4 Conformance Testing

Angelo Gargantini

Dipartimento di Matematica e Informatica
University of Catania
gargantini@dmi.unict.it

4.1 Introduction

In this chapter we tackle the problem of **conformance testing** between finite state machines. The problem can be briefly described as follows [LY96]. Given a finite state machine M_S which acts as specification and for which we know its transition diagram, and another finite state machine M_I which is the alleged implementation and for which we can only observe its behavior, we want to test whether M_I correctly implements or *conforms* to M_S. The problem of conformance testing is also called **fault detection**, because we are interested in uncovering where M_I fails to implement M_S, or **machine verification** in the circuits and switching systems literature.

We assume that the reader is familiar with the definitions given in Chapter 21, that we briefly report here. A finite state Mealy machine (FSM) is a quintuple $M = \langle I, O, S, \delta, \lambda \rangle$ where I, O, and S are finite nonempty sets of *input symbols*, *output symbols*, and *states*, respectively, $\delta : S \times I \to S$ is the *state transition function*, $\lambda : S \times I \to O$ is the *output function*. When the machine M is a current state s in S and receives an input a in I, it moves to the next state $\delta(s, a)$ producing the output $\lambda(s, a)$. An FSM can be represented by a state transition diagram as shown in Figure 4.1. $n = |S|$ denotes the number of states and $p = |I|$ the number of inputs. An input sequence x is a sequence $a_1, a_2, \ldots, a_k$ of input symbols, that takes the machine successively to states $s_{i+1} = \delta(s_i, a_i)$, $i = 1, \ldots, k$, with the final state s_{k+1} that we denote by $\delta(s_1, x)$. The input sequence x produces the output sequence $\lambda(s_1, x) = b_1, \ldots, b_k$, where $b_k = \lambda(s_i, a_i)$, $i = 1, \ldots, k$. Given two input sequences x and y, $x.y$ is the input sequence obtained by concatenating x with y.

The detection of faults in the implementation M_I can be performed by the following experiment. Generate a set of input sequences from the machine M_S. By applying each input sequence to M_S, generate the expected output sequences. Each pair of input sequence and expected output sequence is a test and the set of tests is a test suite (according to the definitions given in Chapter 20). Apply each input sequence to M_I and observe the output sequence. Compare this actual output sequence with the expected output sequence and if they differ, then a fault has been detected. As well known, this procedure of testing, as it has been presented so far, can only *be used to show the presence of bugs, but never to show their absence*[1]. The goal of this chapter is to present some techniques and

[1] Dijkstra, of course

M. Broy et al. (Eds.): Model-Based Testing of Reactive Systems, LNCS 3472, pp. 87-111, 2005.

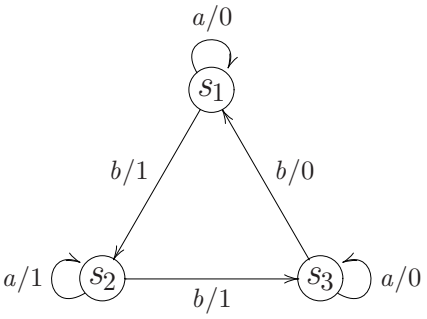


Fig. 4.1. Machine M_S [LY96]

algorithms able to detect faults of a well defined class, and to eventually prove, under some assumptions, that an implementation conforms to its specification. This chapter presents methods leaning toward the definition of ideal testing criteria as advocated in [GG75], i.e. test criteria that can discover any fault in the implementation (under suitable assumptions). Although this approach is rather theoretical, Section 4.8 presents the justifications for theoretical assumptions and the practical implications of the presented results.

Conformance is formally defined as equivalence or isomorphism (as defined in Chapter 21): M_I conforms to its specification M_S if and only if their initial states are equivalent, i.e. they will produce the same output for every input sequence. To prove this equivalence we look for a set of input sequences that we can apply to M_I to prove that it is equivalent to its specification. Note that successively applying each input sequence in the test set is equivalent to applying one input sequence that is obtained concatenating each input sequence in the set. Such an input sequence is called checking sequence.

Definition 4.1. (**Checking sequence**) A checking sequence for M_S is an input sequence that distinguishes the class of machines equivalent to M_S from all other machines.

Although all the presented methods share the unique goal to verify that M_I correctly implements M_S, generating a checking sequence (or a set of sequences, that concatenated act as a unique checking sequence), they differ for their cost to produce test sequences, for the total size of the test suite (i.e. the total length of the checking sequence), and for their fault detection capability. In fact, test suites should be rather short to be applicable in practice. On the other hand a test suite should cover the implementation as much as possible and detect as many faults as possible. The methods we present in this chapter differ with respect to the means and techniques they use to achieve these two opposite goals. However, the main difference among the methods we present regards the assumptions they make about the machines M_S and M_I. Some methods are very efficient to produce a checking sequence but usable only under very strong assumptions. Others produce exponentially long checking sequences, but perform

the test under more general assumptions. Therefore, the choice of one method instead of another is driven more by the facts we know or assume about the machines M_S and M_I, and these assumptions are of great importance.

4.2 Assumptions

Developing a technique for conformance testing without any assumption is impossible, because for every conformance test one can build a faulty machine that would pass such test. We have to introduce some assumptions about the machines we want to verify. These first four assumptions are necessary for every method we present in this chapter.

(1) M_S is *reduced* or *minimal*: we have to assume that machines are reduced, because equivalent machines have the same I/O behavior, and it is impossible to distinguish them by observing the outputs, regardless the method we use to generate the checking sequence. If M_S it is not minimal, we can minimize it and obtain an equivalent reduced machine (algorithms can be found in literature [Moo56, LY96] as well as in Chapter 21). In a minimal machine there are no equivalent states. For every pair of states s and t, there exists an input sequence x, called *separating sequence*, that can distinguish s from t because the outputs produced by applying x to s and t, $\lambda(s, x)$ and $\lambda(t, x)$ differ (see Section 1.3).
(2) M_S is *completely specified*: the state transition function δ and the output function λ are total: they are defined for every state in S and every input in I.
(3) M_S is *strongly connected*: every state in the graph is reachable from every other state in the machine via one or more state transitions. Note that some methods require only that all states are reachable from the initial one, allowing machines with deadlocks or states without any exiting transition. However these methods must require a reset message (Assumption 7) that can take the machine back to its initial state, otherwise a deadlock may stop the test experiment. The reset message makes de facto the machine strongly connected.
(4) M_I does not change during testing. Moreover it has the same sets of inputs and outputs as M_S. This implies that M_I can accept and respond to all input symbols in I (if the input set of M_I is a subset of the input set of M_S, we could redefine conformance).

The four properties listed above are requirements. Without them a conformance test of the type to be discussed is not possible. Unlike the first four requirements, the following assumptions are convenient but not essential. Throughout this chapter we present methods that can successfully perform conformance testing even when these assumptions do not hold.

(5) *Initial state*: machines M_I and M_S have an initial state, and M_I is in its initial state before we conduct a conformance test experiment. If M_I is not

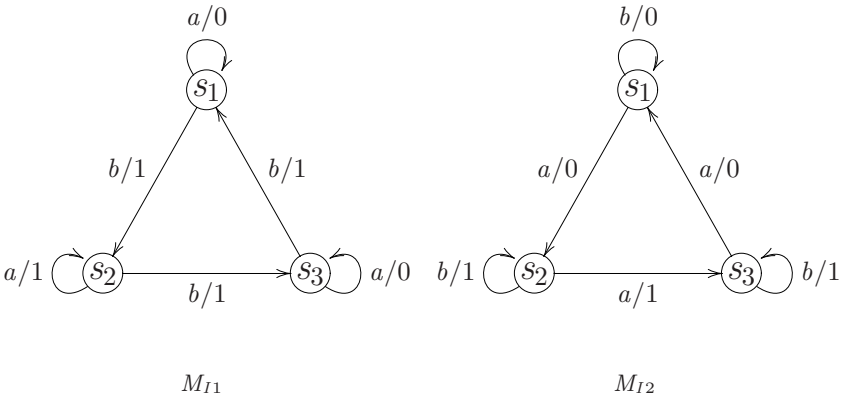


Fig. 4.2. Two faulty implementations of M_S

in its initial state we can apply a homing sequence (presented in Section 1.1.3) and then start the conformance test. If the machine M_I does not conform to its specification and the homing sequence fails to bring M_I to its initial state, this will be discovered during the conformance test. We denote the initial state by s_1.

(6) *Same number of states*: M_I has the same number of states as M_S, hence faults do not increase the number of states. Due to this assumption, the possible faults in M_I are of two kinds: *output faults*, i.e. a transition in the implementation produces the wrong output, and *transfer faults*, i.e. the implementation goes to a wrong state. Figure 4.2 shows two faulty implementations of the specification machine M_S given in Figure 4.1. Machine M_{I1} contains only one output fault for the transition from s_3 to s_1 with the input b: the output produced by M_{I1} is 1 instead of 0. Machine M_{I2} has several transfer faults: every transition moves the machine to a wrong final state. Moreover the transitions in M_{I2} from state s_3 and s_1 with input b produce wrong outputs.
Although this assumption is very strong, we show in Section 4.7 that many methods we present work well with modifications under the more general assumption that the number of states of M_I is bounded by an integer m, which may be larger than the number of states n in M_S.

(7) *reset* message: M_I and M_S have a particular input *reset* (or briefly r) that from any state of the machine causes a transition which ends in the initial state s_1 and produces no output. Formally, for all $s \in S$, $\delta(s, reset) = s_1$ and $\lambda(s, reset) = -$. Starting from Section 4.5 we present some methods that do not need a *reset* message.

(8) *status* message: M_I and M_S have a particular input *status* and they respond to a *status* message with an output message that uniquely identifies their current state. Since we label the states $s_1, s_2, ..., s_n$, we assume that *status* outputs the index i when applied to state s_i. The machines do not change

state. Formally for all $s_i \in S$, $\lambda(s_i, status) = i$ and $\delta(s_i, status) = s_i$. This rather strong assumption is relaxed starting from Section 4.4.

(9) *set* message: the input set I contains a particular set of inputs $set(s_j)$ and when a $set(s_j)$ message is received in the initial system state, the machines move to state s_j without producing any output. Formally for all $t \in S$, $\delta(reset, set(t)) = t$ and $\lambda(s, set(t)) = -$.

Given a machine with all the properties listed above, a simple conformance test can be performed as described by the simple Algorithm 9 (Chapter 9 of [Hol91]).

Algorithm 9 Conformance testing with a *set* message

For all $s \in S$, $a \in I$:

(1) Apply a *reset* message to bring the M_I to the initial state.
(2) Apply a $set(s)$ message to transfer M_I to state s.
(3) Apply the input message a.
(4) Verify that the output received conforms to the specification M_S, i.e. is equal to $\lambda_S(s, a)$
(5) Apply the *status* message and verify that the final state conforms to the specification, i.e. it is equal to $\delta_S(s, a)$

This algorithm verifies that M_I correctly implements M_S and it is capable to uncover any output or transfer fault. An output fault would be detected by step 4 while a transfer fault would be uncovered by step 5. Note that should the set of input signals I to be tested include the *set*, *reset*, and *status* messages, the algorithm must test also these messages. To test the *status* message we should apply it twice in every state s_i after the application of $set(s_i)$. The first application, given in step 3, is to check that in s_i the *status* message correctly outputs i (if also *set* is faulty and sets the current state to s_j instead of s_i and the *status* message in s_j has the wrong output i, we would discover this fault when testing s_j). The second application of *status*, given by step 5, is to check that the first application of *status* did not change the state. Indeed, if the first application of *status* in s_i did change the state to s_j and in s_j *status* is wrongly implemented and outputs i instead of j, we would discover this fault when testing s_j. Once that we are sure that *status* is correctly implemented, we can test *set* and *reset* applying them in every state and then applying *status* to check that they take the machine to the correct state.

Note that the resulting checking sequence obtained by Algorithm 9 is equal to the concatenation of the sequence *reset*, $set(s)$, a, and *status*, repeated for every s in S and every a in I. The length of the resulting checking sequence is exactly $4 \cdot p \cdot n$ where $p = |I|$ is the number of inputs and $n = |S|$ is the number of states.

The main weakness of Algorithm 9 is that it needs the *set* message, which may be not available. To avoid the use of *set* and to possibly shorten the test

suite, we can build a sequence that traverses the machine and visits every state and every transition at least once without restarting from the initial state after every test and without using a *set* message. Such sequence is called *transition tour*. Formally

Definition 4.2. An input sequence $x = a_1 a_2 \dots a_n$ that takes the machine to the states $s_1, s_2, \dots, s_n$ such that for all $s \in S$ there exists j such that $s_j = s$ (x visits every state) and such that for all $b \in I$ and for all $s \in S$ there exists j such that $a_j = b$ and $s_j = s$ (every input b is applied to each s), is a **transition tour**.

In the next section we present some basic techniques for the generation and use of transition tours for conformance testing that does not assume anymore the existence of a *set* message, i.e. relaxing Assumption 9.

4.3 State and Transition Coverage

By applying the *transition tour* (TT) *method,* the checking sequence is obtained from a transition tour, by adding a *status* message, that we assume reliable, after every input. Formally if $x = a_1 a_2 \dots a_n$ is a transition tour, the input sequence is equal to $a_1\,status\,a_2 status \dots a_n\,status$. This is a checking sequence. Indeed, since every state is checked with its *status* message after every transition, this input sequence can discover any transfer fault. Furthermore, every output fault is uncovered because every transition is tested (by applying the input a_j) and its output observed explicitly.

At best this checking sequence starts with a reset and exercises every transition exactly once followed by a *status* message. The length of such sequence is always greater than $2 \cdot p \cdot n$. The shortest transition tour that visits each transition exactly once is called **Euler tour**. Since we assume that the machine is strongly connected (Assumption 3), a sufficient condition for the existence of an Euler tour is that the FSM is *symmetric*, i.e. every state is the start state and end state of the same number of transitions. In this case, an Euler tour can be found in a time that is linear in the number of transitions, pn [EJ73]. This is a classical result of the graph theory and algorithms for generating an Euler tour can be found in an introductory book about graphs [LP81] and in the Chapter 9 of [Hol91]. In non symmetric FSMs searching the shortest tour is another classical direct graph problem, known as the *Chinese Postman Problem*, that can be solved in polynomial time. It was originally introduced by a Chinese mathematician [Kwa62] and there exist several classical solutions [EJ73] for it.

Example. For the machine in Fig. 4.1 the following checking sequence is obtained from the transition tour *bababa* (that is, more precisely, an Euler tour).

checking sequence	*b*	*status*	*a*	*status*	*b*	*status*	*a*	*status*	*b*	*status*	*a*	*status*
start state	1	2	2	2	2	3	3	3	3	1	1	1
output	1	*2*	1	*2*	1	*3*	0	*3*	0	*1*	0	*1*
end state	2	2	2	2	3	3	3	3	1	1	1	1

This checking sequence is able to detect the faults in the machines shown in Figure 4.2. The fault in M_{I1} is detected by the application of a b message in state s_3, while the faults in M_{I2} are immediately detected by the first *status* message.

If the *status* message is unreliable, we have to test it too. Assume that the *status* message may produce a wrong output or it may erroneously change the machine state. Both faults are detected by applying a *status* message twice in every state, the first one to test that the previous message has taken the machine to the correct state and to check that *status* message produces the correct output and the second one to verify that the first *status* message did not change the state of the machine.

Note the TT method was originally proposed without using a *status* message [NT81]. In this case the TT methods achieves only *transition coverage*. A test that visits only all the states, but not necessarily all the transitions, is often called *state tour* (ST) *method* [SMIM89] and achieves only *state coverage*. The coverage of every transition and the use of a *status* message are needed to discover every fault. Indeed, simply generating input sequences covering all the edges of M_S and test whether M_I produces the same outputs is not enough, as demonstrated by the following example.

Example. Consider the machines in Figure 4.2 as alleged equivalent machines to M_S in Figure 4.1. The sequence *abababa* is an Euler tour. Applying this tour to M_{I1} without using the *status* message, we would discover the output fault of the transition from s_3 to s_1: M_{I1} produces the output sequence 011101 instead of 011100. However, if we apply this Euler tour to M_{I2}, we do not discover the faults: M_{I2} produces the output sequence 011100, identical to the expected output sequence produced by M_S. However M_{I2} is a faulty implementation of M_S as demonstrated by another tour, namely *bababa*. This demonstrates that transition coverage is not capable to detect all the faults, in particular, to detect transfer faults.

Unfortunately, a *status* message is seldom available. In the next section we learn how not to rely on a *status* message to determine the current state during a test.

4.4 Using Separating Sequences Instead of Status Messages

We assume now that the machines have no *status* message (but they still have a *reset* message), and we wish to test whether M_S is equivalent to M_I only observing the external behavior. In the following we present some methods that can be unified as proposed by Lee and Yannakakis [LY96]. All these methods share the same technique to identify a state: they replace the use of the *status* message with several kinds of sequences that we can generally call *separating sequences* [LY96] and that are able to identify the state to which they have been applied. Remember that, since M_S is minimal, it does not contain two equivalent states, i.e. for every pair of states s_i, s_j there exists an input sequence x that we

call separating sequence and that distinguishes them because produces different outputs, i.e. $\lambda(s_i, x) \neq \lambda(s_j, x)$. Section 1.3 presents a classical algorithm to compute a separating sequence for two states.

4.4.1 W Method

The W method [Cho78] uses a particular set of separating sequences that is called characterizing set and another set to visit each transition in the machine, that is called *transition cover set* or *P set* for short, and is defined as follows.

Definition 4.3. (**Transition Cover Set**) the transition cover set of M_S is a set P of input sequences such that for each state $s \in S$ and each input $a \in I$ there exists an input sequence $x \in P$ starting from the initial state s_1 and ending with the transition that applies a to state s. Formally for all $s \in S$ and for all $a \in I$ there exist an input sequence $x \in P$ and an input sequence $y \in P$ such that $x = y.a$ and $\delta(s_1, y) = s$.

A P set forces the machine to perform every transition and then stop. A P set can be built by using a normal breadth-first visit of the transition diagram of the machine M_S. Note that a P set is closed under the operation of selecting a prefix: if x belongs to P, then any prefix of x is in P too. One way of constructing P [Cho78] is to build first a testing tree T of M_S as explained in Algorithm 10 and then to take the input sequences obtained from all the *partial paths* of T. A partial path of T is a sequence of consecutive branches, starting from the root of T and ending in a terminal or non terminal node. Since every branch in T is labeled by an input symbol, the input sequence obtained from a partial path q is the sequence of input symbols on q. The empty input sequence ε is considered to be part of any P set. Note that Algorithm 10 terminates because the number of states is finite.

Algorithm 10 Building a test tree

(1) Label the root of the tree T with s_1, the initial state of M_S. This is the level 1 of T
(2) Suppose that we have already built the tree T up to the level k. Now we build the $k+1$st level.
   (a) consider every node t at level k from left to right
   (b) if the node t is equal to another node in T at level j, with $j \leqslant k$, then t is terminated and must be considered a leaf of T
   (c) otherwise, let s_i be the label of the node t. For every input x, if the machine M_S goes from state s_i to state s_j, we attach to t a branch with label x and a successor node with label s_j

Example. A test tree for M_S of Fig. 4.1 is shown in Fig. 4.3. From this test tree we obtain $P = \{\varepsilon, a, b, ba, bb, bba, bbb\}$.

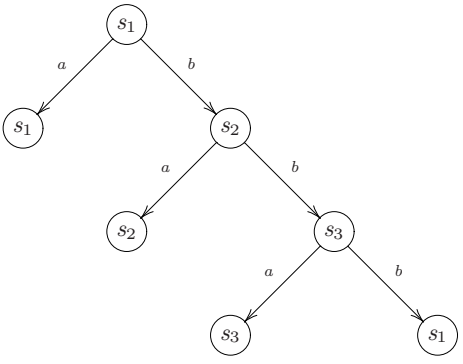


Fig. 4.3. A test tree for M_S of Figure 4.1

The W method uses a P set to test every transition of M_I and uses another set, called *characterizing set* of M_S or W set, instead of the *status* message, to verify that the end state of each transition is the one expected. A characterizing set is defined as follows.

Definition 4.4. (**Characterizing Set**) a characterizing set of M_S is a set W of input sequences such that for every pair of distinct states s and t in S, there exists an input sequence x in W such that $\lambda(s, x) \neq \lambda(t, x)$

The characterizing set is briefly called W set or sometimes *separating set.* The input sequences in the W set are also called *separating sequences.* A W set exists for every machine that is minimal (Assumption 1). The choice of a W set is not unique and the fewer are the elements in the W set the longer are the separating sequences in the W set. An algorithm for building a W set follows.

Partition the set of states S into blocks B_i with $i = 1, \ldots, r$. Initially W is $\emptyset$, $B_1 = S$ and $r = 1$. Until every B_i is a singleton, take two distinct states s and t in a B_i (that contains at least two states) and build their separating sequence x by means of the algorithm presented in Section 1.3. Add x to W and partition the states s_{ik} in every B_j into smaller blocks $B_{j1}, \ldots, B_{jh}$ based on their different output $\lambda(s_{ik}, x)$. Repeat the process until each B_i becomes a singleton and r becomes n. For every pair of states s_i and s_j, the resulting W set contains an input sequence x that separates s_i from s_j. Note that there are no more than $n - 1$ partition and therefore W set has no more than $n - 1$ separating sequences.

The W method consists in using the entire W set instead of the *status* message to test that the end state of each transition is the one expected. Since W may contain several sequences, we have to visit the same end state of every transition several times to apply all the separating sequences in a W set and for this goal we can use a *reset* message and the sequences in a P set. The set of input sequences is simply obtained concatenating every input sequence in a P set with every input sequences in a W set and apply them in order after a reset message to take the machine back to the initial state. In this way each input sequence p_{ij}

is the concatenation of the i-th sequence of a P set (to test the i-th transition) with the j-th sequence of a W set, with an initial *reset* input.

Formally, given two sets of input sequences X and Y, we denote with $X.Y$ the set of input sequences obtained concatenating all the input sequences of X with all the input sequences of Y. The set of input sequences produced by the W method is equal to $\{reset\}.P.W$.

If we do not observe any fault, the implementation is proved to be correct [Cho78]. Indeed, any output fault is detected by the application of a sequence of P, while any transfer fault is detected by the application of W.

Example. For the machine in Fig. 4.1 a characterizing set W is $\{a,b\}$ (equal to the input set I). In fact we have:

For state s_1, transitions $a/0$ $b/1$

For state s_2, transitions $a/1$ $b/1$

For state s_3, transitions $a/0$ $b/0$

a distinguishes s_1 from s_2 and s_3 from s_2. b distinguishes $s_1$from s_3.

$P = \{\varepsilon,\ a,\ b,\ ba,\ bb,\ bba,\ bbb\}$

The set of test sequences $P.W$ is reported in the following table, where we indicate with r the *reset* message.

P	ε		a		b		ba		bb		bba		bbb	
r.P.W	ra	rb	raa	rab	rba	rbb	$rbaa$	$rbab$	rbb	$rbbb$	$rbbaa$	$rbbab$	$rbbba$	$rbbbb$
trans. to test	$s_1 \xrightarrow{\varepsilon}$		$s_1 \xrightarrow{a/0} s_1$		$s_1 \xrightarrow{b/1} s_2$		$s_2 \xrightarrow{a/1} s_2$		$s_2 \xrightarrow{b/1} s_3$		$s_3 \xrightarrow{a/0} s_3$		$s_3 \xrightarrow{b/0} s_1$	
output	0	1	00	11	11	11	111	111	110	110	1100	1100	1100	1101

The total length of the checking sequence is 52.

The fault in machine M_{S1} of Figure 4.2 is detected by the input sequence $rbbb$, while the transfer faults in machine M_{S2} are detected by the pair of input sequences that tests the end state of the transition: for example the fact the transition from s_1 with input a erroneously moves the machine to s_2 is detected by the input sequences raa and rab.

4.4.2 Wp Method

The partial W or Wp method [FvBK+91] has the main advantage of reducing the length of the test suite with respect to the W method. This is the first method we present that splits the conformance test in two phases. During the first phase we test that every state defined in M_S also exists in M_I, while during the second phase we check that all the transitions (not already checked during the first phase) are correctly implemented.

For the first phase, the Wp method uses a *state cover set* instead of a transition cover set. The state cover set or Q *set*, for short, covers only the states, is smaller than the transition cover set, and it is defined as follows.

Definition 4.5. (**State Cover Set**) the state cover set is a set Q of input sequences such that for each $s \in S$, there exists an input sequence $x \in Q$ that takes the machine to s, i.e. $\delta(s_1, x) = s$

Using a Q set we can take the machine to every state. A Q set can be built using a breadth first visit of the transition graph of M_S. For the second phase, the Wp method uses an *identification set* W_i for state s_i instead of a unique characterizing set W for all the states. W_i is a subset of W and is defined as follows.

Definition 4.6. (**Identification Set**) an identification set of state s_i is a set W_i of input sequences such that for each state s_j in S (with $i \neq j$) there exists an input sequence x of W_i such that $\lambda(s_i, x) \neq \lambda(s_j, x)$ and no subset of W_i has this property.

Note that the union of all the identification sets W_i is a characterizing set W.

Wp Method Phase 1 The input sequences for phase one consist in the concatenation of a Q set with a characterizing set (W set) after a reset. Formally, the set of input sequences is $\{reset\}.Q.W$. In this way every state is checked in the implementation with a W set.

We say that a state q_i in M_I is *similar* to state s_i if it produces the same outputs on all the sequences in a W set. A state q_i in M_I can be similar to at most one state of M_S, because if we suppose that q_i is similar to states s_i and s_j then s_i and s_j produce the same output for each sequence in a W set, that is against Definition 4.4. We say that the machine M_I is *similar* to M_S if for every state s_i of M_S, the machine M_I has a state similar to s_i. If M_I is similar, since it has n states (Assumption 6), then there exists a one-to-one correspondence between similar states of M_S and M_I.

If the input sequences do not uncover any fault during the first phase, we can conclude that every state in M_S has a similar state in the implementation and we can say that M_I is similar to M_S. Note that is not sufficient to verify that it is also equivalent. The equivalence proof is obtained by the next phase.

Wp Method Phase 2 The second phase tests all the transitions. To this aim, Wp method uses the identification sets. For every transition from state s_j to state s_i on input a, we apply a sequence x (after a *reset* message) that takes the machine to the state s_j along transitions already verified, then we apply the input a, which takes the machine to s_i and we apply one identification sequence of W_i. We repeat this test for every identification sequence in W_i and if these tests do not uncover any fault, we have verified that the transition in the machine M_I from a state that is similar to s_j on input a produces the right output (there is no output fault) and goes to a state that is similar to s_i (there is no transfer fault). By applying these tests to every transition, we can verify that M_I conforms to its specification.

The set of input sequences that covers every transition (and that is closed under the operation of selecting a prefix) is a P set. Therefore, the input sequences of phase 2 consist of the sequences of a P set ending in state s_i that are not contained in the Q set used during phase 1, concatenated with all the sequences contained in the identification set W_i. Formally if $R = P - Q$ and x_i in R ends in s_i, the set of sequences applied during the second phase is $\{reset\}.R.W_i$.

A complete formal proof of correctness for the Wp method is given in the paper that introduced the Wp method [FvBK+91].

Example. The machine in Fig. 4.1 has the following state cover set $Q = \{\varepsilon, b, bb\}$.

During the first phase we generate the following test sequences:

state to test	1		2		3	
Q	ε		b		bb	
r.Q.W	ra	rb	rba	rbb	$rbba$	$rbbb$
output	0	1	11	11	110	110

During the second phase, we first compute the identification sets.
$W_1 = \{a,b\}$ all the sequences in W are needed to identify s_1
$W_2 = \{a\}$ distinguishes the state s_2 from all other states
$W_3 = \{b\}$ distinguishes the state s_3 from all other states
$R = P - Q = \{\ a,\ ba,\ bba,\ bbb\}$

R	a		ba	bba	bbb	
start state	1		2	3	1	1
r.R.Wi	raa	rab	$rbaa$	$rbbab$	$rbbba$	$rbbbb$
output	00	01	111	1100	1100	1101
end state	1	2	2	1	1	2

The total length of the checking sequence is 44 (note that Wp method yields a smaller test suite than the W method).

The output fault in machine M_{I1} of Figure 4.2 is detected during the first phase again by the input sequence $rbbb$. Some transfer faults in machine M_{I2} are detected during the first phase, while others, like the transfer fault from state s_3 with input a is detected only by the input sequences $rbbab$ during phase 2.

4.4.3 UIO Methods

If a W_i set contains only one sequence, this sequence is called *state signature* [YL91] or *unique input/output* (UIO) sequence [SD88] , that is unique for the state s_i. UIO sequences are extensively studied in Chapter 3 for state verification. Remember that applying a UIO sequence we can distinguish state s_i from any other state, because the output produced applying a UIO sequence is specific to s_i. In this way a UIO sequence can determine the state of a machine before its application. A UIO sequence has the opposite role of a homing sequence or a synchronizing sequence, presented in Chapter 1: it identifies the first state in the sequence instead of the last one. Note that not every state of a FSM has UIOs and algorithms to check if a state has a UIO sequence and to derive UIOs provided that they exist, can be found in Chapter 3. If an UIO sequence exists for every state s_i, then UIOs can be used to identify each state in the machine; in

this case the UIO sequence acts as *status* message, except it moves the machine to another state.

The original UIO method [SD88] builds a set of input sequences that visit every transition from s_i to s_j by applying a transition cover set P and then check the end state s_j by applying its UIO sequence. In this case the UIO sequence is used instead of a *status* message.

Although used in practice, the UIO method does not guarantee to discover every fault in the implementation [VCI90] because the uniqueness of the UIO sequences may not hold in a faulty implementation. A faulty implementation may contain a state s' that has the same UIO as another state s (because of some faults) and a faulty transition ending in s' instead of s may be tested as correct. Note that for this reason the Wp method uses the W_i sets only in the second phase, while in the first phase it applies the complete W instead.

A modified version of the UIO method, called UIOv, generates correct checking sequences [VCI90]. The UIOv method builds the test suite in three phases:

(1) *Uv process*: for every state s in M_S, apply an input sequence x that begins with a *reset* and reaches s and then apply the UIO sequence of s. To reach each state use a Q set. The set of input sequences consist of the concatenation of Q with the UIO sequence of the final state of the sequence in Q with an initial *reset*.
(2) $\neg Uv$ *process*: visit every state s and apply the input part of the UIO sequences of all other states and check that the obtained output differs from the output part of the UIO sequence applied. Skip UIO sequences that have the input part equal to a prefix α of the input part of the UIO sequence of s. Indeed, in this case, we have already applied α during the Uv process and we know that the output differs, because two states cannot have the same input and output part of their UIO sequences. At the end of Uv and $\neg Uv$ process we have verified that M_I is similar to M_S.
(3) *Transition test phase*: check that every transition not already verified in 1 and 2 produces the right output and ends in the right state by applying its UIO sequence.

Note that the UIOv method can be considered as a special case of Wp method, where the W set is the union of all the UIO sequences and phase 1 of the Wp method includes both Uv process and $\neg Uv$ process and phase 2 is the transition test phase.

Example. For the machine in Fig. 4.1 the UIO sequences are:
$UIO_1 = ab/01$ distinguishes the state s_1 from all other states
$UIO_2 = a/1$ distinguishes the state s_2 from all other states
$UIO_3 = b/0$ distinguishes the state s_3 from all other states
1. Uv process

Q	ε	b	bb
state to test	1	2	3
r.Q.UIO	rab	rba	$rbbb$
output	01	11	110

2. $\neg Uv$ process

state to test	1	2		3	
r.Q.¬UIO	rb	$rbab$	rbb	$rbbab$	$rbba$
output	1	111	11	1100	110

3. Transition test phase:

transition to test	$s_1 \xrightarrow{a/0} s_1$	$s_2 \xrightarrow{a/1} s_2$	$s_3 \xrightarrow{b/0} s_1$	$s_3 \xrightarrow{a/0} s_3$
input sequence	$raab$	$rbaa$	$rbbbab$	$rbbab$
output	001	111	11001	1100

The output fault of machine M_{I1} of Figure 4.2 is detected during the Uv process again by the input sequence $rbbb$. Some transfer faults in machine M_{I2} are detected during the first phases, while others, like the transfer fault from state s_3 with input a is detected only by the input sequences $rbbab$ during the transition test phase.

4.4.4 Distinguishing Sequence Method

In case we can find one sequence that can be used as UIO sequence for every state, we call such sequence **distinguishing sequence** (DS) (defined and extensively studied in Chapter 2). In this situation we can apply the DS method using the *reset* message [SL89]. Note that this DS method can be viewed as a particular case of the W method when the characterizing set W contains only a preset distinguishing sequence x. The test sequences are simply obtained combining a P set with x.

Example. For the machine in Fig. 4.1 we can take the sequence $x = ab$ as a preset distinguishing sequence. In fact

$\lambda_{Ms}(s_1, x) = 01$
$\lambda_{Ms}(s_2, x) = 11$
$\lambda_{Ms}(s_3, x) = 00$

P	ε	a	b	ba	bb	bba	bbb
r.P.x	rab	$raab$	$rbab$	$rbaab$	$rbbab$	$rbbaab$	$rbbbab$
trans. to test	$s_1 \xrightarrow{\varepsilon}$	$s_1 \xrightarrow{a/0} s_1$	$s_1 \xrightarrow{b/1} s_2$	$s_2 \xrightarrow{a/1} s_2$	$s_2 \xrightarrow{b/1} s_3$	$s_3 \xrightarrow{a/0} s_3$	$s_3 \xrightarrow{b/0} s_1$
output	01	001	111	1111	1100	11000	11001

4.4.5 Cost and Length

All the methods presented in Section 4.4 share the same considerations about the length of the checking sequence and the cost of producing it. For the W method, the cost to compute a W set is $\mathcal{O}(pn^2)$ and a W set contains no more than $n-1$ sequences of length no more than n. The cost to build the tree T set using the Algorithm 10 is $\mathcal{O}(pn)$ and its maximum level is n. The generation of a P set, by visiting T, takes time $\mathcal{O}(pn^2)$ and produces up to pn sequences with the maximum length n. Since we have to concatenate each transition from in a P set with each transition in a W set, we obtain up to pn^2 sequences of length $n+n$, for a total length of $\mathcal{O}(pn^3)$ and a total cost of $\mathcal{O}(pn^3)$. The Wp method has the same total cost $\mathcal{O}(pn^3)$ and same length $\mathcal{O}(pn^3)$. Experimental results [FvBK+91] show that checking sequences produced by the Wp method are generally shorter than the checking sequences produced by the W method.

The UIO method and the method using a preset distinguishing sequence are more expensive, because determining if a state has UIO sequences or a preset distinguishing sequence was proved to be PSPACE hard (as shown in Sections 3.2 and 2.3). Note that in practice UIO sequences are more common than distinguishing sequences (as explained in Chapter 3). However, as shown in Section 2.4, finding an *adaptive* distinguishing sequences has cost $\mathcal{O}(n^2)$ and adaptive distinguishing sequences have maximum length n^2. We can modify the method of Section 4.4.4 by using adaptive distinguishing sequences instead of preset distinguishing sequences. Because there are pn transitions, the total length for the checking sequence is again pn^3.

There are specification machines with a *reset* message, that require checking sequences of length $\Omega(pn^3)$ [Vas73].

4.5 Using Distinguishing Sequences Without Reset

If the machine M_S has no *reset* message, the reset message can be substituted by a *homing sequence,* already introduced in Section 1.1.3. However this can lead to very long test suites and it is seldom used in practice.

On the other hand, since methods like UIO (Section 4.4.3) and DS (Section 4.4.4) require the application of a single input sequence for each state, instead of a set of separating sequences as in W and Wp methods, they can be easily optimized for the use without *reset*, using instead a unique checking sequence similar to a transition tour. These methods can substitute the transition tour method when a *status* message is not available and they are often used in practice. The optimized version of the UIO method without *reset* is presented by Aho et al. [ADLU91], while the optimized version of the DS method [Hen64] without *reset* is presented in this section. Some tools presented in Chapter 14 are based on these methods.

To visit the next state to be verified we can use transfer sequences, that are defined as follows.

Definition 4.7. (**Transfer Sequence**) A transfer sequence $\tau(s_i, s_j)$ is a sequence that takes the machine from state s_i to s_j

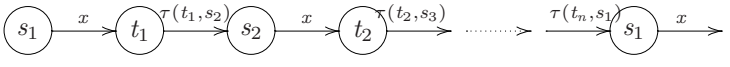
Such a transfer sequence exists for each pair of states, since M_S is strongly connected (by Assumption 3) and cannot be longer than $n-1$. Moreover, if the machine has a distinguishing sequence x, this sequence can be used as unreliable *status* message because it gives a different output for each state. It is like a *status* message, except that it moves the machine to another state when applied.

The method presented in this section has, as many methods presented in the previous section, two phases. It first builds an input sequence that visits each state using transfer sequences instead of reset and then applies its distinguishing sequence to test whether M_I is similar to M_S. It then builds an input sequence to test each transition to guarantee that M_I conforms to M_S.

Phase 1 Let t_i be the final state when applying the distinguishing sequence x to the machine from state s_i, i.e. $t_i = \delta(s_i, x)$ and $\tau(t_i, s_{i+1})$ the transfer sequence from t_i to s_{i+1}. For the machine in the initial state s_1, the following input sequence checks the response to the distinguishing sequence in each state.

$$x\,\tau(t_1, s_2)\,x\,\tau(t_2, s_3)\,x\,\ldots\,\tau(t_n, s_1)\,x \tag{4.1}$$

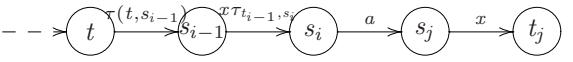
This sequence can be depicted as follows.



Starting from s_1 the first application of the distinguishing sequence x tests s_1 and takes the machine to t_1, then the transfer sequence τ_1 takes the machine to s_2 and the second application of x tests this state and so on till the end of of the state tour. At the end, if we observe the expected outputs, we have proved that every state of M_S has a similar state in M_I, since we have tested that every state in M_I correctly responds to its distinguishing sequence.

Phase 2 In the second phase, we want to test every transition. To test a transition from s_i to s_j with input a we can take the machine to s_i, apply a, observe the output, and verify that the machine is in s_j by applying x. Assuming that the machine is in state t, to take the machine to s_i we cannot use $\tau(t, s_i)$ because faults may alter the final state of $\tau(t, s_i)$. Therefore, we cannot go directly from t to s_i. On the other hand, we have already verified by (4.1) that $x\tau(t_{i-1}, s_i)$ takes the machine from s_{i-1} to s_i. We can build an input sequence that takes the machine to s_{i-1} , verifies that the machine is in s_{i-1} applying x and moves to s_i using $\tau(t_{i-1}, s_i)$, then applies a, observes the right output, and verifies that the end state is s_j by applying again the distinguishing sequence x:

$$\tau(t, s_{i-1})x\tau(t_{i-1}, s_i)ax \tag{4.2}$$



Therefore, the sequence (4.2) tests the transition with input a from state s_i to s_j and moves the machine to t_j. We repeat the same process for each transition to obtain a complete checking sequence.

Example. A distinguishing sequence for the machine in Fig. 4.1 is $x = ab$ and the corresponding responses from state s_1, s_2, and s_3 are: 01 11, and 00 respectively. The distinguishing sequence, when applied in states s_1, s_2, and s_3 takes the machine respectively to $t_1 = s_2$, $t_2 = s_3$ and $t_3 = s_1$. the transfer sequences are $\tau(t_1, s_2) = \tau(t_2, s_3) = \tau(t_3, s_1) = \varepsilon$.

The sequence (4.1) becomes

	x	$\tau(t_1, s_2)$ x	$\tau(t_2, s_3)$ x	$\tau(t_3, s_1)$ x
checking sequence	ab	ab	ab	ab
output	01	11	00	01

This input sequence ends in state $t_1 = s_2$

The input sequences (4.2) can be concatenated to obtain:

trans. to test	$s_3 \xrightarrow{b/0} s_1$	$s_2 \xrightarrow{a/1} s_2$	$s_3 \xrightarrow{a/0} s_3$	$s_1 \xrightarrow{a/0} s_1$	$s_2 \xrightarrow{b/1} s_3$	$s_1 \xrightarrow{b/1} s_2$
	$\tau(t_1, s_3)bx$	$\tau(t_1, s_2)ax$	$\tau(t_2, s_3)ax$	$\tau(t_3, s_2)ax$	$\tau(t_1, s_2)bx$	$\tau(t_3, s_1)bx$
input sequence	$bbab$	aab	aab	aab	bab	bab
end state	2	3	1	2	1	3
output	1001	111	000	001	100	111

The total length of the checking sequence is 27.

Note that the first input sequence is not able to find the faults in machine M_{I2} of Fig. 4.2, since M_{I2} when we apply the input sequence $ababab$ produces the expected output 01110001. Only during the second phase the faults are detected.

Adaptive DS Instead of using a unique preset distinguishing sequence for all the states, we can use an adaptive distinguishing sequence as explained in the following. An adaptive distinguishing sequence (ADS) is a decision tree that specifies how to choose the next input adaptively based on the observed output to identify the initial state. Adaptive distinguishing sequences are studied in Section 2.4. In that Chapter, the reader can find the definition (2.12), an algorithm to check the existence of an ADS and to build an ADS if it exists.

Example. An adaptive distinguishing sequence for the machine in Fig. 4.1 is depicted in Figure 4.4. We apply the input a and if we observe the output 1 we know that the machine was in the state s_2. If we observe the output 0, we have to apply b and if we observe the output 1 the machine was in s_1 otherwise we observe 0 and the machine was in s_3.

Using adaptive distinguishing sequence for our example, we obtain $x_1 = ab$, $x_2 = a$, $x_3 = b$, and $\tau = \varepsilon$ and the sequence (4.1) becomes

	x_1	$\tau(t_1, s_2)$	x_2	$\tau(t_2, s_3)$	x_3	$\tau(t_3, s_1)$	x_1
input sequence	ab		a	b	ab		ab

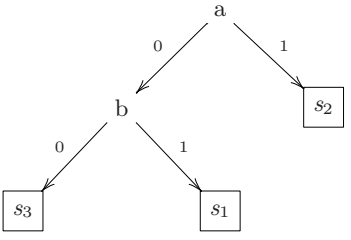


Fig. 4.4. Adaptive distinguishing sequence of machine in Fig. 4.1

Length and Cost An adaptive distinguishing sequence has length $\mathcal{O}(n^2)$, and a transfer sequence cannot be longer than n . The sequence (4.1) is long $\mathcal{O}(n^3)$. Because there are pn transitions, and every sequence (4.2) has length $\mathcal{O}(n^2)$, the cost is again $\mathcal{O}(pn^3)$ to find the complete checking sequence. Therefore, all the methods presented in Section 4.4 and in this section, have the same cost. The advance of the method presented in this section, is that it does not need a *reset* message. A comparison among methods from a practical point of view is presented in Section 4.8.

Minimizing the Sequence Length Note that there exist several techniques to shorten the length of the checking sequence obtained by applying the distinguishing sequence method [UWZ97], but still resulting checking sequences have length $\mathcal{O}(pn^3)$. The problem of finding the shortest transition tour covering all the transitions and then applying an extra sequence, that is a UIO or a DS sequence in this case, to their end state is called the *Rural Chinese Postman Problem* [ADLU91].

4.6 Using Identifying Sequences Instead of Distinguishing Sequences

Not every finite state machine has distinguishing sequences (as shown in Section 2.1). In case the machine has no *reset* message, no *status* message, no UIO sequences, and no distinguishing sequences, we cannot apply the methods proposed so far. We can still use the Assumption 1 and exploit the existence of separating sequences, that can distinguish a state from any other state in M_S. In this case, conformance testing is still possible [Hen64], although the resulting checking sequences may be exponentially long.

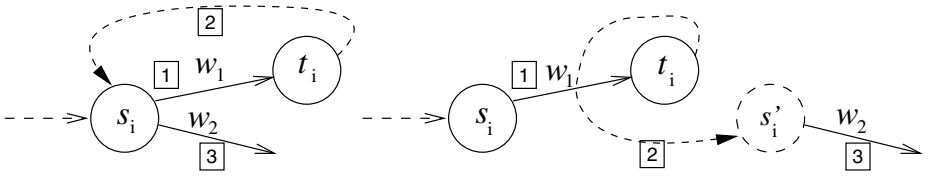


Fig. 4.5. Using two separating sequences to identify the state

As usual, we first check that M_I is similar to M_S. We display for each state s_i the responses to all the separating sequences in a characterizing set W (Definition 4.4). Suppose that W has two separating sequences w_1 and w_2. We want to apply the steps shown (in square boxes) in Figure 4.5 (a) : take M_I to s_i, apply w_1 (step 1), take the machine back again to s_i (step 2) and then apply w_2 (step 3). If we observe the right output, we can say that the machine M_I has a state q_i similar to s_i. We can start from $i = 1$ and proceed to verify all the states without using neither reset nor a distinguishing sequence. The problem is that we do not know how to bring the machine M_I back to s_i in a verifiable way, because in a faulty machine, as shown in Figure 4.5 (b), the transfer sequence $\tau(t_i, s_i)$ (step 2) may take the machine to another state s_i' where we could observe the expected output applying the w_2 sequence, without being able to verify that s_i' is s_i and without able to apply again w_1. We use now the Assumption 6 on page 90, namely that M_I has only n states. Let x be an input sequence and n be an integer, x^n is the concatenation n times of x.

Theorem 4.8. *Let s be a state of M_I, x be an input sequence, o the expected output sequence produced applying x to s, i.e. $o = \lambda(s, x)$, τ a transfer sequence from $t = \delta(s, x)$ back to s, and o' the expected output produced applying τ to t. By applying the input sequence $(x\,\tau)^n$ to state s in M_I, if we observe the output sequence $(o\,o')^n$, then the machine ends in a state where applying again x we observe the same output o.*

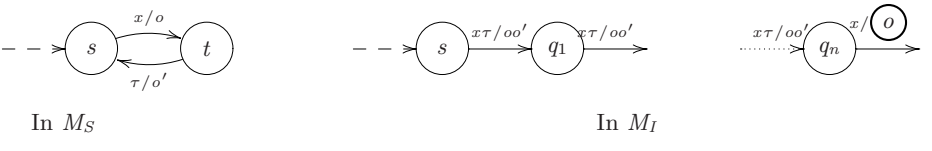


Fig. 4.6. Applying n times x and τ

Proof. The scenario described in theorem is shown in Figure 4.6. Suppose that M_I is initially in state s. Applying $x\,\tau$ the machine should come back to s. However, due to some faults, the machine M_I may go to another state q_1 even if

the output we observe is the one expected, i.e. $o\ o'$. Assume that applying n times $x\,\tau$, we observe every time the same output $o\ o'$. Let q_r be the state of M_I after the application of $(x\,\tau)^r$. Note that even if the n applications of $x\,\tau$ produce n times the same correct output $o\ o'$, we are not sure that $s, q_1, \ldots, q_n$ are the same state yet. However the $n{+}1$ states $s, q_1, \ldots, q_n$ cannot be all distinct, because M_I has n states. Hence q_n is equal to some q_r with $r < n$ and, therefore, it would produce the same output o if we apply x.

Example. Consider the machine in Figure 4.1 and take any alleged implementation M_I. Apply the input a (in this case $\tau = \varepsilon$) to the initial state s_1 of M_I and check that the output is 0. We are not sure that M_I is now in state s_1 as well. We can apply again a and observe the output 0 and so on. When we have applied aaa and observed the output 000, M_I may have traversed states s_1, q_1,q_2, and the final state q_3. Because M_I has only 3 states, q_3 is equal to one of s_1, q_1, or q_2 and we are sure that if we applied again a we would observe 0.

We use Theorem 4.8 as follows. Assume that M_S has the characterizing set $W = \{w_1, w_2\}$ and let s_i be the state we are going to verify. Let τ be the transfer sequence that takes M_S back to s_i from $t_i = \delta(w_1, s_i)$. We first apply $(w_1\tau)^n$ to s_i. If we observe a wrong output we have proved that M_I does not conform to M_S. Otherwise we can apply theorem with $x = w_1$ and we are sure that M_I ends in a state that would produce the same output as if we applied w_1. We apply w_2 instead. If we observe the specified output we can conclude that s_i has a similar state in M_I.

We can generalize this method when the characterizing set W contains m separating sequences. Suppose that the characterizing set is $W = \{w_1, \ldots, w_m\}$. Let τ_j be the transfer sequence that takes the machine back to s after the application of w_j, i.e. $\tau_j = \tau(\delta(s, w_j), s)$. We can define inductively the sequences β_r as follows:

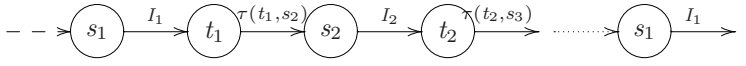
$$\begin{aligned} \beta_1 &= w_1 \\ \beta_r &= (\beta_{r-1}\tau_{r-1})^n w_r \end{aligned} \tag{4.3}$$

By induction, one can prove that applying β_{r-1} after applying $(\beta_{r-1}\tau_{r-1})^n$ would produce the same output. Considering how β_i are defined, this means that applying $w_1, \ldots, w_{r-1}$ would produce the same output . For this reason we apply w_r after $(\beta_{r-1}\tau_{r-1})^n$. Therefore, one can prove that β_m is an *identifying sequence* of s_i, in the following sense: if the implementation machine M_I applying β_m produces the same output as that produced by the specification machine starting from s_i, then M_I has a state that is similar to s_i and such state is the state right before the application of the last w_m (regardless of which state M_I started from). We indicate the identifying sequence for state s_i with I_i.

Once we have computed the identifying sequence for every state, we can apply a method similar to that explained in Section 4.5 to visit each state, verify its response to the identifying sequence, and then transfer to the next state. Let I_i be the identifying sequence of state s_i and $\tau(t_i, s_{i+1})$ the transfer sequence from

$t_i = \delta(s_i, I_i)$ to s_{i+1}, by applying the following input sequence we can verify that M_I is similar to M_S.

$$I_1\, \tau(t_1, s_2)\, I_2 \tau(t_2, s_3) \ldots I_1 \tag{4.4}$$



Once we have proved that M_I is similar to M_S we have to verify the transitions. To do this we can use any I_i as reliable reset. For example, we can take I_1 as reset to the state $t_1 = \delta_I(s_1, w_m)$ and use t_1 as the initial state to test every transition. Indeed, we are sure that if we do not observe any fault, I_1 takes the machine to t_1. If we want to reset the machine from the state s_k to t_1 we apply $\tau(s_k, s_1)I_1$ and even if $\tau(s_k, s_1)$ fails to take the machine to s_1, we are sure that I_1 will take it to t_1. Now we proceed as explained in Section 4.4. To test a transition from s_i to s_j we apply a pseudo reset I_1 to t_1, then a transfer sequence along tested transitions to s_i, then we apply the input, observe the output, and apply the identifying sequence I_j to check that the end state is s_j.

Example. Consider the machine M_S in Fig. 4.1. $W = \{a, b\}$.
For s_1, $\tau_1 = \varepsilon$, $I_1 = (w_1\tau)^3 w_2 = aaa\ b$
For s_1, $\tau_1 = \varepsilon$, $I_2 = (w_1\tau)^3 w_2 = aaa\ b$
For s_1, $\tau_1 = \varepsilon$, $I_3 = (w_1\tau)^3 w_2 = aaa\ b$
The sequence (4.4) becomes

	I_1	$\tau(t_1, s_2)$	I_2	$\tau(t_2, s_3)$	I_3	$\tau(t_3, s_1)$	I_1
input sequence	$aaab$	ε	$aaab$	ε	$aaab$	ε	$aaab$

Length and Cost The length of an identifying sequence grows exponentially with the number of separating sequences and with n the number of the states. Indeed, by equation 4.3, every β_i is n times longer than β_{i-1}, the identifying sequence I is equal to β_m and m is the number of separating sequences that can be up to n. The resulting checking sequence is exponentially long. The IS method can be optimized using a different separating family Z_i for every state s_i [LY96].

4.7 Additional States

The Assumption 6, that the implementation has the same number of states as the specification, may not hold in general. The problem of testing each edge in a finite state machine with arbitrary extra states, is similar to the classical problem of traversing an unknown graph, that is called the *universal traversal* problem [LY96].

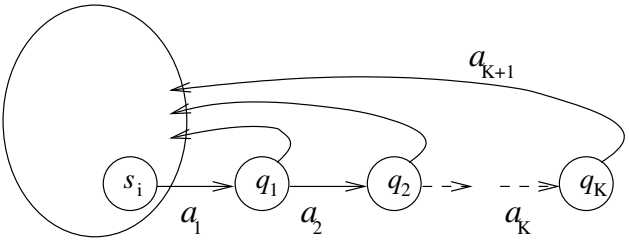


Fig. 4.7. A faulty machine M_I with K extra states

Assume that a faulty machine M_I, depicted in Figure 4.7, is identical to M_S except it has K extra states $q_1, \ldots, q_k$ and except for the transition from state s_i on input a_1 where M_I moves to the extra state q_1. Moreover M_I moves from q_1 to q_2 on input a_2, from q_2 to q_3 on input a_3, and so on. Assume the worst case, that only the transition from state q_k on input a_{K+1} has a wrong output or moves to a wrong next state. To be sure to test such transition, the input sequence applied to state s_i must include all possible input sequences of length K+1, and thus it must have length p^{K+1}. Such input sequence is also called *combination lock* because in order to unlock the machine, it must reach the state q_K and apply the input a_{K+1}. Vasilevski [Vas73] showed that also the lower bound on the input sequence is multiplied by p^K; i.e. it becomes $\Omega(p^{K+1}n^3)$ (discussed also in Section 5 of Chapter 19). Note that such considerations hold for every state machine M_I with K extra state: to test all the transitions we need to try all possible input combinations of length K+1 from all the states of M_I, and thus the input sequence must have length at least $p^{K+1}n$.

Using similar considerations, many methods we have presented can be easily extended to deal with implementations that may add a bounded number of states. This extension, however, causes an exponential growth of the length of the checking sequence.

In this section we present how the W method presented in Section 4.4.1 is extended to test an implementation machine with m states with $m > |S_S| = n$ [Cho78]. Let Q be a set of input sequences and k be an integer, Q^k is the concatenation k times of Q. Let W be a characterizing set (Definition 4.4). The W method in this case uses instead of a W set another set of sequences called the *distinguishing set* $\mathrm{Y} = (\varepsilon \cup I \cup I^2 \cup \ldots \cup I^{m-n}).W$. Therefore, we apply up to m-n inputs before applying W. The use of Y instead of W has the goal to discover states that may be added in M_I. Let P be a transition cover set. The resulting set of input sequences is equal to $\{reset\}.P.Y$. Each input sequence starts with a *reset*, then applies a sequence to test each transition, applies up to $m-n$ inputs, then applies a separating sequence of W. The set of input sequences $P.Y$ detects any output or transfer error as long as the implementation has no more than m states. The proof is given in [Cho78]. If $m = n$ then Y=W and we obtain the W method of Section 4.4.1.

Example. Consider the machine in Fig. 4.8 as faulty implementation of the specification machine M_S of Fig. 4.1 with one state more, namely s_4. The original sequences generated with the W method assuming that the machine has the same number of states are not capable to discover the fault. If we use the W method with $m = 4$, we generate for bbb in P, b in I and b in W the sequence $rbbbbb$ that is able to expose the fault.

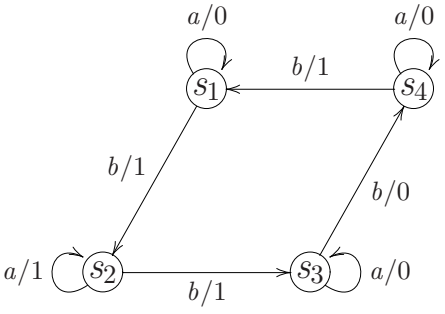


Fig. 4.8. A faulty implementation of machine M_S with 4 states

4.8 Summary

In this chapter we have presented several methods, which can uncover any fault in an implementation under different assumptions and producing checking sequences of different length and with different cost. We have initially supposed that all the assumptions of Section 4.2 hold, mainly that the machines are minimal, that the implementation does not add extra states, and that the machines have *reset*, *status* and *set* messages. Throughout the chapter we have presented the following methods which are capable to discover faults under a successively restricted subset of assumptions.

- The method of Section 4.3, the Transition Tour (TT) method, exploits all the assumptions, except the *set* message. It uses a *status* message to check that the implementation is in the correct state. The checking sequence has length and cost linear with pn. Without a *status* message this method does not guarantee the detection of transfer faults.
- If even a *status* message is not available, but the machine has still a *reset* message, one can use one of the methods proposed in Section 4.4, namely the W method, the Wp method, the unique input output (UIO) sequence method, the UIOv method, and the method using distinguishing sequences (DS) with *reset*. The DS method requires a distinguishing sequence, the UIO methods need UIOs, while W and Wp method are always applicable for minimized machines. The W, Wp, UIOv, and DS methods detect faults

of any kind, while the UIO method may miss some faults. The W, Wp, and DS method with an adaptive distinguishing sequence produce checking sequences of length $\mathcal{O}(pn^3)$ with cost $\mathcal{O}(pn^3)$. The others have greater cost.
- If even a *reset* message is not available, but a machine has a distinguishing sequence, the method presented in Section 4.5 uses transfer sequences instead of *reset*, produces checking sequences of length $\mathcal{O}(pn^3)$ and has cost $\mathcal{O}(pn^3)$ when used in conjunction with adaptive distinguishing sequences.
- If the machine has not even a distinguishing sequence nor UIOs, the identifying sequences (IS) method, presented in Section 4.6, still works. The IS method uses only the assumptions that the implementation does not add states and that the machines are minimized and therefore they have separating sequences. It produces exponentially long checking sequences.
- The problem of testing finite state machines with extra states is discussed more in general in Section 4.7, where the method originally presented by Chow [Cho78] is introduced.

It is of practical interest to compare the fault detection capability of the methods when the assumptions under which they should be applied, do not hold [SL88, ZC93]. Indeed, assumptions like the equal number of states for implementation may be not verifiable in practice. The assumption of the existence of a *reset* message is more meaningful, but empirical studies suggest to avoid the use of the methods using reset messages for the following reason. As shown in Section 4.7, faults in extra states are more likely to be discovered when using long input sequences. The use of a *reset* message may prevent the implementation to reach such extra states where the faults are present. For this reason methods like UIO or DS method without reset are better in practice than the UIOv method or the DS method with reset.

Although the study presented in this chapter is rather theoretical, we can draw some useful guidelines for practice testing for FSMs or for parts of models that behave like finite state machine and the reader should be aware that many ideas presented in this chapter are the basics for tools and case studies presented in Chapters 14 and 15. Such practical suggestions can improve the fault detection capability of the testing activity.

- Visiting each state in a FSM (like a *statement* coverage) using a ST method, should not be considered enough. One should at least visit every transition using a transition tour (TT) method, that can be considered as a *branch* coverage.
- Transition coverage should be used in conjunction of a *status* message to really check that the end state of every transition is the one expected. The presence of a *status* message in digital circuits is often required by the tester because it is of great help to uncover faults. If a *status* message may be not reliable, a double application of it helps to discover when it fails to reveal the correct state.
- If a *status* message is not available (very often in software black box testing), one should use some extra inputs to verify the states. Such inputs should be unique, like in Wp, UIO and DS.

- If one suspects that the implementation has more states than the implementation, he/she should prefer methods that produce long input sequences, like the DS and the IS method. However, only methods like the W method with extra states [Cho78], that add some extra inputs after visiting the transition and before checking the state identity, can guarantee to detect faults in this case.