CONFORMANCE TESTING 9

9.1 INTRODUCTION

The goals of protocol conformance testing and of protocol validation are easily confused.

- A *conformance test* is used to check that the external behavior of a given implementation of a protocol is equivalent to its formal specification.
- A *validation* is used to check that the formal specification itself is logically consistent.

If a formal specification has a design error, a faithful implementation of that specification should *pass* a conformance test if and only if it contains the same error. A conformance test should fail only if implementation and specification differ. A consistency validation of the protocol, however, must always reveal the design error. In this chapter we study conformance testing methods. Chapters 11 and 13 are devoted to consistency validation.

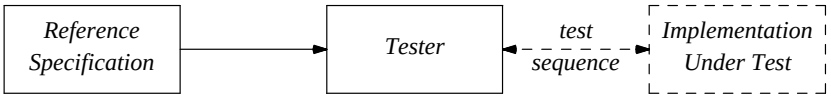

Figure 9.1 — Conformance Testing

We are given a known reference specification, for instance in finite state machine format, and an unknown implementation. For all practical purposes, the implementation is a black box with a finite set of inputs and outputs. The only type of experiment we can do with the black box is to provide it with sequences of input signals (messages) and observe the resulting output signals. The implementation under test, commonly referred to as the *IUT*, passes the test only if all observed outputs match those prescribed by the formal specification. A series of input sequences that is used to exercise the protocol implementation in this way is called a *conformance test suite*. The test is derived from the reference specification, ideally by a mechanical procedure

(Figure 9.1).

There are two main problems to be solved.

- Finding a generally applicable, efficient procedure for generating a conformance test suite for a given protocol implementation.
- Finding a method for applying the test suite to a running implementation.

The second problem looks simpler than it is. The IUT may be a single layer in a hierarchy of protocol functions with two interfaces to surrounding layers, as illustrated in Figure 2.12 on page 31. To test this layer we may need both an upper and a lower tester, and some systematic method for coordinating the sequences they generate. Another complicating factor exists when the IUT and the tester are physically separated from each other, as illustrated in Figure 9.2.

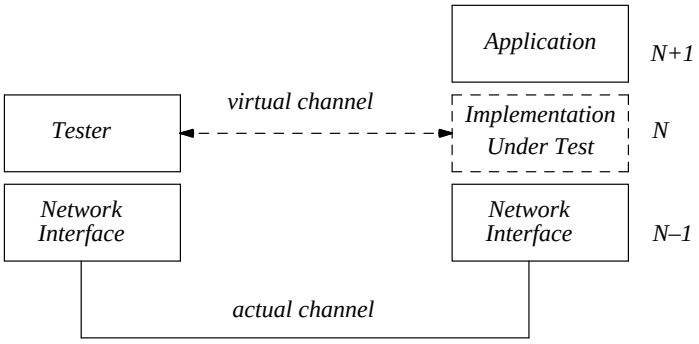

Figure 9.2 — The General Conformance Testing Problem

The tester may only be able to access the IUT via a remote network connection, and may not be able to supply inputs and retrieve outputs from the IUT in a completely reliable manner. In this chapter we discuss only the first problem: the problem of deriving high-quality conformance test sequences.

9.2 FUNCTIONAL TESTING

Protocol conformance testing became an issue when the administrators of the first public data networks had to determine the adequacy of commercial equipment that was to be used on their networks. The problem was to verify the conformance of the equipment to the network standard without having access to the, often proprietary, internal details of the equipment. In the early 1980s, the first attempts to build effective protocol test suites, therefore, had two main goals.

- To establish that a given implementation realizes all functions of the original specification, over the full range of parameter values.
- To establish that a given implementation can properly reject erroneous inputs in a way that is consistent with the original specification.

An example of a functional test of the first type could be a basic *interconnection test*, meant to establish that the IUT is minimally able to set up and to tear down a standard

connection. An example of a test of the second type could be a *format test*, used to verify that the IUT properly rejects violations of the required packet format and violations of the consistency of the packet content (e.g., checksum or byte-count errors).

The problem encountered in these tests is a conflict between complexity and standardization. It is virtually impossible to exhaustively test all possible behaviors of an unknown implementation by simply probing it and observing its responses. There is always a possibility that some untried sequence of probes would reveal a new behavior that is unacceptable. The specific test suite selected for a conformance test of this type, therefore, is always a small selection of the infinite set of all possible test suites. To prevent a manufacturer from rigging a device to pass a given conformance test rather than making it equivalent to the specification proper, the test suites for functional conformance testing cannot be standardized or published.

There is, however, a basic unfairness in requiring a manufacturer to pass a test without making public what the test is, or without standardizing the test in such a way that all competing manufacturers have to submit their equipment to the same test. Ultimately, the complexity of conducting the tests, the difficulty of standardizing them, and the uncertainty of their value has led to a new approach to conformance testing. This new approach has the following purpose.

- To establish that the control *structure* of the implementation conforms to the structure of the specification. Implementation and specification have the same structure if they model equivalent sets of states and allow for the same state transitions.

Though a test suite of this type can easily be standardized, there are no methods available yet that will work for protocols of arbitrary complexity, taking into account, for instance, internal variables, message parameter values, and timer settings. Good methods are known only for a restricted class of protocols that can be specified by non-extended finite state machines. The remainder of this chapter is devoted to a discussion of those methods.

9.3 STRUCTURAL TESTING

No data or parameter values are considered in this type of test. Instead, the emphasis is on the control structure of the protocol. Again, probing an unknown device to compare its internal structure with that of a reference specification is generally impossible. We have to make some simplifying assumptions.

1. The IUT models a deterministic finite state machine with a known maximum number of states and with a known input and output vocabulary.
2. The IUT produces a response to an input signal within a known, finite amount of time.
3. The states and the transitions of the IUT form a strongly connected graph: every state in the graph is reachable from every other state in the machine via one or more state transitions.

A *state* of the IUT, for the purposes of this discussion, is defined as a stable condition in which the IUT is waiting for a new input signal. A *transition* is defined as the

consumption of an input signal, the possible generation of an output signal, and the possible move to a new state. For a reproducible test result the move must be a deterministic one, which means that the model of a finite state machine that we can use is a subset of the model discussed in Chapter 8. However, since we are discussing concrete implementations rather than abstract designs, the determinism is not likely to be a restriction.

The three properties listed above are requirements. Without them a conformance test of the type to be discussed is not possible. In the remainder we also assume that the IUT corresponds to a completely specified finite state machine.

4. In each state the IUT can accept and respond to all input symbols from the complete system vocabulary. A null response, i.e., a transition back to the same state, is a valid response.

This *completeness assumption* allows us to generalize the algorithms below by removing a special case, but it is not a requirement. In many cases, a conformance test can even be shortened if not all possible input combinations need to be tested.

The IUT can also have properties that can simplify the task of conformance testing itself. Unlike the first three requirements, the following three properties are convenient but not essential.

5. *Status property*. When a ''status'' message is received, the IUT responds with an output message that uniquely identifies its current state. The IUT does not change state.

6. *Reset property*. When a ''reset'' message is received, the IUT responds by making a transition to a known initial state, independent of its current state. The IUT need not produce an output.

7. *Set property*. When a ''set'' message is received in the initial system state, the IUT responds by making a transition to the state that is specified in a parameter of that message. The IUT need not produce an output.

Given a machine with all seven properties listed above, a conformance test can be performed as follows.

ALGORITHM 9.1 — CONFORMANCE TESTING

1. For all possible combinations of a state *i* and an input signal *j*, perform the following three steps.

2. Use the *reset* message to bring the IUT to the initial state, and then use the *set* message to transfer the IUT to state *i*.

3. Apply input signal *j*. Verify that any output received, including the null output, matches the output required by the specification.

4. Use the *status* message to interrogate the IUT about its final state. Verify that this final state matches the one required by the specification.

The test verifies that the IUT is capable of correctly performing all state transitions in the formal specification. The set of input signals tested should, of course, include the *set*, *reset*, and *status* messages. If the IUT passes these tests, it is *capable* of reproducing the behavior of the formal specification, but it remains unknown if the IUT is

capable of any other behavior. Specifically, if the IUT is faulty, it may violate the first requirement for conformance testing that we listed above. The acceptance of an input signal that is outside the official input vocabulary may then cause a transition of the faulty IUT into a set of states that produces erroneous behavior.

Within these constraints, the result of Algorithm 9.1 is the best we can hope to achieve with a conformance test. But is it also the best possible algorithm? The cost of the test can be expressed as the length of the test suite, that is as the total number of messages that is sent to the IUT. Assume that the formal specification contains $S = |Q|$ states and has an input vocabulary of V distinct messages, which includes the set, reset, and status messages. The length of the test suite for Algorithm 9.1 then is

$$4SV$$

After every test the IUT is forced back into the initial state. We can avoid that if we can find a sequence of state transitions that passes through every state and every transition at least once. Such a sequence of transitions is called a *transition tour*. At best such a transition tour starts with a single reset message and exercises every transition exactly once, each time followed by a status message to verify the destination state. A set message is no longer required. The length of the test suite is now minimally

$$1 + 2SV$$

The problem is now to find the minimal transition tour or one that is as close as possible to it. It is a standard problem from graph theory. An *Euler tour* in a directed graph is a sequence of transitions that starts and ends at the same state and contains every transition exactly once. A sufficient condition for the existence of an Euler tour is that the graph (the finite state machine) be both strongly connected and *symmetric*, that is, every vertex (state) must be the destination and the origin of the same number of edges (transitions).

Given a symmetric and strongly connected graph, an Euler tour can be found with a standard procedure that is summarized in Algorithm 9.2. In the first step, the algorithm derives a simple spanning tree of the graph. A spanning tree of a graph contains all its vertices, but only a subset of the edges. In the spanning tree that is constructed in Algorithm 9.2 any vertex can have multiple incoming edges, but is restricted to only one outgoing edge, as illustrated in Figure 9.3. In graph theory, a spanning tree with this property is called a *spanning arborescence*.

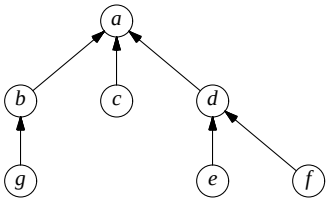

Figure 9.3 — A Spanning Arborescence

Every directed edge represented in the spanning arborescence must be present in the original graph. In Figure 9.3, for instance, there must exist edges in the original graph from vertex *b* to *a*, from *g* to *b*, and so on. In the second step of Algorithm 9.2, the tree is used to decide in which order edges should be added to the transition tour. Edges in the spanning arborescence are added last.

An edge *i* in the graph that starts at vertex *s* is written (s,i). Its destination is written $dest(s,i)$. The set of vertices that are represented in the spanning arborescence is called *T*. After the first step of Algorithm 9.2 is completed *T* should equal the set of vertices (states) in the original graph *Q*.

ALGORITHM 9.2 — DERIVING A TRANSITION TOUR

1. Spanning Arborescence — Choose an arbitrary vertex of the original graph and add it to set *T*. This vertex will become the root of the spanning arborescence. Next, select an edge (s,i) with $s \notin T$ and $dest(s,i) \in T$ add vertex *s* and edge (s,i) to the spanning arborescence. Vertex *s* is added to *T*. Continue to grow the tree until no more vertices can be added.

2. Transition Tour — Beginning at the vertex that was chosen as the root node of the spanning arborescence, select an outgoing edge and move to the corresponding destination vertex. The lowest priority in the selection of outgoing edges is given to the edges that are part of the spanning arborescence. The other edges can be chosen in arbitrary order. Continue to grow the transition tour until no more edges can be added.

As an example, consider the symmetric graph in Figure 9.4. To identify the edges, we have labeled them with letters. Multiple labels represent multiple edges. There are, for instance, three directed edges from `q0` to `q2`, named *d*, *e*, and *f*.

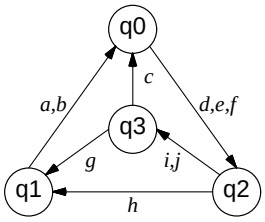

Figure 9.4 — A Symmetric Graph

A spanning arborescence with root `q3` could contain, for instance, edges *i*, *d*, and *b*.

Using Algorithm 9.2, two different transition tours based on this spanning arborescence are then *c, e, j, g, a, f, h, b, d, i* and *g, a, f, j, c, e, h, b, d, i*.

If the graph we are considering is not symmetric, we must first transform it into a symmetric graph by duplicating edges. Every edge that is duplicated then corresponds to a transition that will be executed more than once in the final transition tour. In graph theory, the duplication is called an *augmentation* of the original graph. Algorithm 9.3 is a simple way to derive such an augmentation (cf. Exercise 9-12).

ALGORITHM 9.3 — GRAPH AUGMENTATION

1. Label every vertex in the graph with an integer that represents the difference between the number of outgoing and incoming edges for that vertex. This number can be positive, zero, or negative. Since, by definition, every edge has both an origin and a destination, the sum of all label values must be zero.

2. Select a vertex *A* with a negative label and a vertex *B* with a positive label. Find the shortest path from *A* to *B* by traversing the fewest number of edges in the *original* graph. Duplicate the edges along this path. Update the labels of *A* and *B*. The labels on the intermediate vertices do not change.

3. If the augmented graph is symmetric the algorithm terminates. The cost of the augmentation is the total number of edges that have been duplicated. If the graph is not symmetric, return to step 2.

Verify, for instance, that the graph in Figure 9.4 is a symmetric augmentation of the state transition diagram in Figure 8.1 (cf. Exercise 9-4).

Step 2 of the algorithm calls for the calculation of the shortest distance between two vertices. A range of algorithms has been studied to solve this problem efficiently. Refer to the Bibliographic Notes at the end of this chapter for an overview. The ultimate cost of the augmentation depends in a subtle way on the choice of vertices that is made in step 2 of Algorithm 9.3. Fortunately, there is only a finite number of ways in which these choices can be made each time step 2 is executed. Therefore, we could try to find the minimum-cost augmentation by exhaustive search. There are, however, better methods. One method is to study graph augmentation as a network flow problem. The problem can then be represented as a *minimum-cost flow* problem, which can be solved in polynomial time (see Bibliographic Notes).

After a symmetric augmentation Algorithm 9.2 can be used to derive a transition tour. For a minimum-cost augmentation, the transition tour produced by Algorithm 9.2 will also be the shortest, and therefore the lowest cost test suite for the IUT. The problem of finding a transition tour in a non-symmetric graph, where every transition is exercised at least once, and possible more than once, is known as the *Chinese Postman Problem*. As indicated above, the problem can be solved in polynomial time.

To derive the test sequence with Algorithms 9.1 we assumed that the IUT has three properties: a reset message, a set message, and a status message. The set message is an oddity that is not likely to be present in many IUTs, but fortunately, in the construction of test sequences based on a transition tour with Algorithm 9.2, we did not

need it anymore.

The absence of the *reset* and the *status* messages are more problematic. The reset message can be replaced by a sequence of transitions, called a *homing sequence*, that is known to bring the system back to the initial state, whatever its current state may be. In general, a homing sequence is defined as an adaptive procedure, where the responses generated by the machine can be used to determine what the next input message should be. It can be shown that every strongly connected finite state machine must have such an adaptive homing sequence. Better still, the homing sequence can be derived algorithmically. It can be shown that it need never take more than $S(S-1)/2$ transitions before the machine reaches a known state (see Bibliographic Notes). To reach the initial system state after that point is reached requires between zero and $S-1$ extra transitions. Consider, for example, the machine in Table 9.1.

Table 9.1 – Example

Current State		In	
q0	0	0	q1
q0	1	0	q1
q1	0	1	q0
q1	1	1	q0

The machine has two states, so $S=2$ and $S(S-1)/2 = 1$. A sequence of length one can tell us in which state the machine is. It never takes more than $(S-1) + S(S-1)/2 = 2$ inputs to reset the machine to state q0 with certainty.

A more significant challenge is posed by the omission of the *status* message. The status property is also not likely to be present in an IUT, if it is not needed for the normal operation of the protocol. To replace the status message, we can use a sequence of transitions called a *state signature* or *Unique Input/Output* (UIO) sequence. A UIO sequence can determine whether the IUT is in a given state when the UIO begins. A UIO sequence then has the opposite goal of a homing sequence: it identifies the first instead of the last state in the sequence. To be able to verify every state in the IUT, we must be able to derive a UIO sequence for every state separately.

Not all UIO sequences are necessarily different. In fact, it may be possible to derive a single UIO sequence that can be used to identify any state in a finite state machine. Such a sequence is called a *distinguishing sequence*. The finite state machine that was discussed in Chapter 8 (cf. Figure 8.1) illustrates, however, that not all finite state machines have such a distinguishing sequence. It can also be shown, in much the same way, that not all states have a UIO sequence. In the next two sections we first assume that a UIO sequence can be derived for all states in the specification. We show how these UIO sequences can be used to replace the status messages in a transition tour.

The method discussed below is popular, but it must be remembered that, with the

replacement of the status message by a UIO sequence, the fault-detecting power of the transition tour is reduced. The UIO sequences, after all, assume a correct implementation. A faulty machine could in principle fool an observer into believing that a given state has been reached by accidentally generating just the right responses to a precomputed UIO sequence. An alternative method that is not based on UIO sequences is discussed in Section 9.6.

9.4 DERIVING UIO SEQUENCES

In this section we first show how UIO sequences can be derived, assuming that they exist. In the next section we show how the transition tour can be modified to counter the side-effect of the application of UIO sequences.

So far, the best known method to find UIO sequences is to enumerate all possible I/O sequences and to check them for the UIO property. Algorithm 9.4 accomplishes that by building an exponentially expanding tree of I/O sequences. The nodes in the tree at distance N from the root correspond to the I/O sequences of length N. Each node has associated with it two sets of states. The first set, P, contains a partitioning of the set of S states into classes, where S, by definition, is equal to $|Q|$. Two states are in the same class of the partitioning P if and only if they cannot be distinguished from one another by the application of the I/O sequence represented by the node: the specification produces the same outputs under this sequence, no matter which of these states is chosen as an initial state. The members of the second (ordered) set T define for each state in S what the final state will be if the I/O sequence represented by the node were applied with that state as an initial state.

Let $dest(i,j)$ again be the state that the IUT should reach if input j is received while in state i, and let $output(i,j)$ define the output signal that is generated during the transition, if any. Further, let $T[i]$ define the i-th element of set T for the current node, and $T_j[i]$ the i-th element of set T for its j-th successor node.

ALGORITHM 9.4 — UIO DERIVATION

1. Initially, partitioning P consists of a single set that includes all S states of the specification. Set T has S members. The initial value for the i-th member in T, with $1 \le i \le S$, is i. The tree of I/O sequences is initialized to a single node, called the root node, which corresponds to the null sequence. Initially, the root node is the only leaf node in the tree. (A leaf node is a node without successors.)

2. Sort the leaf nodes of the tree in a list and delete duplicates. To every leaf node now assign V successor nodes in the tree, one for every possible input signal. Set $T_j[i] = dest(T[i],j)$.

3. The partitioning P_j for the j-th successor node is derived from the current partitioning P as follows. Let set O define the output signals associated with each of the S states for the last transition. Consider each class in P separately. Make a list of all distinct output signals that are generated by the states that are included in the class considered. This class in P is now split into sub-classes in such a way that all states that generated the same output signal are assigned to the same sub-class. If all states within the class considered generated the same output signal for the last input symbol

applied, they all remain in the same class of the partitioning.

4. If there is a class in the partitioning of P at this point that contains just one state, the node that holds this partitioning will define a UIO sequence for that state.

5. Steps 2 to 4 are repeated until UIO sequences for all states have been found (or until available memory is exhausted).

This algorithm searches UIO sequences for all states in the specification at the same time. Because it exhaustively checks all possible input sequences, with the shortest sequences being inspected first, it finds the shortest UIO sequences first. The algorithm requires a rapidly growing amount of space to pursue the search for sequences beyond the first few levels. In the worst case the number of nodes in the tree for sequences of length n is

$$\sum_{i=0}^{n} V^i$$

This means that for an input alphabet of ten messages or more it becomes impractical to search for sequences that are longer than five or six steps. The problem of determining if a state has a UIO sequence was proven to be PSPACE hard, and similarly the problem of determining the shortest possible UIO sequence, given that at least one such sequence exists (see Bibliographic Notes). In practice, however, UIO sequences can often be found within the limits of the algorithm.

9.5 MODIFIED TRANSITION TOURS

Next it must be shown how the UIO sequences can be incorporated into a transition tour to construct a conformance test. A problem is that a UIO sequence will in general leave the IUT in a state different from the one that is being verified and thus interferes with the transition tour.

Call the UIO sequence that identifies state i UIO_i, and call its final state $dest(UIO_i)$. For every state i we now augment the graph of the original specification with a ''pseudo-transition'' for each input symbol j in state i:

$$(i, dest(UIO_{dest(i,j)}))$$

This pseudo-transition consists of the edge traversed for the test of input signal j followed by the verification of the target state, using the UIO sequence for that state. With S states and V input symbols, there are trivially SV pseudo-transitions. The problem of modifying an existing transition tour for the inclusion of UIO sequences, then, is really the problem of finding a new transition tour through the pseudo-transitions only. Call the graph containing only pseudo-transitions the pseudo-graph.

We make a symmetric augmentation of the pseudo-graph and then compute an Euler tour with Algorithm 9.2. For the augmentation we can use Algorithm 9.3, but with one important exception: the calculation of the shortest distance in step 2 is based on the original graph without the pseudo-transitions.

This problem of finding a transition tour through a subset of the edges of a graph (i.e.,

the pseudo-edges) is known as the *Rural Chinese Postman Problem*.

9.6 AN ALTERNATIVE METHOD

The method based on UIO sequences can be used to produce conformance tests of good quality, but it has some drawbacks. First, not all states in a finite state machine necessarily have a UIO sequence, and even if they do, the UIO sequence may be too long to be derived algorithmically. The problem of deriving UIO sequences is PSPACE-complete, which means that only very short UIO sequences can be found in practice. Second, the UIO sequences can only reliably identify states in a correct IUT. It is unknown, and unknowable, what their behavior is for faulty IUTs. In particular, they cannot guarantee that any type of fault in an IUT remains detectable with the modified transition tours.

If further assumptions can be made about the types of faults in the IUT, the construction of a reliable test is possible, and can be done with a polynomial time algorithm. The main assumption is that no fault can increase the number of reachable states or the number of input signals of the IUT. The method we discuss below is based on the use of *characterizing sequences*.

CHARACTERIZING SEQUENCES

Assume that the original protocol specification corresponds to a minimized finite state machine. For every two states from this machine there exists a finite sequence of inputs that triggers a different sequence of outputs. Such a sequence is called a *characterizing sequence*. It can be shown that every characterizing sequence has a length smaller than S steps, the number of states in the machine. Though there are $S(S-1)/2$ distinct pairs of states in a machine, it is easy to see that no more than $(S-1)$ different characterizing sequences are needed to separate any combination of two states. The $S-1$ characterizing sequences can be selected from the maximal set of $S(S-1)/2$ sequences as follows.

ALGORITHM 9.5 — SELECTING CHARACTERIZING SEQUENCES

1. Select two arbitrary states from the machine and find a sequence that separates them. The different output sequences in response to this sequence can be used to partition the S states into at least two different sets. The sets are blocks in a partitioning of states.

2. Select one of these blocks containing more than one state. Select two states from that block and find a sequence from the original collection that can separate them.

3. The number of state sets (blocks in a partitioning of the states) is increased by at least one extra set for each new characterizing sequence that we find. The procedure, therefore, can be repeated at most $S-1$ times.

At this time each block in the partitioning contains just a single state. For any two states in two different blocks we have now selected a sequence that can separate them. The set of $S-1$ characterizing sequences selected from the original collection can be used to distinguish between any pair of states in the machine.

Let $CS(i,j)$ be the characterizing sequence that distinguishes state i from state j. Let $P(i)$ be a sequence of inputs that leads the machine from the initial state to state i in

the reference specification, and let R be the reset message that returns the machine to the initial state. The conformance test can now be performed as follows. The algorithm starts by numbering the states of the finite state machine in breadth-first search order. The numbers are later used to make sure that states that can be reached in the fewest number of transitions from the initial state are tested first.

ALGORITHM 9.6 — CONFORMANCE TESTING

1. Number the states of the machine in breadth-first order. This can be done by constructing a spanning tree of the states. The initial state of the machine becomes the root node of the tree. Initially it is the only leaf in the tree (a node without successors) and it gets the lowest number in the breadth-first search order.

2. To every leaf of the tree we connect all states that can be reached by a single transition in the finite state machine. No state, however, can be added to the tree more than once. The new leafs are numbered consecutively, in arbitrary order. Step 2 is repeated until all states from the machine are represented.

3. The k-th state in breadth-first search order, $1 \le k \le S$, is tested with the input sequence.

$$\forall(i), i<k \;\rightarrow\; R\ P(i)\ CS(i,k)\ R\ P(k)\ CS(i,k)$$

The test sequence checks that the k-th state can be distinguished from all states with a lower breadth-first search number, i.e., from all states that were checked before. Passing the k-th test in this series shows that the IUT has at least k distinct states and that transitions along the edges of the tree, from the initial state to each one of these states, are correctly implemented.

4. Next, all remaining transitions of the machine must be verified. In general, for every state i and transition j, we must perform the following test

$$R\ P(i)\ j$$

We can skip testing transitions that correspond to edges in the breadth-first search tree; they were already tested in step 3. After the output in response to input j has been verified, the new state that has been reached must again be checked, using the method from step 3, by comparing it systematically against all other states in the specification.

The breadth-first search order guarantees that the paths $P(i)$ are verified one transition at a time. If state i can only be reached from the initial state after passing through some other state j, the search order guarantees that state j is verified first.

The total cost of identifying the S states is $O(S^3)$. The cost of verifying a single transition is $O(S^2)$, and since there are VS transitions the total cost is $O(VS^3)$. The cost, finally, of deriving the characteristic sequences (another standard graph theory problem) is $O(VS^2)$.

This cost of the conformance test, therefore, is still polynomial in S and V, and, unlike the UIO based method, the cost of its construction is also polynomial in S and V. Also, unlike the UIO based method, a conformance test such as this can always be constructed and is guaranteed to detect any fault in the IUT other than those that increase the number of states or input signals. Algorithm 9.6 assumes the existence of a reliable reset message. Yannakakis and Lee have shown that for a machine without

a reset message there still exists a polynomial length conformance test with the same fault coverage as Algorithm 9.6 (see Bibliographic Notes). No polynomial time algorithm is known, however, to derive such a test sequence for a reset-less machine.

9.7 SUMMARY

A conformance test is designed to verify whether an unknown implementation of a protocol can be considered to be equivalent to a known specification. The test can never produce an answer that is completely reliable. Only the presence of desirable behavior can be tested for, not the absence of undesirable behavior. It is therefore always possible that an implementation is capable of responses that are not part of the specification.

In principle, there are two approaches to the conformance testing problem. One is a rather *ad hoc* approach where, by trial and error, the correct provision of the main protocol functions is verified for as broad a range of parameter values as possible. For example, if the purpose of the protocol is connection management, we can test randomly chosen sequences for connection setup and tear-down. A second method is to systematically probe the implementation with test sequences to establish whether its internal structure, seen as a finite state machine, conforms to the structure of the specification. Parameter and data values are not considered in this type of test. Instead the focus is on the control structure of the protocol proper. Most of the progress in the development of protocol conformance testing tools has been made with the second type of testing.

Not surprisingly, an effective conformance test can be greatly facilitated if specification and implementation were developed with the feasibility of a test in mind. A systematic conformance test is only possible if the IUT has at least the three properties listed in Section 9.3. A particularly simple algorithm (Algorithm 9.1) can be applied if, in addition, the IUT has a reset, status, and set transition on every state. The hard work in conformance testing comes when one or more of the desirable properties are missing.

Without the set property, the best method is to find a transition tour of all states using Algorithms 9.2 and 9.3, testing the response of the IUT to all possible input signals. A status message can be replaced by a test sequence, called a UIO sequence, that can similarly reveal the state of the machine. Algorithm 9.4 can be used to derive these UIO sequences. The main problem to be solved here is that the UIO sequences disturb the state of the IUT when they are applied. Section 9.5 shows how a transition tour can be constructed that avoids this problem.

An alternative method, that also avoids the use of status and set messages is discussed in Section 9.6. All methods discussed can fail when an implementation error in the IUT increases the number of reachable states or the number of input signals. In the absence of such errors, the conformance testing sequence produced by Algorithm 9.6 can guarantee the detection of all faults. The length of the test sequence, however, can be considerably larger than the one based on UIO sequences.

EXERCISES

9-1. Explain in detail why it is essential that each of the first three requirements on the structure of an IUT be fulfilled for a conformance test to be feasible. □

9-2. Is it necessary that the finite state machine modeled by the IUT has been minimized? □

9-3. How will the algorithms in this chapter have to be changed for incompletely specified finite state machines? □

9-4. Is the symmetric augmentation of Figure 8.1 shown in Figure 9.4 unique. If not, is it an augmentation with the lowest cost? □

9-5. Can a conformance test detect whether the IUT has more states than the formal specification? Fewer states? □

9-6. Can the fault coverage of a conformance test ever reach 100%? □

9-7. Explain the difference between a distinguishing sequence, a homing sequence, a characterizing sequence, and a UIO sequence. □

9-8. How would the fault coverage of a conformance test be affected if the application of a status message, or its equivalent, were deleted from a transition tour? □

9-9. Write an algorithm for finding homing sequences. □

9-10. Write an algorithm for finding characterizing sequences. □

9-11. How does the cost of the above two algorithms (the number of operations to be performed in the worst case) depend on the number of edges and vertices in the graph? Can you derive upper bounds? □

BIBLIOGRAPHIC NOTES

Conformance testing methods are of interest to all protocol users who want to assess the quality of protocol implementations. They are also of interest to international standardization bodies, who aim to provide a neutral third party certification of protocols and protocol implementation. Organizations such as the ISO and the CCITT are in the process of developing standards and guidelines for the certification of protocols meant to comply with, for instance, the reference model for Open Systems Interconnection discussed in Chapter 2, Rayner [1987].

The general conformance testing problem, i.e., the problem of testing implementations of arbitrary extended finite state machines, using remote testers across a data network, is an active research area. Progress is reported in the yearly IFIP Working Group 6.1 Symposia on Protocol Specification, Testing and Verification, IFIP [1982-present]. An excellent overview of the general problem can be found in Rayner [1987]. Work is also underway to standardize a notation for conformance test suites, named the *Tree and Tabular Combined Notation* or TTCN, see ISO [1987].

One of the first efforts to develop an independent center for the assessment and certification of protocol implementations was begun by the National Physical Laboratory (NPL) in England, in the early 1980s, Rayner [1982, 1987]. In the U.S.A., this work was undertaken by the National Institute of Science and Technology (NIST, formerly the National Bureau of Standards or NBS), Nightingale [1982]. An overview of other test and certification centers is given in Wang and Hutchinson [1987].

Two issues complicate the work of the certification centers. First, the certification centers should be able to perform remote testing of implementations, across a trusted data network. Second, the certification tests sometimes have to be applied to a single protocol layer in a hierarchy of otherwise trusted layers.

The certification centers have concentrated mainly on service, or functional, conformance testing. Structural testing, as described here, is a more recent development. The conformance testing work we have described is based on both finite state machine theory and on graph theory. The concept of a test sequence was studied as early as 1956 by E.F. Moore in one of his first papers on finite state machines, Moore [1956]. Moore was also the first to define homing sequences and distinguishing sequences. The concept of a distinguishing sequence was further developed in Gill [1962] and in Hennie [1964]. Huffman independently studied problems similar to those in Moore [1956]. His results can be found in Huffman [1964]. A complete discussion of homing sequences and characterizing sequences can be found in Kohavi [1978].

Naito [1981] and Sarikaya [1984] were among the first to study systematic protocol test generation techniques using transition tours. The concept of a UIO sequence, was introduced in Hsieh [1971], and was discussed in Friedman and Menon [1971]. Independently it was also discovered by Gobershtein [1974]. K.K. Sabnani and A.T. Dahbura [1985, 1988] rediscovered the principle and applied it to the conformance testing problem. The term UIO sequence was coined by them. Hsieh used the term *simple I/O* sequence; Gobershtein used *check word*. Yannakakis and Lee [1990] introduced the term *state signature*.

A method to reduce the length of a conformance test sequence by computing multiple UIO sequences per state is described in Shen and Lombardi [1989].

COMPLEXITY

The problem of finding the minimum length transition tour of a finite state machine, described for instance in Klee [1980], can be solved in polynomial time. Algorithm 9.2 comes from Edmonds and Johnson [1973] and was applied to the conformance testing problem in Uyar and Dahbura [1986]. A symmetric augmentation of a graph can also be found in polynomial time with network flow algorithms. An Euler tour can be found in a time that is linear in the number of transitions in the symmetric graph. The corresponding algorithm can be found in Edmonds and Johnson [1973].

The problem of finding a transition tour through a subset of the transitions in a graph, for instance the pseudo-transitions corresponding to UIO sequences in a modified transition tour, can be shown to be NP-complete. If, however, the graph consisting of pseudo-transitions is weakly connected, the problem reduces to one that can be solved in polynomial time, as shown in Aho, Dahbura, Lee and Uyar [1988]. Efficient algorithms for the derivation of transition tours for restricted classes of finite state machines are studied in Edmonds and Johnson [1973]. The algorithms were first applied to protocol conformance testing in Uyar and Dahbura [1986] and extended in

Aho, Dahbura, Lee and Uyar [1988]. The Chinese Postman Problem was first described by the Chinese mathematician M-K. Kuan [1962].

In general, the cost of traversing a transition in the finite state machine machine can be given by a real number. A well-known algorithm for finding the shortest distance (i.e., lowest cost path) between two vertices in a graph, given those constraints, is Dijkstra's shortest path algorithm, e.g., Dijkstra [1959], Aho [1974], Price [1971]. There are, however, also faster methods, see for instance Tarjan [1983]. In conformance testing it is often possible to associate simply a unit cost with the traversal of all transitions. In this case, the best algorithm for finding shortest paths is breadth-first search.

Efficient solutions to minimum-cost — maximum-flow problems are also discussed in detail in Tarjan [1983] and Gibbons [1985]. An overview can be found in Aho, Dahbura, Lee and Uyar [1988].

Algorithm 9.6, and the analysis of its complexity, was described in Yannakakis and Lee [1990]. It is similar to the W-method from Chow [1978]. A proof of the PSPACE hardness of the UIO derivation problem can also be found in Yannakakis and Lee [1990].

An issue not discussed here is the problem of estimating the quality or fault coverage of a conformance test. Note that, despite the first three requirements from Section 9.3, a faulty IUT may well have more states or more inputs than our reference specification. In Vasilevskii [1973] it was shown that a checking sequence becomes inherently exponential if faults increase the number of states. The machine may also have nondeterministic responses, or it may not be strongly connected, as required. There is always an infinite number of such implementations that can pass a given conformance test without being equivalent to the reference specification. In theory, therefore, the fault coverage of every conformance test of the type we have described must approach zero. Yet, the relative fault coverage of individual conformance test methods may well differ. Empirical methods to measure such differences are illustrated in Dahbura and Sabnani [1988] and Sidhu and Leung [1989]. By randomly modifying transitions in a finite state machine description of the IUT it can be measured what percentage of this restricted class of errors is caught by a conformance testing method. As yet, however, such tests have only been used successfully to confirm results that can also be proven theoretically.

Not discussed in this chapter are two alternative methods for conformance testing that have been explored. The W-method was introduced in Vasilevskii [1973] and elaborated in Chow [1978]. The method is also explained in Shih and Sidhu [1986]. The second method is based on the use of grammars to generate test sequences and was explored in Linn and McCoy [1983], and Probert and Ural [1983]. A general overview of test methods is given in Sidhu [1990]. An interesting formal study of the conformance testing problem is described in Brinksma et al. [1989].