs by various test generation methods. These studies are described in some detail in Chapter 17. Here we compare the fault detection effectiveness of the W and Wp-methods with four control-flow based criteria to assess the adequacy of tests. Some of the control theoretic se can be applied to assess the adequacy of tests derived from FSMs against the FSM itself. In this section we define four such criteria and show that test derived from the W- and Wp-methods are superior in terms of their fault detection effectiveness than the four control flow based methods considered.

Tests generated using the W- and the Wp methods guarantee the detection of all missing

transitions, incorrect transitions, extra or missing states, and errors in the output associated with a transition given that the underlying assumptions listed in Section 6.5.1 hold. We show through an example that tests generated using these methods are more effective in detecting faults than the tests that are found adequate with respect to *state cover*, *branch cover*, *switch cover*, and the *boundary interior* cover test adequacy criteria. First a few definitions before we illustrate this fact.

State cover:

A test set T is considered adequate with respect to the state cover *criterion for an FSM M if the execution of M against each element of T causes each state in M to be visited at least once.*

Transition cover:

A test set T is considered adequate with respect to the branch*, or transition, cover criterion for an FSM M if the execution of M against each element of T causes each transition in M to be taken at least once.*

Switch cover:

A test set T is considered adequate with respect to the 1-switch cover *criterion for an FSM M if the execution of M against each element of T causes each pair of transitions (tr_1, tr_2) in M to be taken at least once, where for some input substring $ab \in X^*$, $tr_1 : q_i = \delta(q_j, a)$ and $tr_2 : q_k = \delta(q_i, b)$ and q_i, q_j, q_k are states in M.*

Boundary-interior cover:

A test set T is considered adequate with respect to the boundary-interior cover *criterion for an FSM M if the execution of M against each element of T causes each loop to body to be traversed zero times and at least once. Exiting the loop upon arrival covers the "boundary" condition and entering it and traversing the body at least once covers the "interior" condition.*

The next example illustrates weaknesses of the state cover, branch cover, switch cover, and the boundary interior cover test adequacy criteria.

EXAMPLE 6.34. Machine M1 in Figure 6.20 represents correct design while M1' has an output error in state q_2. Consider the input sequence $t = abba$. t covers all states and all transitions in M1 and hence is adequate with respect to the state coverage and transition coverage criteria. However, we obtain $\delta_{M1}(q_1, t) = 0111 = \delta_{M1'}(q_1, t)$. While t covers all states and all transitions (branches) in M1 and M1', it does not reveal the transfer error in M1'.

Machine M2 in Figure 6.21 represents correct design while M2' has an output error in state q_3. In order for a test set to be adequate with respect to the switch cover criterion, it must cause the following set of branch pairs to be exercised.

$$\begin{aligned} S \quad = \quad & \{(tr_1, tr_2), (tr_1, tr_3), (tr_2, tr_2), (tr_2, tr_3), (tr_3, tr_4), (tr_3, tr_5), \\ & (tr_4, tr_4), (tr_4, tr_5), (tr_5, tr_6), (tr_5, tr_1), (tr_6, tr_4), (tr_6, tr_5)\} \end{aligned}$$

The following table lists a set of test sequences adequate with respect to the switch cover criterion but do not reveal the transfer error in state q_3. The second column in the table shows the output generated and the rightmost column lists the switches covered.

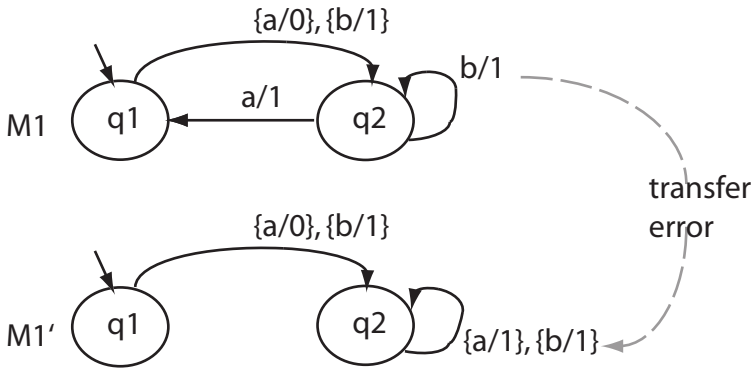


Figure 6.20: Machine M1′ contains a transfer fault with respect to machine M1 in state q_2.

Test sequence (t)	$O_{M2}(q_1, t)$ $(= O_{M2'}(q_1, t))$	**Switches covered**
$abbaaab$	0111001	$(tr_1, tr_2), (tr_2, tr_2), (tr_2, tr_3), (tr_3, tr_4),$ $(tr_4, tr_4), (tr_4, tr_5)$
$aaba$	0110	$(tr_1, tr_3), (tr_3, tr_5), (tr_5, tr_1)$
$aabb$	0110	$(tr_1, tr_3), (tr_3, tr_5), (tr_5, tr_6)$
$baab$	0001	$(tr_6, tr_4), (tr_4, tr_4), (tr_4, tr_5)$
bb	01	(tr_6, tr_5)

A simple inspection of machine M2 in Figure 6.21 reveals that that all states are 1-distinguishable, i.e. for each pair of states (q_i, q_j) there exists a string of length 1 that distinguishes q_i from q_j, $1 \leq (i, j) \leq 3, i \neq j$. Later we define an n-switch cover and show how to construct a n-switch cover that will reveal all transfer and operation error for an FSM that is n-distinguishable.

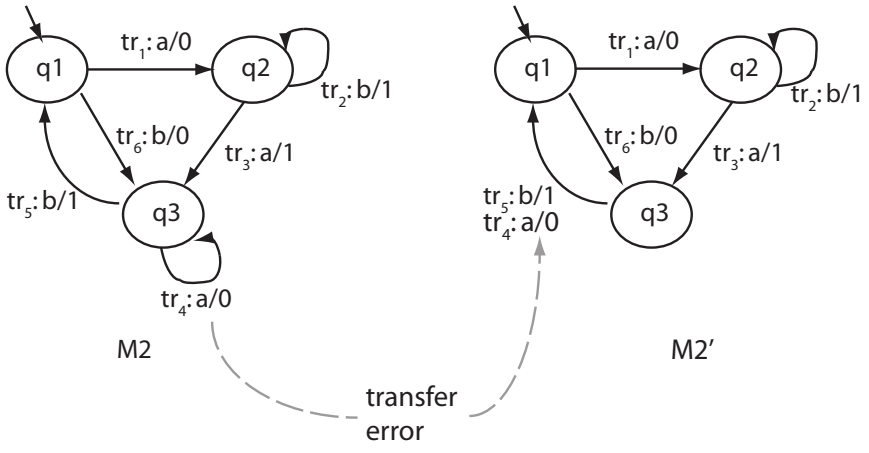


Figure 6.21: Machine M2′ contains a transfer fault with respect to machine M2 in state q_3.

In Figure 6.22 we have machine M3' that has a transfer error in state q_2 with respect to machine M3. There are two loops in M3, one in state q_2 and the other in q_3. Test sequence $t_1 =$

aab causes both loops to exit without ever looping in either state q_2 or q_3. Also, test sequence $t_2 = abaab$ causes each state to be entered once and exited. Thus the set $T = \{t_1, t_2\}$ is adequate with respect to the boundary-interior cover criterion. We note that $O_{M3}(q_1, t_1) = O_{M3'}(q_1, t_1) = 000$ and $O_{M3}(q_1, t_2) = O_{M3'}(q_1, t_2) = 00010$. We conclude that T it is unable to distinguish M3 from M3' and hence does not reveal the error. Notice that machine M3 is 1-distinguishable.

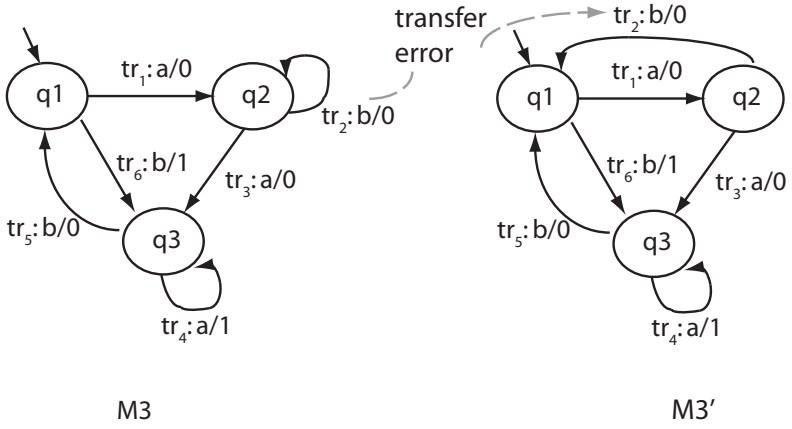


Figure 6.22: Machine M3′ contains a transfer fault with respect to machine M3 in state q_2.

6.8.1. n-switch-cover

The switch-cover criterion can be generalized to an n-switch cover criterion as follows. An *n-switch* is a sequence of $(n + 1)$ transitions. For example, for machine M3 in Figure 6.22, transition sequence tr_1 is a 0-switch, tr_1, tr_2 is a 1-switch, tr_1, tr_2, tr_3 is a 2-switch, and transition sequence tr_6, tr_5, tr_1, tr_3 is a 3-switch. For a given integer $n > 0$, we can define a n-switch set for a transition tr as the set of all n-switches with tr as the prefix. For example, for each of the six transitions in Figure 6.22, we have the following 1-switch sets.

$tr_1 : \{(tr_1, tr_2), (tr_1, tr_3)\}$

$tr_2 : \{(tr_2, tr_2), (tr2, tr_3)\}$

$tr_3 : \{(tr_3, tr_4), (tr_3, tr_5)\}$

$tr_4 : \{(tr_4, tr_4), (tr_4, tr_5)\}$

$tr_5 : \{(tr_5, tr_6), (tr_5, tr_1)\}$

$tr_6 : \{(tr_6, tr_4), (tr_6, tr_5)\}$

An n-switch set S for a transition tr in FSM M is considered covered by a set T of test se-

quences if exercising M against elements of T causes each transition sequence in S to be traversed. T is considered an *n-switch set cover* if it covers all n-switch sets for FSM M. It can be shown that an n-switch set cover can detect all transfer, operation, extra, and missing state errors in a minimal FSM that is n-distinguishable (see Exercise 6.25). Given a minimal, 1-distinguishable FSM M, the next example demonstrates a procedure to obtain a 1-switch cover using the testing tree of M.

EXAMPLE 6.35. Figure 6.23 shows a testing tree for machine M3 in Figure 6.22. To obtain the 1-switch cover we traverse the testing tree from the root and enumerate all complete paths. Each path is represented by the sequence s of input symbols that label the corresponding links in the tree. The sequence s corresponding to each path is concatenated with the input alphabet of M as $s.x$ where $x \in X$, X being the input alphabet. Traversing the testing tree in Figure 6.23 and concatenating as described gives us the following 1-switch cover.

$T = \{aba, abb, aaa, aab, baa, bab, bba, bbb\}$

We leave it for the reader to verify that T is a 1-switch cover for M3. Recall from Example 6.34 that the error in M3' is not revealed by a test set adequate with respect to the boundary-interior cover criterion. However, if a test set is derived using the method explained here, it will be adequate with respect to the 1-switch cover criterion and will always reveal the transfer, operation, missing, and extra state errors if each state is 1-distinguishable and the usual assumptions hold.

For machine M2', we note that test $aba \in T$ distinguishes M3 from M3' as $O_{M3}(q_1, abb) = 000 \neq O_{M3'}(q_1, abb)$. Exercise 6.27 asks for the derivation of a test set from M2 in Figure 6.21 that is adequate with respect to 1-switch cover criterion.

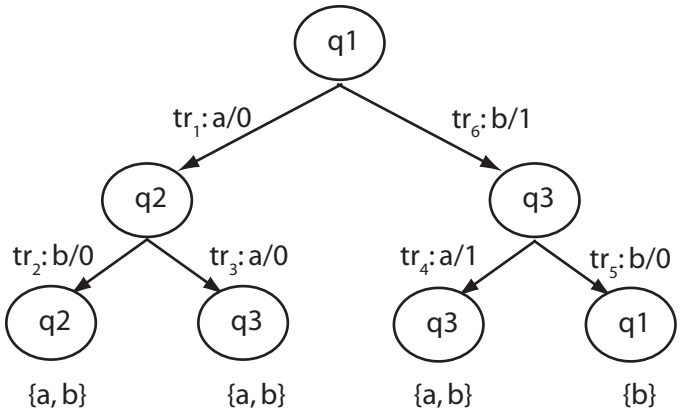


Figure 6.23: Testing tree of machine M3 in Figure 6.22.

6.8.2. Comparing automata theoretic methods

Figure 6.24 shows the relative fault detection effectiveness of the transition tour (TT), Unique Input/Output (UIO), UIOv, Distinguishing Sequence (DS), Wp, and W methods. As is shown in the figure, the W-method, UIOv, Wp, and DS can detect all faults using the fault model described in Section 6.3.

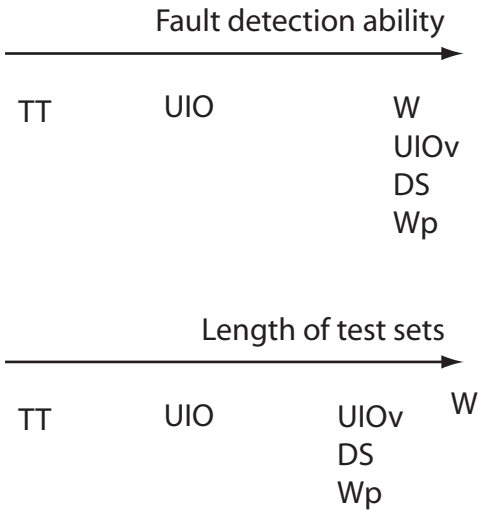


Figure 6.24: Relative fault detection effectiveness and length of the test sets of various automata theoretic techniques.

The distinguishing sequence method constructs an input sequence x from an FSM M such that $O(q_i, x)$ is different for each q_i in M. Sequences so constructed are able to detect all faults in our fault model. The UIO method is able to detect all output faults along transitions but not necessarily all transfer faults.

Note that the transition tour method has the lowest fault detection capability. In this method test sequences are generated randomly until all transitions in the underlying FSM are covered. Redundant sequences are removed using a minimization procedure. One reason for the low fault detection ability of the TT method is that it checks only whether or not a transition has been covered and does not check the correctness of the start and tail states of the transition (see Exercise 6.22).

The relative lengths of test sequences is shown in Figure 6.24. Once again the W-method generates the longest, and the largest, set of test sequences while the TT method the shortest. From the relative lengths and fault detection relationships we observe that larger tests tend to be more powerful then shorter tests. However, we also observe that smaller tests can be as powerful in their fault detection ability than larger tests as indicated by comparing the fault detection effectiveness of the Wp-method with that of the W-method.

6.9. Summary

The area of test generation from finite state models is wide. Research in this area can be categorized as follows:

- test generation techniques,
- empirical studies, and
- test methodology and architectures.

In this chapter we have focused primarily on test generation techniques. Of the many available, we have described three techniques-the W-method, UIO method, and the Wp method. We selected these for inclusion due to their intrinsic importance in the field of FSM-based test generation and their high fault detection effectiveness. each of these methods has also found its way into test generation from more complex models such as statecharts and timed automata as described in Chapters 7 and 8. Thus it is important for the reader to be familiar with these foundational test generation methods and prepare for more advanced and complex test generation methods.

Test generation from each of the techniques introduced in this chapter has been automated by several researchers. Nevertheless students may find it challenging to write their own test generators and conduct empirical studies on practical FSM models. Empirical studies aimed at assessing the fault detection effectiveness of various test generation methods and the lengths of the generated test sequences are discussed in Chapter 17. Test architectures and methodologies for testing based on finite state and other models are introduced in Chapter 3.

Bibliographic Notes

Finite state machines (FSMs) have long been in use as models of physical systems, software systems, and hybrid systems. Gill offers an excellent introduction to the theory of FSMs [159]. The algorithm for the construction of the characterization set W is found in Chapter 4, section 4.4 of Gill's book. Several other algorithms related to the processing and testing of FSMs are also found in this book. Another useful text on the theory of FSM's is due to Hennie [201].

Testing an IUT against an FSM is known as conformance testing. A significant research effort has been directed at the generation of test sequences from FSM models. Gonenc [164] describes early work in the design of experiments to test finite state machines using a set of distinguishing sequences. These sequences were constructed using the testing tree, also known as the distinguishing tree [247]. The W-method for the construction of test cases from FSMs was first proposed by Chow [75]; examples in Section 6.8 are minor modifications of those by Chow [75]. A solution to Exercise 6.25 is also found in [75].

The W-method of Chow led to a flurry of algorithms for test generation from FSMs, mostly improvements over the W-method in the size of the generated test set and in the efficiency of the test generation algorithm. Bernhard has proposed three variations of the W-method that in most cases generate smaller test suits with no reduction in the fault detection capability [35]. The partial-W method, also known as the Wp method, was proposed by Fujiwara et al. [146] as

an improvement over the W-method in terms of the size of the test set generated while retaining the fault detection ability.

Naito and Tsunoyama [324] proposed a transition tours algorithm. Generation of optimal transition tours using the Chinese Postman tours algorithm has been proposed by Uyar and Dahbura [450]. Sabnani and Dahbura [393] proposed the Unique Input/Output (UIO) sequence approach that generated significant research interest. Aho et al. [10] exploited the rural Chinese postman tours to reduce the size of tests generated using the UIO method. Chan et al. [64] suggested the UIOv method which is an improvement over the original UIO method in terms of its fault detection ability. Vuong and Ko [455] formulated test generation as a constraint satisfaction problem in artificial intelligence. Their method is as good as the UIO and W methods in terms of its fault detection ability and generates short test sequences.

Shen et al. proposed an optimization of the UIO method by proposing multiple UIO (MUIO) sequences for each state in the FSM [400]. Yang and Ural [482], Ural et al. [448], and Hierons and Ural [209] describe methods for further reducing the length of test sequences generated from an FSM [482]. Miller and Paul develop an algorithm for the generation of optimal length UIO sequences under certain conditions [311]. Naik [322] proposed an efficient algorithm for the computation of minimal length UIO sequences when they exist. Pomeranz and Reddy describe an approach to test generation from FSMs for the detection of multiple state-table faults [372]. Survey articles on conformance testing include those by Wang and Hutchison [457], Lee and Yannakakis [265], and Sarikaya [395].

A methodology and architectures for testing protocols is described by Sarikaya et al. [397] and Bochmann et al. [37]. Bochmann et al. study the problem of constructing oracles that would assess the validity of a trace of an IUT derived by executing the IUT against a test sequence; the IUT in this case being an implementation of a protocol specification. Sarikaya and Bochmann [396] provide upper bounds on the length of the test sequences for Naito and Tsunoyama's transition tours method and Chow's W-method.

Fault models for FSM's have been proposed by several researchers including Koufareva et al. [250] and Godskesen [163]. Several studies have been reported that assess the fault detection ability of various test generation methods. These include studies by Sidhu and Leung [404, 406] and Petrenko et al [365]. Sidhu and Chang have proposed probabilistic testing of protocols [405]. The study reported by Sidhu and Leung [406] used the National Bureau of Standards Class 4 transport protocol [417] as the subject for the comparison of T [324], UIO [393], D [164], and W-method [75]. Their results are summarized for the U- and the W-methods in Section 6.8. Karoui et al. [235] discuss factors that influence the testability and diagnostics of IUTs against FSM designs.

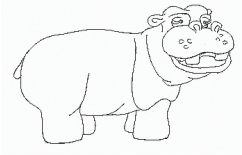
Several variations of FSM models and methods for testing them have been proposed. Extended Finite State machines (EFSMs) are FSMs with memory. Wang and Liu [458], Kim et al. [242], and Uyar and Duale [451] describe algorithms for generating tests from EFSMs. The problem of generating tests from a set of communicating FSMs has been addressed by Hierons [206], Lee et al. [264], and Gang et al. [148]. Gang et al. [148] have also proposed an extension of the Wp method to generate tests from a single nondeterministic FSM. Test suite minimization for nondeterministic FSMs is discussed by Yevtushenko et al. [484]. Petrenko and Yevtushenko describe an algorithm for generating tests from partial FSMs [366]. They define a weak conformance relation between the specification FSM and the FSM corresponding to the

object under test. El-Fakih et al. propose a method for generating tests from FSMs when the system specification and development happens incrementally [125]. They show that significant gains are obtained in the length of test suits when these are derived incrementally in contrast to directly from complete specifications. Shehady and Siewiorek [399] point to the weakness of an FSM in modeling GUIs with variables as part of their internal state and introduce VFSM (Variable Finite State Machine) as an alternative. VFSMs are obtained from FSMs by adding variables, and simple functions on variables, to transitions. The authors provide an automata-theoretic method for test generation from a VFSM.

Test adequacy assessment and test enhancement for FSMs based on mutation coverage has been proposed by Fabbri et al. [134]. Fabbri et al. [135] also report a tool, named Proteum/FSM, for testing FSM's. Gören and Ferguson [170] propose fault coverage of a test suite as an adequacy criterion and provide an algorithm for the incremental generation of tests. This method is an extension of the method proposed by the same authors for asynchronous sequential machines [169]. The test adequacy criteria defined in Section 6.8, and used for comparison with the W- and Wp-methods, have been considered by Howden [217], Huang [224], and Pimont and Rault [369].

The FSM-based test generation techniques described in this chapter have also been applied to the automatic generation of tests from SDL specifications [282]. Belina and Hogrefe [31, 32] provide an introduction to the SDL specification language.

EXERCISES



6.1 Show that the set W of Example 6.3 is a characterization set for the machine in Figure 6.9.

6.2 Prove that the FSM M must be a *minimal* machine for the existence of a characterization set.

6.3 Prove that the method for the construction of k-equivalence partitions described in Example 6.4 will always converge, i.e. there will be a table P_n, $n > 0$, such that $P_n = P_{n+1}$.

6.4 The W-procedure for the construction of the W-set from a set of k-equivalence partitions is given on page 187. This is a brute force procedure that determines a distinguishing sequence for every pair of states. From Example 6.5 we know that two or more pairs of states might be distinguishable by the same input sequence. Rewrite the W-procedure by making use of this fact.

6.5 Prove that the k-equivalence partition of a machine is unique.

6.6 Prove that in an FSM with n states, at most $n-1$ constructions of the equivalence classes are needed, i.e. one needs to construct only $P_1, P_2, \ldots, P_{n-1}$.

6.7 Given the FSM of Example 6.2, construct all mutants that can be obtained by adding a state to M.

6.8 Generate a set T of input sequences that distinguish all mutants in Figure 6.8 from machine M.

6.9 Show that any *extra* or *missing* state error in the implementation of design M will be detected at least one test generated using the W-method.

6.10 Construct the characterization set W and the transition cover for the machine in Figure 6.19(a). Using the W-method construct set Z assuming that $m = 3$ and derive a test set T. Does any element of T reveal the transfer error in the IUT of Figure 6.19(b) ? Compare T with the test set in Example 6.29 with respect to the number of tests and the average size of test sequences.

6.11 Consider the design specification M as shown in Figure 6.11(a). Further, consider an implementation M_3 of M as shown in Figure 6.25. Find all tests in T_1 and T_2 from Example 6.32 that reveal the error in M_3.

6.12 Consider the design specification in Figure 6.26(a). It contains three states q_0, q_1, and q_2. The input alphabet $X = \{a, b, c\}$, and the output alphabet is $Y = \{0, 1\}$. (a) Derive a transition cover set P, the state cover set S, the characterization set W, and state identification sets for each of the three states. (b) From M derive a test set T_w using the W-method and test set T_{wp} using the Wp-method. Compare the sizes of the two sets.

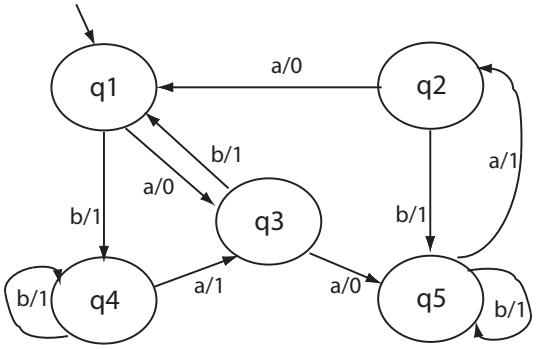


Figure 6.25: An implementation of M shown Figure 6.11(a) with a transfer error in state q_1 on input a.

(c) Figure 6.26(b) shows the transition diagram of an implementation of M that contains a transfer error in state q_2. Which tests in T_w and T_{wp} reveal this error ? (d) Figure 6.26(c) shows the transition diagram of an implementation of M that contains an extra state q_3, and a transfer error transfer error in state q_2. Which tests in T_w and T_{wp} reveal this error ?

6.13 (a) Given an FSM $M = (X, Y, Q, q_0, \delta, O)$ where $\mid X \mid = n_x$, $\mid Y \mid = n_y$, $\mid Q \mid = n_z$, calculate the upper bound on the number of tests that will be generated using the W-method. (b) Under what condition(s) will the number of tests generated by the Wp-method be the same as those generated by the W-method ?

6.14 Using the tests generated in Example 6.33 determine at least one test for each of the two machines in Figure 6.12 that will reveal the error. Is it necessary to apply phase 2 testing to reveal the error in each of the two machines ?

6.15 What is the difference between sequences in the W set and the UIO sequences ? Find UIO sequences for each state in the machine with transition diagram shown in Figure 6.9. Find the distinguishing signature for states for which a UIO sequence does not exist.

6.16 What will be the value of counter k when control arrives at Step 2.3 in procedure *gen-long-uio* on page 202 ?

6.17 For machine $M2$ used in Example 6.24, what output sequences are generated when the input portion of $Sig(q_1)$ is applied to states q_2 and q_3 ?

6.18 Suppose that tests $TE(e_1)$ and $TE(e_2)$ are derived using the method in Section 6.6.6 for a given FSM M. Is it possible that $TE(e_1) = TE(e_2)$?

6.19 Generate UIO sequences for all states in the specification shown in Figure 6.9. Using the sequences so generated, develop tests for weak conformance testing of an IUT built that must behave as per the specification given.

6.20 Generate tests for weak conformance testing of machine $M2$ in Figure 6.14. Use the UIO sequences for $M2$ given on page 199.

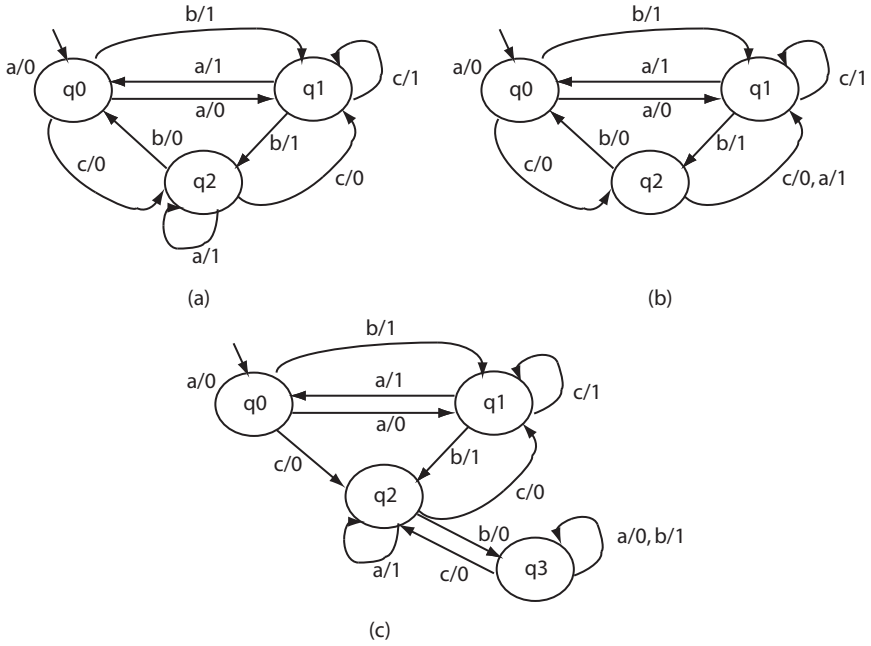


Figure 6.26: Three FSMs. (a) Design specification machine, (b) and (c) are machine indicating an erroneous implementation of the FSM in (a).

6.21 Consider an implementation of the machine shown in Figure 6.17. The state diagram of the implementation is shown in Figure 6.27. Note that the IUT has two errors, a transfer error in state q_2 and an operation error in state q_6. In Example 6.28, the transfer error in q_2 is revealed by test 4 (and 13). Are any of these tests successful in revealing the error in q_2 ? Is there any test derived earlier that will reveal the error in q_2 ? Which one of the tests derived in Example 6.25 and 6.27 will reveal the error in q_6 ?

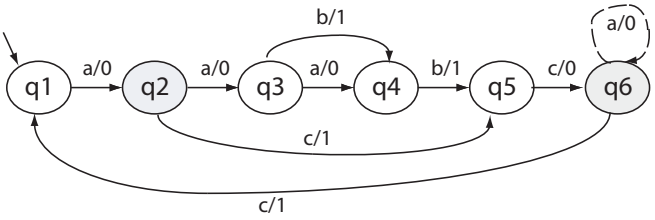


Figure 6.27: State diagram of an erroneous implementation of the machine in Figure 6.17.

6.22 Transition tours is a technique to generate tests from FSM specifications. A transition tour is an input sequence that when applied to an FSM in its start state traverses each edge at least once. (a) Find a transition tour for each of the two FSMs shown in Figure 6.14. (b) Show that a transition tour is able to detect all operation errors but may not be able to detect all transfer errors. Assume that the FSM specification satisfies the assumptions in Section 6.5.1. (c) Compare the size of tests generated using transition tours and the W-method.

6.23 In Example 6.34 we developed adequate tests to show that certain errors in the IUT corresponding to an FSM are not detected. Derive tests using the W-method, and then using the Wp-method, and show that each test set derived detects all errors shown in Figure 6.28.

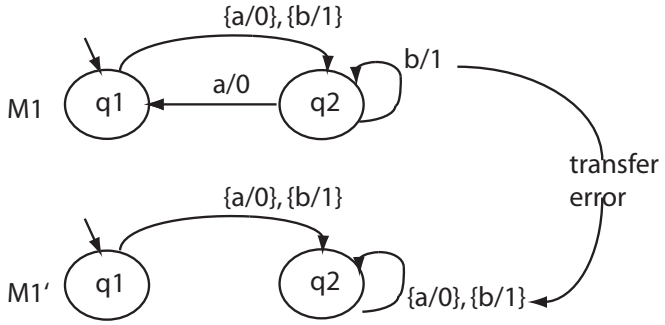


Figure 6.28: A transfer error.

6.24 FSM models considered in this chapter are pure in the sense that they capture only the control flow and ignore data definitions and uses. In this exercise you will learn how to enhance a test set derived using any of the methods described in this chapter by accounting for data flows.

Figure 6.29 shows machine M, an augmented version of the FSM in Figure 6.9. We assume that the IUT corresponding to the FSM in Figure 6.29 uses a local variable Z. This variable is *defined* along transitions $tr_1 = (q_1, q_4)$ and $tr_3 = (q_3, q_5)$. Variable Z is *used* along transitions $tr_2 = (q_2, q_1)$ and $tr_4 = (q_3, q_1)$. Also, x and y are parameters of, respectively, inputs b and a. A data flow path for variable Z is a sequence Tr of transitions such that Z is defined on one transition along Tr and, subsequently used along another transition also in Tr. For example, tr_1, tr_4, tr_5 is a data flow path for Z in M where Z is defined along tr_1 and used along tr_5. We consider only finite length paths and those with only one definition and one corresponding use along a path.

While testing the IUT against M, we must ensure that all data flow paths are tested. This is to detect faults in the defining or usage transitions. (a) Enumerate all data flow paths for variable Z in M. (b) Derive tests, i.e. input sequences, that will traverse all data flow paths derived in (a). (c) Consider a test set T derived for M using the W-method. Does exercising M against all elements of T exercise all data flow paths derived in (a) ? (Note: Chapter 15 provides details of data flow based assessment of test adequacy and enhancement of tests.)

6.25 Let T be a set of test inputs that forms an n-switch set cover for a minimal n-distinguishable FSM M. Prove that T can detect all transfer, operation, extra, and missing state errors in M.

6.26 Show that a test set T that is adequate with respect to the boundary-interior coverage criterion may not be adequate with respect to the 1-switch cover criterion.

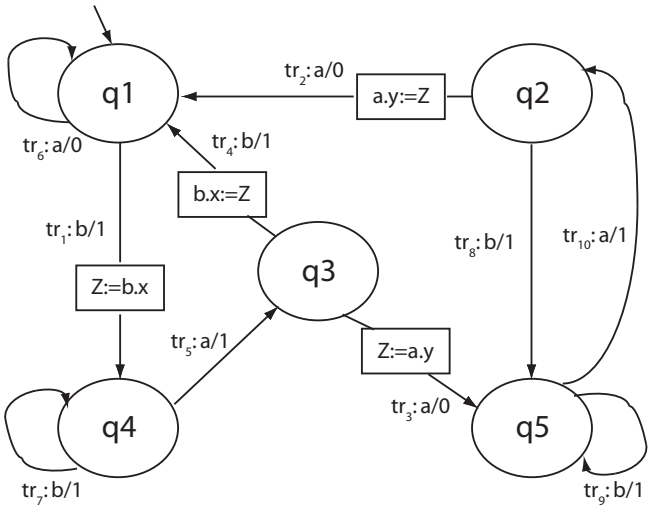


Figure 6.29: FSM of Figure 6.9 augmented with definition and use of local variable Z.

6.27 Derive a test set T from M2 shown in Figure 6.21 that is adequate with respect to the 1-switch cover criterion. Which test in T distinguishes M2 from M2' and thereby reveal the error ? Compare T derived in this exercise with the test given in Example 6.34 and identify a property of the method used to construct T that allows T to distinguish M2 from M2'.