Conformance Testing

Angelo Gargantini

Dipartimento di Matematica e Informatica
University of Catania - Italy
Email: gargantini@dmi.unict.it

1 Introduction

In this chapter we tackle the problem of conformance testing between finite state machines. The problem can be briefly described as follows [?]. 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* 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 familar with the definitions given in Chapter FSM(ref)chapter, 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*, *ouput symbols*, and *states*, respectively, $\delta : S \times I \rightarrow S$ is the *state transition function*, $\lambda : S \times I \rightarrow O$ is the *output function*. When the machine 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 rapresented by a state transition diagram as shown in Figure 1. We denote the number of states $n = |S|$ and the number of inputs $p = |I|$. An input sequence x is XXX TO ADD. CONCATENATION among set of input sequences TO ADD.

$$\begin{tikzcd}
& s_1 \arrow[loop above, "a/0"] \arrow[dl, "b/1"'] & \\
s_2 \arrow[loop left, "a/1"] \arrow[rr, "b/1"'] & & s_3 \arrow[loop right, "a/0"] \arrow[ul, "b/0"']
\end{tikzcd}$$

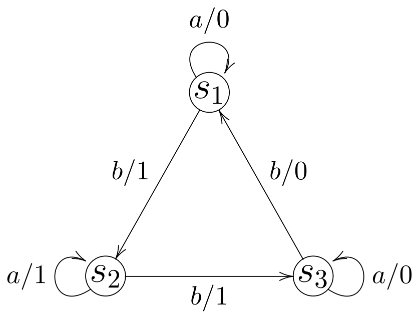


Fig. 1. Machine M_S (taken from [?])

The detection of faults in the implementation M_I can be performed by the following experiment. Generate a test suite (as defined in Chapter GL(ref)chapter) from the machine M_S. For every test case in the test suite separate the input sequence from the output sequence. Apply the 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 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 [**?**], i.e. test criteria that can discover any fault in the implementation (under suitable assumptions). Although this approach is rather theoretical, Section 8 presents the justifications for the theoretical assumptions and the practical implications of the presented results.

Conformance is formally defined as equivalence or isomorphism (as defined in Chapter FSM(ref)chapter): 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 an input sequence that we can apply to M_I to prove that is equivalent to its specification. Such input sequence is called checking sequence.

Definition 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 an 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. They differ also for the assumptions they make about the machines M_S and M_I.

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 admit some assumptions abut the machines we want to verify. Some assumptions are very natural and necessary, other are convenient and very used in practice but can be relaxed. These first four assumptions are necessary for every method we present in this chapter.

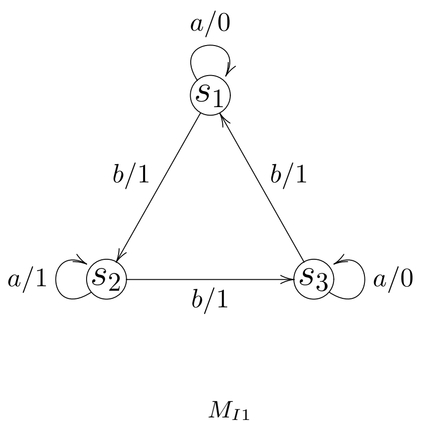
[1] Dijkstra, of course

1. M_S is *reduced* or *minimal*: the reason is that equivalent machines have the same I/O behavior, and we cannot distinguish them by observing the outputs. If M_S if it is not minimal, we can minimize it and obtain an equivalent reduced machine (an algorithm is presented in [?] and informally explained in [?] as well as in Chapter FSM(ref)chapter). In a minimal machine there are no equivalent states. Every minimal machine has *separating sequences* (defined in Section FSM1(ref)sec:algorithms). A separating sequence can distiguish two states in M_S.
2. M_S is *completely specified*: the state transition function δ and the output function λ 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 state 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 from the complete system vocabulary (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 in its initial state we can apply an homing sequence (presented in Section FSM1(ref)sec:intro-homing) 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 faults in M_I are of two kinds: *output faults*, i.e. a transition produces the wrong output, and *transfer faults*, i.e. the implementation goes to a wrong state. Although this assumption is very strong, we show in Section 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. Figure 2 shows two faulty implementations of the specification machine M_S given in Figure 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 only transfer faults: every transition produces the right output but moves the machine to a wrong final state.



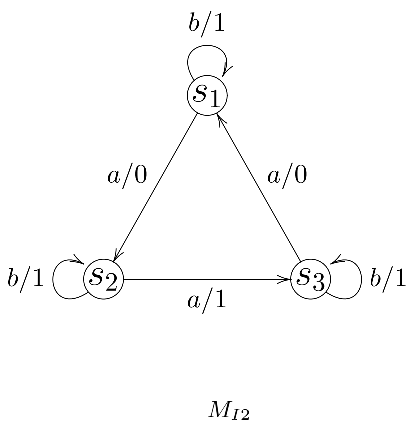


Fig. 2. Two faulty implementations of M_S

7. *reset* message: M_I and M_s have a particular input *reset* (or simply r) that from any state of the machine causes a transition which ends into the initial state and produces no output. Formally, forall $s \in S$, $\delta(s, reset) = s_1$ and $\lambda(s, reset) = -$. This assumption is relaxed starting from Section 5.
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 (we assume that they output the number of the state). The machines do not change state. Formally forall $s_i \in S$, $\lambda(s_i, status) = i$ and $\delta(s_i, status) = s_i$. This rather strong assumption is relaxed starting from Section 4.
9. *set* message: the input set I of M_s 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 forall $s, t \in S$, $\delta(s, set(t)) = t$ 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 1 (Chapter 9 of [**?**]).

Therefore, the resulting checking sequence x is obtained concatenating the sequence *reset*, *set*(s), *a*, and *status*, repeated for every $s$in S and every a in I.

This algorithm verifies that M_I correctly implements M_S and it is capable to uncover any output or transfer error. 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

Algorithm 1 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 input message $a$.
4. Verify that the output received conforms 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 the specification,
   i.e. it is equal to $\delta_S(s, a)$
```

state s_i after the application of $set(s_i)$. The first application 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* 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.

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.

This methods exploits 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 possibly without restarting from the initial state after every test. Such sequence is called *transition tour*. Formally

Definition 2. *An input sequence $x = a_1a_2 \ldots a_n$ that takes the machine to the states $s_1, s_2, \ldots, s_n$ such that $\forall s \in S\ \exists j\ s_j = s$ (x visits every state) and such that $\forall b \in I \forall s \in S\ \exists j\ a_j = b \wedge s_j = s$ (every input b is applied to each s), is called transition tour*

In the next Section we present some basic techniques for the generation and use of transition tours for conformance testing and we show their limits.

3 State and Transition Coverage

A simple conformance test can be performed by generating a transition tour and checking that every state in M_S is represented in M_I by a status message and to verify that each transition is correctly implemented. This methods is called transition tour (TT) method and it was originally proposed without using any status message [?]. At best this checking sequence starts with a reset and exercises every transition exactly once followed by a status message to check that the state is correct. The length of such sequence is always greater than $1 + 2 \cdot p \cdot n$. 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. An Euler tour can be found in linear time in $p = |I|$. This is a classical result of the graph theory and algorithms for generating an Euler tour can be found in any introductory book about graphs (for example, see Chapter 9 of [?]). 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 [?] and there exist several classical solutions [?] for it.

A conformance test using a transition tour achieves the so called *transition coverage*. A test that covers only all the states is often called state coverage or state tour method [?].

Example 1. For the machine in Fig. 1 the following checking sequence is a transition tour (it is, more precisely, an Euler tour).

input sequence	*b*	*status*	*a*	*status*	*b*	*status*	*a*	*status*	*b*	*status*	*a*	*status*
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

If the status message is unreliable and we have to test is too, we can apply 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 the second one to verify that the first status message did not change the state of the machine.

Without the status message, the transition coverage does not guarantee the detection of every fault. Indeed, simply generating tests covering all the edge of M_S and test whether M_I produces the same outputs is not enough, as demonstrated by the following example.

Example 2. Consider the machines in Figure 2 as alleged equivalent machines to M_S in Figure 1. The sequence *ababab* is an Euler tour. Applying this tour to M_{I1} we would discover the output fault of the transition from s_3 to s_1: M_{I1} produces the output sequence 011100 instead of 011101. However, if we apply this Euler tour to M_{I2}, we do not discover the faults: M_{I2} produces the output sequence 011101, 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 always detect transfer faults.

In the next section we learn how to not rely on a status message to determine the current state during a test.

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 generalized as proposed in [?]. All these methods share the same techinque to indentify a state: they replacing the use of the status message with several kinds of sequences that we can generally call *separating sequences* [?] and that are able to identify in some way 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 α that we call separating sequence and that distinguishes them because produces different outputs, i.e. $\lambda(s_i, \alpha) \neq \lambda(s_j, \alpha)$. Separating sequences are studied in Section FSM1(ref)sec:algorithms . Note that the subjects of state indetification and verification studied in Chapters FSM2(ref)chapter and FSM3(ref)chapter share with the methods presented in this section the same goal and several definitions.

4.1 W method

The W method [?] 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 3. *([transition cover set]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 $\forall s \in S, \forall a \in I, \exists x \in P$ such that $x = \alpha.a$ and $\delta(s_1, \alpha) = s$.*

The transition cover set is called P set. The P set forces the machine to perform every transition and then stop. The P set can be built by using a normal breadth-first visit of the transition diagram of the machine M_S. One way of constructing P is to build first a testing tree T of M_S as explained in Algorithm 2 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 the P set. Note that Algorithm 2 terminates because the number of states is finite.

Example 3. The test tree for M_S of Figure 1 is shown in Figure 3

The W method uses 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. The characterizing set is defined as follows.

Algorithm 2 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 alredy build the tree $T$ up to the level $k$. Now we build the $k + 1$th 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 $j \leq k$, 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 succesor node with label $s_j$
```

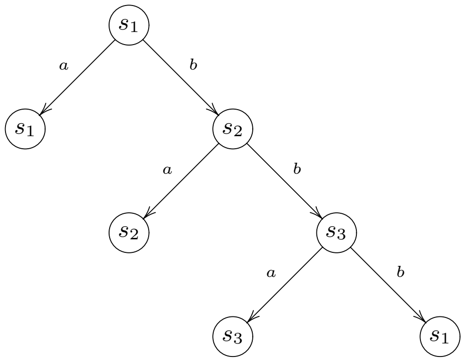


Fig. 3. A test tree for M_S of Figure 1

Definition 4. *([characterizing set]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, 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 x in the W set are also called *separating sequences.* The W set exists for every machine that is minimal (Assumption 1) and can be built as shown in Section FSM1(ref)sec:algorithms . Note that the choice of a W set is not unique and the fewer are the elements in W set the longer are the 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. Note that because W may contain several sequences, we have to visit the same state several times to apply all the separating sequences in the W set. The set of test sequences is simply obtained concatenating the P and W sets and apply them in order after a reset message to take back the machine to the initial state. In this way each test sequence p_{ij} is the concatenation of the i-th sequence of the P set with the the j-th sequence of the W set, with an initial *reset* input. Eeach p_{ij} starts from the initial state (using a *reset* message), then the i-th sequence of the P set takes the machine to $s_i$where we apply the j-th sequence of the W set to observe the output.

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 produce by the W method is equal to $\{reset\}.P.W$.

If we do not observe any fault, the implementation is proved to be correct [?]. 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 4. For the machine in Fig. 1 a characterizing set W is $\{a,b\}$ (equal to the input set I). In fact we have:

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

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

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

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

The set of test sequences P.W is reported in the following table.

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

The total length of the checking sequence is 52.

CONSIDER FAULTY IMPL.

4.2 Wp method

The partial W or Wp method proposed by [?] 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 5. *([state cover set]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 $q \in Q$ that takes the machine to s, i.e. $\delta(s_1, q) = s$*

The state cover set is briefly called Q set. Using a Q set we can take the machine to every state. For the second phase, the Wp method uses an *identification set* $\mathrm{W_i}$ for state s_i instead of an unique characterizing set W for all the states. $\mathrm{W_i}$ is a subset of W and is defined as follows.

Definition 6. *([identification set]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* p *of W_i such that $\lambda(s_i, p) \neq \lambda(s_j, p)$ 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.

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 the W set. Remember that 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 the W set, that is against Definition 4.

If the test does 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 say in this case 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.

Phase 2 During the second phase we have to test all the transitions that were not tested during the first phase. To this aim, Wp method uses the identifications sets. The test 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 pahse 1, concatenated with all the sequences contained in the identification set $\mathrm{W_i}$. Formally if R = P-Q and p_i in R ends in s_i, the set of sequences applied during the second phase is $\{reset\}.R.W_i$. If these tests do not uncover any fault, we have verified that the machine M_I conforms its specification. A proof of correctness for the Wp method is given in [?].

Example 5. The machine in Fig. 1 has the following state cover set: Q = $\{\epsilon$, b, $bb\}$

During the first phase we generate the following test sequences:

state to test	1	1	2	2	3	3
Q	ϵ		b		bb	
Q.W	ra	rb	rba	rbb	$rbba$	$rbbbb$
output	0	1	11	11	110	1110
final state	1	2	2	3	3	1

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.Wi	raa	rab	$rbaa$	$rbbab$	$rbbba$	$rbbbb$
output	00	01	111	1100	1100	1101
final 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)

4.3 UIO methods

If a W_i set contains only one sequence, this sequence is called *state signature* [?] or *unique input/output* (UIO) sequence [?] , that is unique for the state s_i. UIO sequences are extensively studied in Chapter FSM3(ref)chapter for state verification. Remember that applying an UIO sequence we can distinguish state s_i from any other state, because the output produced applying 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 an homing sequence or a synchronizing sequence, presented in Chapter FSM1(ref)chapter: 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 FSM3(ref)chapter. 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 UIO sequences act as status messages.

UIO sequences can be used instead of status messages in a transition tour, visiting every transition from s_i to s_j and then checking the end state s_j by applying its UIO. The UIO method [?] applies first a transition cover set P and then to the state s_i its UIO sequence. This method can substitute the transition coverage method when a status message is not present and it is often used in practice. Some tools presented in Chapter TCS1(ref)chapter use this method. Moreover, because this method requires the application of a single sequence of inputs for each state, instead of a set of separate sequences as in W and Wp methods, it can be easily optimized for the use without reset, using instead a unique checking sequence similar to a transition tour. Such optimized version is given in [?] and the problem of finding the shortest transition tour covering all the transition and then applying an extra sequence, that is a UIO sequence in this case, to their end state is called *Rural Chinese Postman Problem.*

Although used in practice, the UIO method does not guarantee to discover every fault in the implementation [?] 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 Wi 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, that correctly generates checking sequences, is given in [?]. The UIOv method builds the test suite in three phases:

1. *Uv process*: visit every state in S and apply its UIO sequence to check that the state is correct and the transitions to reach that state are correctly implemented. To reach each state use the Q set. This corresponds with the state verification phase of the UIO method. The input sequences consist of the concatenation of Q with the UIO sequence of the final state of the sequence in Q.
2. $\neg$*Uv process*: visit every every state 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 of the input part of the UIO sequence applied in the phase 1. Indeed, in this case, we know already that the outputs differ, 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 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 phase.

Example 6. For the machine in Fig. 1 the UIO sequences are:
$UIO_1 = ab$ distinguishes the state s_1 from all other states
$UIO_2 = a$ distinguishes the state s_2 from all other states
$UIO_3 = b$ distinguishes the state s_3 from all other states
1. Uv process

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

2. ¬Uv process

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

3. Transition test phase:

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

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 in Chapter FSM2(ref)chapter). In this situation we can apply the DS method presented in [?] as modification of the method given in [?] and exploiting the reset message. 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 7. For the machine in Fig. 1 we can take the sequence $x = ab$ as a 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
P.x	rab	$raab$	$rbab$	$rbaab$	$rbbab$	$rbbaab$	$rbbbab$
trans to test	s_1: null	s_1: $a \ / \ 0$	s_1: $b/1$	s_2: $a/1$	s_2: $b/1$	s_3: $a \ / \ 0$	s_3: $b \ / \ 0$
output	01	001	111	1111	1100	11000	11001

Cost and length

All the methods presented in Section 4 share the same considerations about the maximal length of the checking sequence and the cost of producing it. For the W method, the cost to compute the W set is $\mathcal{O}(pn^2)$ and it contains no more than $n-1$ sequences of length no more than n (as shown in Chapter FSM1(ref)chapter). The cost to build the tree T set using the Algorithm 2 is $\mathcal{O}(pn)$ and its maximum level is n. The generation of the 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 the P set with each transition in the 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 $\mathcal{O}(pn^3)$. Wp method has the same cost and same maximum length.

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 FSM3(ref)complexity and FSM2(ref)FSM2$_pds).NotethatinpracticeUIOsequencesaremorecommonthandistin$
and adaptive distinguishing sequences have maximum length n^2. Using adaptive DS, we appy such sequences after every transition. Because there are pn transitions, the total length for the checking sequence is again pn^3.

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 FSM1(ref)sec:intro-homing. However this can lead to very long test suites and it is seldom used in practice. On the other hand, methods like UIO and DS use a single input sequence to test the final state of each transition and they can, therefore, be easily extended in a way that they do not need to use the reset message to visit the next state to be verified. Instead of the reset message we can use transfer sequences.

Definition 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. 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 move the machine to another state when applied. The method proposed by [**?**] exploits distinguishing sequences to perform the conformance test. The methods has, as many methods presented in the previous section, two phases. It first builds a test sequence that visits each state using transfer sequences instead of reset and then applies its distinguishing sequence to test if M_I is similar to M_S. It then builds a test sequence to test each transition to guarantee that M_I conforms with 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 τ_i the transfer sequence from t_i to s_{i+1}, i.e. : $\tau_i = \tau(t_i, s_{i+1})$. For the machine in the initial state s_1, the following test sequence checks the response to the distinguishing sequence in each state.

$$x\,\tau_1\,x\,\tau_2\,x \ldots \tau_n\,x \tag{1}$$

This sequence can be depicted as follows.

$$\begin{tikzcd} s_1 \arrow[r, "x"] & t_1 \arrow[r, "\tau_1"] & s_2 \arrow[r, "x"] & t_2 \arrow[r, "\tau_2"] & {} \arrow[r, dotted] & {} \arrow[r, "\tau_n"] & s_1 \arrow[r, "x"] & {} \end{tikzcd}$$

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.

Phase 2 In the second phase, we want to test every state 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 $\tau(t, s_i)$. Hence, we cannot go directly from t to s_i. On the other hand, we have already verified by (**??**) that $x\tau(t_{i-1}, s_i)$ takes the machine from $s_{i-1}$to s_i. We can build a test 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 final state is s_j by applying again the distinguishing sequence x:

$$\tau(t, s_{i-1})x\tau(t_{i-1}, s_i)ax \tag{2}$$

$$\begin{tikzcd} {} \arrow[r, dashed] & t \arrow[r, "\tau(t{,}s_{i-1})"] & s_{i-1} \arrow[r, "x\tau_{i-1}"] & s_i \arrow[r, "a"] & s_j \arrow[r, "x"] & t_j \end{tikzcd}$$

Therefore, the sequence (**??**) 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 state transition to obtain a checking sequence. The size of the checking sequence is polynomial in the size of the machine M_S and the length of x.

Example 8. A distinguishing sequence for the machine in Fig. 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 $\tau(t_1, s_2) = \tau(t_2, s_3) = \tau(t_3, s_1) = \epsilon$.

The sequence (**??**) becomes

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

The test sequence ends in state s_2
The test sequences (**??**) can be concatenated to obtain:

trans to test	s_3: b / 0	s_2: $a/1$	s_3: a / 0	s_1: a / 0	s_2: $b/1$	s_1: $b/1$
	$\tau(t,_1\, s_3)bx$	$\tau(t_2, s_2)ax$	ax	ax	bx	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.

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 FSM2(ref)FSM2$_a ds.InthatChapter, thereadercanfindthedefinition(FSM2(ref)def : def_{ads}), analgorithmt$

Example 9. An adaptive distinguishing sequence for the machine in Fig. 1 is depicted in Figure 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 $\tau = \epsilon$ and the sequence (**??**) 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

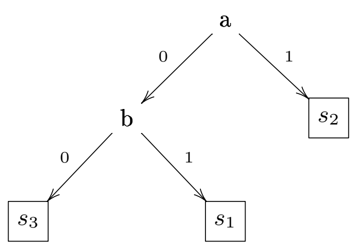


Fig. 4. Adaptive distinguishing sequence of machine in Fig. 1

Length and cost An adaptive distinguishing sequence has length $\mathcal{O}(n^2)$, and a transfer sequence cannot be longer than n . The sequence (**??**) is long $\mathcal{O}(n^3)$. Because there are pn transitions, and every sequence (**??**) 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 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 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 [**?**], but still resulting checking sequences have length $\mathcal{O}(pn^3)$.

6 Using Identifying Sequences instead of Distinguishing Sequences

Not every finite state machine has distinguishing sequences. 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 (Definition 4), that can distinguish a state from any other state in M_S. In this case, conformance testing is still possible [**?**], although the resulting checking sequences may be exponentially long.

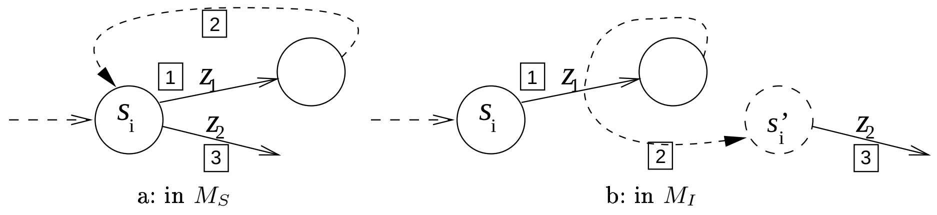


Fig. 5. Using two separating sequences to identify the state

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 its separating set Z_i. Suppose that Z_i has two separating sequences z_1 and z_2. We want to apply the steps shown (in square boxes) in Figure 5 (a) : take $M_I$to s_i, apply z_1(step 1), take the machine back again to s_i (step 2) and then apply z_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 back the machine M_I to s_i in a verifiable way, because in a faulty machine, as shown in Figure 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 z_2 sequence, without being able to verify that s'_i is s_i and without able to apply again z_1. We use now the Assumption 6 on page 4, namely that M_I has only n states.

Theorem 1. *Let s be a state, x be an input sequence, o the output sequence produced applying x to s, and τ a transfer sequence from $t_x = \delta(s, x)$ back to s. By applying the test sequence $(x\,\tau)^n$ to state s, the machine ends in a state where applying again x we observe the same output o.*

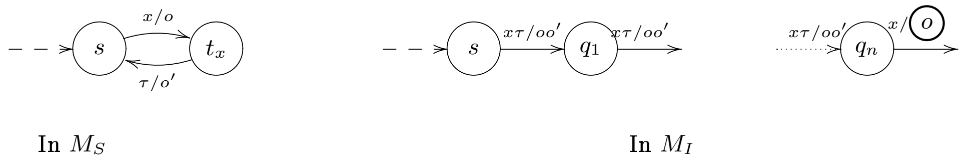


Fig. 6. Applying z_1 amd z_2

Proof. XXX DA RIVEDERE The scenario described in the theorem is shown in Figure 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. Applying n times $x\,\tau$, we check that the output is always the same. Let q_r be the state of M_I after the application of $(x\,\tau)^r$. The n applications of $x\,\tau$ produce the same output, but we are not sure that $s, q_1 ... q_n$ 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 if we apply x.

Example 10. Consider the machine in Figure 1 and take an alleged implementation M_I. Apply the input a (in this case $\tau = \epsilon$) 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 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 q_3. Because M_I has only 3 states, q_3 is equal to one of s_1, q_1,q_2 and we are sure that if we applied again a we would observe 0

We use Theorem 1 as follows. Assume that s_i has the separating set $Z_i = \{z_1, z_2\}$. We first apply $(z_1\tau(t_i, s_i))^n$ and thanks to the theorem we end in a state that would produce the same output as if we applied z_1. We apply z_2 instead. If we observe the specified output we can conclude that s_i has a similar state in M_I.

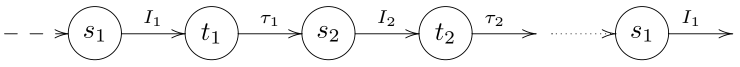
We can generalize this method when the separating set Z_i contains m separating sequences. Suppose that the separating set for state s is $Z_i = \{z_1, \ldots, z_m\}$. Let τ_j be the transfer sequence that takes the machine back to s after the application of z_j, i.e. $\tau_j = \tau(\delta(s, z_j), s)$. We can define inductively the sequences β_r as follows:

$$\beta_1 = z_1$$
$$\beta_r = (\beta_{r-1}\tau_{r-1})^n z_r$$

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 $z_1, ..., z_{r-1}$ would produce the same output . For this reason we apply z_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 z_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 5 to visit each state, verify its response to the identifying sequence, and then transfer to the next state. Let I_i the identifying sequence of state s_i and τ_i the transfer sequence from $t_i = \delta(s_i, I_i)$ to s_{i+1}, applying the following test sequence:

$$I_1\,\tau_1\,I_2 ... I_1 \tag{3}$$



we can verify that M_I is similar to M_S.

Once we have 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 $t_1 = \delta_I(s_1, z_m)$ and use t_1 as the initial state for all the transition verification. if we want to reset the machine from the state s_k to t_1 we can 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. To test a transition from s_i to s_j we apply a pseudo reset I_1 to t_1, then a transfer along tested transitions to s_i, then we apply the input, observe the output, and apply the identifying sequence I_j.

Example 11. Consider the machine M_S in Fig. 1

$I_1 = aaa\ b$

$I_2 = aaa$

$I_3 = bbb$

The sequence (**??**) becomes

	I_1	τ_{12}	I_2	τ_{23}	I_3	τ_{31}	I_1
input sequence	$aaab$	b	aaa	b	bbb	b	$aaab$

Length and cost The length of an identifying sequence grows exponentially with the number of separating sequences of a state and the resulting checking sequence is exponentially long. SPIEGA MEGLIO.

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 *universal traversal* problem [?].

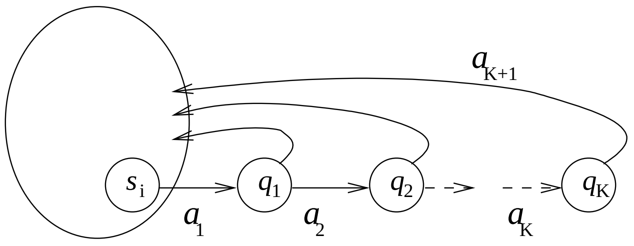


Fig. 7. A faulty machine M_I with K extra states

Assume that a faulty machine M_I , depicted in Figure 7, is identical to M_S except for the state s_i on input a_1 where M_I moves to extra states $q_1, ..., q_k$. 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 test sequence applied to state s_i must include all possible input sequences of length K+1, and thus it must have length $|I|^{K+1}$. Such test 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}. Valisveski [?] showed that also the lower bound on the test sequence is multiplied by $|I|^K$; i.e. it becomes $\Omega(|I|^{K+1}|S|^3)$ (discussed also in Chapter BT2(ref)chapter Section 5). 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 test sequence must have length at least $|I|^{K+1}|S|$.

Using similar considerations, many methods we have presented can be easily extended to deal with implementations that may add a bound 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 is extended to test a machine with m states with $m > |S_S| = n$. The W method in this case use instead of W another set of sequences called ***distinguishing set*** $\mathrm{Y} = (\epsilon \cup I \cup I^2 \cup \ldots \cup I^{m-n}).W$. Hence 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. If $m = n$ then Y=W. Each test sequence starts with a reset, then applies a sequence to test each transition, then a applies a number of inputs till m-n, then applies a separating sequence of W. The set of test 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 [**?**].

Example 12. Consider the machine in Fig. 8 as faulty implementation 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.

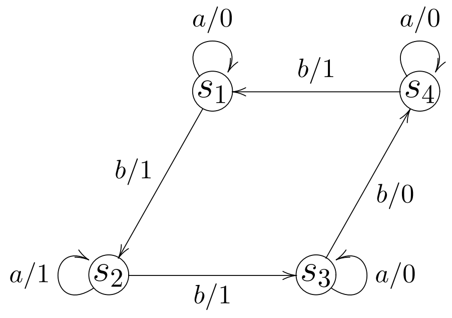


Fig. 8. A faulty implementation of machine M_S with 4 states

8 Comparison and Practical Implications

A comparison among the methods presented in this chapter should include two main factors: the cost and the length to produce and execute the test suites and the fault detection capability. A theoretical study shows that W, Wp, UIOv, DS and IS methods, under the given assumptions, have the same fault detection capability. The TT method and the UIO method can discover any output fault, while the ST method, covering only the states, may miss faults. The total length of the test suite is greater for methods like W and Wp than for methods like TT and UIO. The DS and IS methods without reset leads to even longer test suites. it is interesting to compare the methods when the assumptions do not hold. This kind of study can be found in [**?**,**?**]. Indeed, assumptions like the equal number of states for implementation may be not true 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 7, faults in extra states are more likely to be discovered when using long test sequences. The use of reset function may prevent the implementation to reach such extra states where the faults are present. For this reason methods like UIO or DS method 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 TCS1(ref)chapter and TCS2(ref)chapter. A first practical implication is that 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 method, that can be considered as a branch coverage. Transition coverage should be used in conjunction of a status message to really check the end state of every transition. 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 status. If a status message is not available (very usual 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 many more states than the implementation, he/she should prefer long test sequences that can be obtained simply adding some extra inputs after visiting the transition and before checking the status identity. Such practical suggestions may be not guarantee to discover any possible faults, but may dramatically increase the likelihood of success of the testing activity

9 Summary

In this chapter we have presented the conformance testing for finite state machines. The methods we have presented assume the facts presented in Section 2, some of which, however may be relaxed. The first method in Section 3, the Transition Tour (TT) method, exploits all the assumptions, including a *status* message to check that the implementation is in the correct state. If a status message is not available, but the machine has a *reset* message to go to its initial state, one can use one of the methods proposed in Section 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. In Section 5 we have presented how distinguishing sequences can be used without reset. If the machine has no DS, the identifying sequences (IS) method, presented in Section 6, still works. The IS method exploits only the assumptions that the number of states is finite and that separating sequences exist in minimized specification machines. The problem of testing finite state machines with extra states is discussed more in general in Section 7. Section 8 discusses some practical implications of the methods presented and presents a brief comparison among them.

References

[Aho et al., 1991] Aho, A., Dahbura, A., Lee, D., and Uyar, U. (1991). An optimization technique for protocol conformance test generation based on uio sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615.

[Belina and Hogrefe, 1989] Belina, F. and Hogrefe, D. (1989). The CCITT Specification and Description Language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341.

[Chow, 1978] Chow, T. S. (1978). Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187. Special collection based on COMPSAC.

[Edmonds and Johnson, 1973] Edmonds, J. and Johnson, E. L. (1973). Matching, euler tours and the chinese postman. *Math. Programming*, 5:88–124.

[Fujiwara et al., 1991] Fujiwara, S., v. Bochmann, G., Khendek, F., Amalou, M., and Ghedamsi, A. (1991). Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603.

[Gonenc, 1970] Gonenc, G. (1970). A method for the design of fault detection experiments. *IEEE Trans. Computers*, C-19(6):551–558.

[Goodenough and Gerhart, 1975] Goodenough, J. B. and Gerhart, S. L. (1975). Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173.

[Gurevich, 1994] Gurevich, Y. (1994). Evolving algebras 1993: Lipari Guide. In Börger, E., editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press.

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

[Heitmeyer et al., 1996] Heitmeyer, C., Jeffords, R., and Labaw, B. (1996). Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261.

[Hennie, 1964] Hennie, F. C. (1964). Fault detecting experiments for sequential circuits. In *Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, Princeton, New Jersey. IEEE.

[Holzmann, 1991] Holzmann, G. (1991). *Design and validation of computer protocols.* Prentice Hall.

[Kwan, 1962] Kwan, M.-K. (1962). Graphic programming using odd or even points. *Chinese Math*, 1:273–277.

[Lee and Yannakakis, 1996] Lee, D. and Yannakakis, M. (1996). Principles and methods of testing finite state machines – a survey. *Proc. IEEE*, 84(8):1090–1126.

[Moore, 1956] Moore, E. F. (1956). Gedanken-experiments on sequential machines. In Shannon, C. E. and McCarthy, J., editors, *Automata Studies*, number 34 in Annals of Mathematics Studies, pages 129–153. Princeton University Press, Princeton, NJ.

[Naito and Tsunoyama, 1981] Naito, S. and Tsunoyama, M. (1981). Fault detection for sequential machines by transition tours. *Proc. of IEEE Fault Tolerant Computing Conference*, pages 238–243.

[Rumbaugh et al., 1999] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1 edition.

[Sabnani and Dahbura, 1988] Sabnani, K. and Dahbura, A. (1988). A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15(4):285–297 (or 285–298??).

[Sato et al., 1989] Sato, F., Munemori, J., Ideguchi, T., and Mizuno, T. (1989). Test sequence generation method based on finite automata - single transition checking using w set. *Trans. of EIC (in Japanese)*, J72-B-I(3):183–192.

[Sidhu and Leung, 1988] Sidhu, D. and Leung, T. (1988). Experience with test generation for real protocols. In *Symposium proceedings on Communications architectures and protocols*, pages 257–261. ACM Press.

[Sidhu and Leung, 1989] Sidhu, D. and Leung, T.-K. (1989). Formal methods for protocol testing: a detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426.

[StateFlow,] StateFlow. Stateflow.

[Ural et al., 1997] Ural, H., Wu, X., and Zhang, F. (1997). On minimizing the lengths of checking sequences. *IEEETC: IEEE Transactions on Computers*, 46(1):93–99.

[Vuong et al., 1990] Vuong, S., Chan, W., and Ito, M. (1990). The UIOv-method for protocol test sequence generation. In *Proc. 2nd Int. Workshop on Protocol Test Systems*, pages 161–176.

[Yannakakis and Lee, 1991] Yannakakis, M. and Lee, D. (1991). Testing finite state machines. In Awerbuch, B., editor, *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing*, pages 476–485, New Orleans, LS. ACM Press.

[Zhu and Chanson, 1993] Zhu, J. and Chanson, S. T. (1993). Fault coverage evaluation of protocol test sequences. Technical Report TR-93-19, Department of Computer Science, University of British Columbia.