CHAPTER 5

TESTING FROM FINITE STATE MACHINES

This chapter describes the simplest kind of model-based testing, which is based on finite state machines (FSMs). Each node of an FSM corresponds to a particular state of the SUT and each arc corresponds to an SUT action, so to generate test sequences we can just traverse the FSM.

We start with simple FSM models that are suitable for testing passive systems, such as unit testing of an object or class, and for testing deterministic reactive systems, where events occur in a known order. We use a simple FSM model to test the Qui-Donc service of France Telecom. Then we extend that to *extended finite state machines* (EFSM), which increase the expressiveness of FSMs by adding variables, state updating commands, and transition guards. We use a parameterized EFSM model to do unit testing of a hierarchy of Java classes.

Chapter 7 extends the ideas discussed in this chapter by using Unified Modeling Language (UML) state machines, which support nested machines and parallel machines.

5.1 TESTING QUI-DONC WITH A SIMPLE FSM

Qui-Donc (literal translation, "Who then?") is a service provided by France Telecom that does the opposite of the white pages—it allows you to enter a telephone number and find out the name and address associated with the number. It can be very useful in conjunction with caller ID! It is a vocal service, which you use by dialing the telephone number 08 92 68 01 11 and then listening to the vocal messages and pressing buttons on the keypad of your phone. Of course, the Qui-Donc service is actually all in French, but we've translated it into English for those readers who *ne parlent pas le français*.

Let's assume that we have been contracted to do some independent testing of the Qui-Donc system, to check that its behavior matches its requirements. The next section shows the informal requirements that we have been given. In Section 5.1.2 we develop a simple FSM model of the expected behavior of the Qui-Donc system, and in Section 5.1.3 we generate tests from that FSM. Since the outputs of the Qui-Donc system are all spoken phrases and we want to use a real telephone as our input device, we decided not to try to automate the execution of the tests. Instead, we will aim to choose test generation techniques that produce a smallish set of tests so that manual execution of the tests is practical.

5.1.1 Informal Requirements

After the caller dials 08 92 68 01 11, the Qui-Donc system responds by giving the WELCOME message, "Welcome to Qui-Donc. Please press the star key." This message is repeated up to three times, separated by a timeout of 6 seconds. If no star key is detected during this whole process, then the NOTALLOW message is given: "Your telephone does not allow access to the Qui-Donc service," and the call is disconnected.

If a star key is detected during the above process, then the ENTER message is given: "Please enter the 10-digit telephone number of the subscriber you are looking for, followed by the pound key." This message is repeated up to three times, but with a longer timeout of 20 seconds; if no numbers and hash, or pound (#), key are entered, then the call is terminated with the BYE message: "Thank you for using the Qui-Donc service."

If some digits followed by a pound key were detected, and the number was one of the emergency numbers, 15, 17, or 18 (for ambulance, police, and fire brigade, respectively), then an explanation of that emergency number is given, followed by the message: "If you want to do another search, press star." If the entered number did not contain 10 digits and was not one of the emergency numbers, then the ERROR message is given: "Invalid number

entered. Please enter another number, followed by the pound key." However, if the entered number was 10 digits, then it is looked up in the white pages database. If the number is not in the database, then the SORRY message is given: "Sorry. The number NN NN NN NN NN does not appear in the white pages," and the caller goes back to the ENTER process.

If the entered number is in the database, the NAME message is given: "The number NN NN NN NN NN corresponds to SURNAME, INITIALS," then the Qui-Donc system enters the *information* menu.

The information menu gives the INFO message, "Press 1 to spell the name, press 2 to hear the address, or press star for a new search." If 1 is entered, then the name is spelled out and it returns to the information menu. If 2 is entered, then the address is given and it returns to the information menu. If star is entered, then it goes to the number-entry process, starting with the ENTER message. If no keys are pressed, then the INFO message is repeated after a timeout of 6 seconds; and after this message has been given three times without response, the call is terminated with the BYE message.

In addition to these processes, note that the caller can hang up at any moment.

5.1.2 Modeling Qui-Donc with an FSM

The first step in writing any model is always to decide what to *ignore*. That is, we need to take a very *abstract* view of the system to keep our FSM small enough. After all, there are 10^{10} possible 10-digit phone numbers—we cannot test all of those, and we certainly do not want to create an FSM that big! The kinds of fault that we want to find in the Qui-Donc system are logical faults, such as an incorrect message being given or the wrong menu being presented. We are not intending to test the contents of its database, which is all the numbers, names, and addresses in the white pages.

So, for testing purposes we decide that we will test just the four keys on the phone that Qui-Donc treats as special inputs (`1`, `2`, `*` and `#`), plus four representative telephone numbers for when Qui-Donc asks for a telephone number:

18 This is the emergency number for the fire brigade. We make a *uniformity hypothesis* that if one of the emergency numbers behaves correctly, then all of them will, so testing one of them will be enough.[1]

[1] If Qui-Donc were a safety-critical application, we would not make this kind of assumption about the emergency numbers; rather we would want to test the set of emergency numbers exhaustively.

num1 (03 81 11 11 11). When we ring this number, it seems to be disconnected, so we assume that this number is not in the white pages database.

num2 (03 81 22 22 22). We found this number in the white pages by choosing *K. J. Renard* at random. His address is *45 rue de Vesoul, Besançon*, and we assume that this information is in the white pages database.

bad (12 34 56 78 9). The bad number is a number with just fewer than 10 digits. Here, we make another uniformity hypothesis, assuming that this number is a representative sample for all possible numbers that contain fewer than 10 digits or more than 10 digits. Thinking about boundary testing, it would be good to have an 11-digit input as well, but we leave this as an exercise for the reader.

So the *input alphabet* of our model (the set of all the different inputs that we will send to the Qui-Donc system) is:

```
{dial,num1,num2,bad,18,1,2,*,#,wait}
```

Let's just explain how these will relate to the real-world test inputs. The `dial` input means to pick up the phone (after resetting it by hanging up, if necessary), dial the Qui-Donc service (08 92 68 01 11), and wait for a response. The single digits, `1` and `2`, and the `*` and `#` inputs mean that the person executing the tests should just press that key and wait for the Qui-Donc system to respond. The `18` means press 1, then 8, then the # key, taking a total of less than 6 seconds to do this. The `num1` input means press all 10 digits of that number (0, 3, 8, 1, 1, 1, 1, 1, 1, 1) followed by the # key and do all this in just less than 20 seconds.

Similarly for `num2` (but to be different, we will enter num2 as quickly as we can), and for the `bad` input, which has only 9 digits (1, 2, 3, 4, 5, 6, 7, 8, 9, #). The `wait` input means wait, without pressing any keys, until the Qui-Donc system does something—it should time out after 20 seconds for the `ENTER` states and after 6 seconds for the other states. (For simplicity, we have not bothered to distinguish between these two timeouts in our model, but this could easily be done by annotating some of the output messages with the length of the pause that is expected before the output message is produced—for example, by having several different `ENTER` outputs, like $\texttt{ENTER}_0$, $\texttt{ENTER}_6$, and $\texttt{ENTER}_{20}$, so that the person executing the test knows exactly how long to wait before expecting the `ENTER` message.)

So, given a sequence of abstract input values like this,

```
dial, wait, *, Num1, 2, wait, wait, wait
```

our manual tester will know exactly what to press, and when. For this particular sequence of input stimuli, the resulting sequence of messages from the Qui-Donc system should be:

```
WELCOME, WELCOME, ENTER, NAME+INFO, ADDR, INFO,
INFO, BYE (it hangs up).
```

Rather than write each test case with its input sequence and expected output sequence separated like this, we usually write it as a sequence of input/output pairs:

```
dial/WELCOME, wait/WELCOME, */ENTER, num1/NAME+INFO,
2/ADDR, wait/INFO, wait/INFO, wait/BYE  (it hangs up).
```

Each transition of our finite state machine will be labeled with a single input/output pair like this. This means that our FSM is a *Mealy machine* [LY96], which is the most common kind of FSM used to specify software and hardware systems. In a Mealy machine, each transition is labeled with an input i and an output o. When used for testing purposes, each transition defines one step in a test case—the input i is the stimulus that is sent to the SUT, and the output o is the expected response of the SUT. We sometimes write a transition from state s to state s' as $s \xrightarrow{i/o} s'$.

When designing your FSM model, you may find that some transitions seem to need to be labeled with multiple outputs, or zero outputs, because your SUT does not always produce exactly one response for each input. For example, the Qui-Donc requirements show that the `num1` input can result in our hearing two messages, the `NAME` message immediately followed by the `INFO` prompt message. In this case, we must package up those two outputs and model them as one single output message. You may also come across situations where your SUT does not produce any output for a given input/– this can be modeled as a transition labeled with something like "*input*/−," where the output symbol "−" just means "expect no response from the SUT."

The important thing about Mealy machine models is that every transition must be labeled with exactly one input. It is not acceptable to have transitions with no inputs because it would mean that the SUT could produce outputs spontaneously rather than being under the control of the test case. Later in the book (e.g., Sections 6.5 and 7.3), we discuss more powerful modeling notations that can be used for those kinds of *nondeterministic reactive systems*.

Key Point A Mealy machine model has a finite set of states, plus many transitions that go from one state to another. Each transition must be labeled with exactly one input and one output (which may be "–"). The machine must have one *initial* state and may optionally have one or more *final* states.

Each test generated from a Mealy machine FSM model is a sequence of transitions, which starts from the initial state and ends in one of the final states. If there are no final states, then the test case is allowed to end in any state. If your SUT requires a complete sequence of events (e.g., login, do some transactions, then logout), then adding a final state (after the logout transition) allows you to force all the generated test sequences to be complete.

Figure 5.1 shows the finite state machine that we can design from these requirements and assumptions. It is written graphically as a state transition diagram, and its output messages are listed in Table 5.1. Note that several of them are the concatenation of an informational message followed by the `ENTER` or `INFO` prompt, so the prompt strings are repeated in several messages. In this model, we have one final state, which is the same as our initial state. This means that we want all of the generated tests to end up in the `Start` state, ready to commence a new call.

Does this finite state machine seem to you to be quite complex for such a simple system? We agree. This is because the three levels of timeout create a lot of duplication and make the diagram rather cluttered. We will see later in this chapter that the use of *extended finite state machines* (EFSMs) can eliminate this kind of duplication and dramatically simplify the FSM by adding a variable to count the number of timeouts.

However, things may get worse before they get better because the FSM in Figure 5.1 is not even *complete* yet. That is, it has some states that do not have a transition for *every possible input*. Our input alphabet contains 10 symbols, so every state of our FSM should have 10 transitions leaving it, except the `Start` state, where we decide that `dial` is the only meaningful input because we are not yet connected to the Qui-Donc system. For example, the `Star1` state does not specify what should happen if a number is pressed rather than `*`, and the `Info1` state does not specify what happens if a phone number is entered. It is so tedious to specify all these transitions that it is usual to follow the convention that unexpected inputs are simply *ignored*, leaving the state unchanged. This effectively completes the FSM by adding implicit loop transitions from each state back to itself for each unspecified input. If we showed all those extra transitions on the diagram, it would be too cluttered to read.

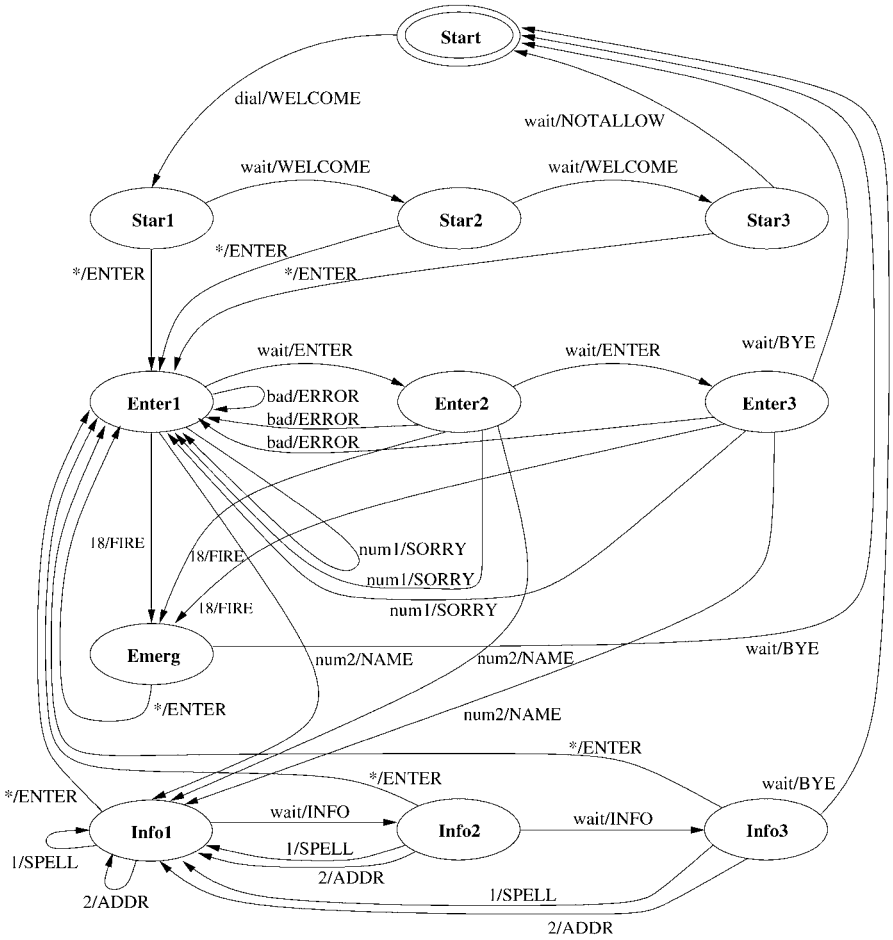

FIGURE 5.1 Qui-Donc FSM model.

Table 5.2 shows the same FSM written as a state table, with one column for each input symbol. This is a good representation for seeing missing transitions. The entries in italics show the transitions that we would need to add to make the FSM complete. The `dial` column models the effect of hanging up in the middle of a transaction and redialing the Qui-Donc system (it would be possible to divide this into separate `hangup` and `dial` inputs, but we have merged them for simplicity). The remaining italic entries show the effect of the *ignore unexpected inputs* convention—every missing transition is assumed to leave the state unchanged. If we add all these italicized transitions, the FSM is complete (except for the `Start` state) because each state has a transition for every input.

TABLE 5.1 The Output Alphabet of the Qui-Donc Model

Output Symbol	Concrete Message
WELCOME	"Welcome to Qui-Donc. Please press the star key."
NOTALLOW	"Your telephone does not allow access to the Qui-Donc service."
ENTER	"Please enter the 10-digit telephone number of the subscriber you are looking for, followed by the pound key."
ERROR	"Invalid number entered. Please enter another number, followed by the pound key."
FIRE	"18 is the emergency number for the fire brigade. If you want to do another search, press star."
SORRY	"Sorry. The number 03 81 11 11 11 does not appear in the white pages. Please enter the 10-digit telephone number of the subscriber you are looking for, followed by the pound key."
NAME	"The number 03 81 22 22 22 corresponds to Renard, K. J. Press 1 to spell the name, press 2 to hear the address, or press star for a new search."
INFO	"Press 1 to spell the name, press 2 to hear the address, or press star for a new search."
SPELL	"Renard is spelled R, E, N, A, R, D. Press 1 to spell the name, press 2 to hear the address, or press star for a new search."
ADDR	"The address of Renard, K. J. is 45 rue de Vesoul, Besançon. Press 1 to spell the name, press 2 to hear the address, or press star for a new search."
BYE	"Thank you for using the Qui-Donc service."

Another common way to represent an FSM is as a set of transitions, written as (CurrentState, Input, Output, NextState) quadruples.

```
(Start,  dial,  WELCOME,  Star1 )
(Star1,  wait,  WELCOME,  Star2 )
(Star1,  *,     ENTER,    Enter1)
(Star2,  wait,  WELCOME,  Star3 )
(Star2,  *,     ENTER,    Enter1)
. . .
```

Whichever representation we use, after we have finished designing our model we should spend a little time validating it. The list on page 148 shows some of the properties that we could check.

TABLE 5.2 Qui-Donc FSM Model as a State Table

Input	dial	num1	num2	bad	18	1	2	*	#	wait
Start	WELCOME Star1									
Star1	*WELCOME* *Star1*	– *Star1*	– *Star1*	– *Star1*	– *Star1*	– *Star1*	– *Star1*	ENTER Enter1	– *Star1*	WELCOME Star2
Star2	*WELCOME* *Star1*	– *Star2*	– *Star2*	– *Star2*	– *Star2*	– *Star2*	– *Star2*	ENTER Enter1	– *Star2*	WELCOME Star3
Star3	*WELCOME* *Star1*	– *Star3*	– *Star3*	– *Star3*	– *Star3*	– *Star3*	– *Star3*	ENTER Enter1	– *Star3*	NOTALLOW Start
Enter1	*WELCOME* *Star1*	SORRY Enter1	NAME Info1	ERROR Enter1	FIRE Emerg	– *Enter1*	– *Enter1*	– *Enter1*	– *Enter1*	ENTER Enter2
Enter2	*WELCOME* *Star1*	SORRY Enter1	NAME Info1	ERROR Enter1	FIRE Emerg	– *Enter2*	– *Enter2*	– *Enter2*	– *Enter2*	ENTER Enter3
Enter3	*WELCOME* *Star1*	SORRY Enter1	NAME Info1	ERROR Enter1	FIRE Emerg	– *Enter3*	– *Enter3*	– *Enter3*	– *Enter3*	BYE Start
Emerg	*WELCOME* *Star1*	– *Emerg*	– *Emerg*	– *Emerg*	– *Emerg*	– *Emerg*	– *Emerg*	ENTER Enter1	– *Emerg*	BYE Start
Info1	*WELCOME* *Star1*	– *Info1*	– *Info1*	– *Info1*	– *Info1*	SPELL Info1	ADDR Info1	ENTER Enter1	– *Info1*	INFO Info2
Info2	*WELCOME* *Star1*	– *Info2*	– *Info2*	– *Info2*	– *Info2*	SPELL Info1	ADDR Info1	ENTER Enter1	– *Info2*	INFO Info3
Info3	*WELCOME* *Star1*	– *Info3*	– *Info3*	– *Info3*	– *Info3*	SPELL Info1	ADDR Info1	ENTER Enter1	– *Info3*	BYE Start

Deterministic: An FSM is *deterministic* if for every state, every transition out of that state is labeled with a different input. This is a crucial property for many FSM algorithms because otherwise, it is ambiguous which transition we should take out of that state.

Initially connected: An FSM is *initially connected* if every state is reachable from the initial state. If your FSM is not initially connected, then you have almost certainly made an error because it means that part of your FSM is unusable and cannot be tested. It is usually easy to see if an FSM is initially connected by a quick visual inspection of the FSM diagram.

Complete: An FSM is *complete* if for each state, the outgoing transitions cover all inputs. As mentioned, it is easy to make an FSM complete by adding all the self-transitions $s \xrightarrow{i/-} s$ for every state s that does not have an outgoing transition with an input i.

Minimal: An FSM is *minimal* if it has no redundant states. That is, it does not have two distinct states that generate the same set of input/output sequences. A nonminimal FSM is not a major problem; it just means that you could have designed a simpler FSM that was equivalent. There are efficient algorithms for transforming a nonminimal FSM into a minimal equivalent FSM [Hop71].

Strongly connected: An FSM is *strongly connected* if every state is reachable from every other state. This means that there are no states or loops that you can get stuck in—there is always a path out to the rest of the FSM. This is a strong restriction and not always desirable. However, if the SUT has a *reset* method that sets it back to the initial state (like hanging up the phone in the Qui-Donc example), then the FSM is automatically strongly connected whenever it is initially connected.

Our Qui-Donc FSM in Figure 5.1 is deterministic, initially connected, strongly connected, and minimal. This is good. Now that we have a model of the expected behavior of the Qui-Donc system, we consider how to generate tests from this model.

5.1.3 Generating Tests

We now show a series of simple techniques for generating tests from our Qui-Donc model, starting with some that generate small and rather inadequate test suites and progressing to techniques that generate such large and comprehensive test suites that they are rarely practical. We focus initially on generating tests from just the *explicit* Qui-Donc model that contains only

the transitions shown in Figure 5.1, rather than from the *complete* model that also includes all the implicit transitions (italicized) shown in Table 5.2.

STATE, INPUT, AND OUTPUT COVERAGE

State coverage is a simple and popular coverage criterion. It measures the percentage of FSM states that are visited during the test execution. For example, here is a minimal length test suite (1 test, with just 12 transitions) that gives us 100 percent state coverage:

```
dial/WELCOME, wait/WELCOME, wait/WELCOME,
*/ENTER, wait/ENTER, wait/ENTER, 18/FIRE, */ENTER,
num2/NAME, wait/INFO, wait/INFO, wait/BYE.
```

As you can see, this is a very weak test suite. It does not test any bad numbers or unknown numbers and does not even ask for the spelling or address information.

Input coverage is another simple coverage metric. It measures how many distinct input symbols have been sent to the SUT. For example, here is a test sequence that gives us 90 percent coverage of our 10 possible inputs.

```
dial/WELCOME, */ENTER, bad/ERROR, num1/SORRY, num2/NAME,
1/SPELL, 2/ADDR, */ENTER, 18/FIRE, wait/BYE.
```

This is an even shorter test than the previous one, and it does not test the timeout behavior at all, so it is a very weak test suite.

Why didn't we get 100 percent input coverage? Because, surprisingly, the remaining input # does not appear on any transitions of our explicit FSM model (Figure 5.1). Failing to reach 100 percent coverage of a simple coverage metric like input coverage can be a useful warning sign that we have forgotten something in our model. However, in this case, our reason for including # in the set of inputs was just so that we could perform some robustness testing using the complete FSM model. So it is acceptable that we cannot reach 100 percent coverage in the explicit model. It is easy to reach 100 percent input coverage in the complete model—just add the transition #/— after the `dial/WELCOME` transition.

It is possible to define *output coverage* in a similar way to input coverage, so that it measures how many distinct output responses have been received from the SUT. For example, the short test that covers 9/10 of the inputs also covers 9/11 of the possible outputs. We have seen little discussion of output coverage in the testing literature, but it is similar to the *cause-effect* heuristic [Mye79], which says to try to test the inputs that cause the SUT to produce different responses/effects/outputs.

All these coverage metrics are too easily satisfied, so even 100 percent state coverage, input coverage, or output coverage gives us no confidence

that we have generated a reasonably comprehensive test suite. In fact, the goal of this section is to persuade you to never be satisfied with just these weak metrics! Let us now investigate a stronger coverage, which covers every transition.

TRANSITION COVERAGE

Transition coverage measures how many of the transitions of the FSM have been tested. So 100 percent transition coverage means that every transition has been tested at least once. There are many ways to generate a test suite that has 100 percent transition coverage. For example, a random path through the FSM will eventually cover all the transitions.[2] However, the best way to generate the smallest possible test suite that has 100 percent transition coverage is to do a *transition tour* of the FSM. A transition tour is a minimum-length circular path through the FSM that visits every transition at least once.

The best way to generate a transition tour of an FSM is to use the *Chinese postman algorithm*, which was invented by the Chinese mathematican Guan Mei Gu [Gua62]. It finds the shortest path through a graph that visits every edge.[3] Thimbleby [Thi03] gives an excellent description of the algorithm and its applications and includes a full Java implementation.[4]

When we apply that algorithm to our Qui-Donc FSM, we get the tour shown in Listing 5.1, which dials up the Qui-Donc service just four times and contains a total of 61 steps. This is a very nice test suite, which exercises all the states, all the inputs, all the outputs, and all the transitions of our model, yet would take only about 15 minutes to execute (assuming that each transition takes an average of 15 seconds and to tap one or two keys and then listen to the response).

Transition coverage is stronger than state coverage because if we have tested all the transitions, we must also have visited every state at each end of those transitions (we assume that the FSM is *initially connected*). So if you use a transition-tour test generation algorithm, you not only have 100 percent transition coverage, but you also get 100 percent state coverage. State coverage is analogous to statement coverage in programming languages, and

[2] We assume here that the FSM is initially connected so that all states are reachable from the initial state. If the FSM graph is not also strongly connected, then the random path generator should occasionally perform a *reset* operation to return to the initial state and ensure that all transitions are reachable.

[3] Compare this with the well-known traveling salesman problem, which visits every node of the graph, but not necessarily every edge. A traveling salesman algorithm gives 100 percent state coverage of an FSM, but not necessarily 100 percent transition coverage.

[4] His Java implementation can be downloaded from *http://www.uclic.ucl.ac.uk/harold/cpp*.

LISTING 5.1 Transition tour of the Qui-Donc partial FSM (4 tests with a total of 61 transitions).

```
1. dial/WELCOME, */ENTER, wait/ENTER, num1/SORRY, wait/ENTER,
   bad/ERROR, wait/ENTER, 18/FIRE, */ENTER, num2/NAME,
   wait/INFO, */ENTER, wait/ENTER, num2/NAME, wait/INFO, 2/ADDR,
   wait/INFO, 1/SPELL, wait/INFO, wait/INFO, */ENTER, wait/ENTER,
   wait/ENTER, num1/SORRY, wait/ENTER, wait/ENTER, bad/ERROR,
   wait/ENTER, wait/ENTER, 18/FIRE, */ENTER, wait/ENTER,
   wait/ENTER, num2/NAME, wait/INFO, wait/INFO, 2/ADDR,
   wait/INFO, wait/INFO, 1/SPELL, 2/ADDR, 1/SPELL, wait/INFO,
   wait/INFO, wait/BYE
2. dial/WELCOME, wait/WELCOME, */ENTER, num1/SORRY, bad/ERROR,
   wait/ENTER, wait/ENTER, wait/BYE
3. dial/WELCOME, wait/WELCOME, wait/WELCOME, */ENTER, num2/NAME,
   */ENTER, 18/FIRE, wait/BYE
4. dial/WELCOME, wait/WELCOME, wait/WELCOME, wait/NOTALLOW,
```

transition coverage is analogous to branch coverage. In the case where every input appears somewhere in your explicit FSM model, then a transition tour also gives 100 percent input coverage. It works similarly for output coverage.

If we had included all the `dial` transitions in our FSM model to model the fact that from any state we can hang up the phone and then redial the Qui-Donc number to start a new session (these transitions are shown in the leftmost column of Table 5.2), then the transition tour would contain 14 tests, with a total of 89 transitions. This test suite extends the previous test suite by also testing that the Qui-Donc system resets itself correctly when a caller hangs up unexpectedly.

More interesting is Listing 5.2, which shows the test suite generated by a transition tour of the (almost) *complete* version of our FSM model, which includes all the default transitions shown in italics in Table 5.2, except for the italicized `dial` transitions. This test suite tries all the illegal and unexpected input values in each state, so it tries to test that the Qui-Donc system does indeed ignore all inputs that are not mentioned in its requirements.

However, if we execute this test suite, we find a few surprises (see the transitions in boxes in Listing 5.2):

- The 16th transition in test 1 has an input of `#` (from state `Enter2`), and we find that this is not ignored by Qui-Donc; because it is interpreted as a telephone number with zero digits, it produces the `ERROR` output. In fact, this is the correct behavior, so we realize that we have an error in

LISTING 5.2 Transition tour of the Qui-Donc complete FSM (4 tests with a total of 126 transitions).

```
1. dial/WELCOME, #/-, 2/-, 1/-, 18/-, bad/-, num2/-, num1/-,
   dial/-, */ENTER, wait/ENTER, num1/SORRY, wait/ENTER,
   bad/ERROR, wait/ENTER, #/-, */-, 2/-, 1/-, dial/-, 18/FIRE,
   */ENTER, num2/NAME, wait/INFO, */ENTER, wait/ENTER, num2/NAME,
   wait/INFO, 2/ADDR, wait/INFO, 1/SPELL, wait/INFO, wait/INFO,
   */ENTER, wait/ENTER, wait/ENTER, num1/SORRY, wait/ENTER,
   wait/ENTER, bad/ERROR, wait/ENTER, wait/ENTER, #/-, */-,
   2/-, 1/-, dial/-, 18/FIRE, */ENTER, wait/ENTER, wait/ENTER,
   num2/NAME, wait/INFO, wait/INFO, 2/ADDR, wait/INFO,
   wait/INFO, 1/SPELL, #/-, 18/-, bad/-, num2/-, num1/-,
   dial/-, 2/ADDR, 1/SPELL, wait/INFO, #/-, 18/-, bad/-, num2/-,
   num1/-, dial/-, wait/INFO, #/-, 18/-, bad/-, num2/-, num1/-,
   dial/-, wait/BYE
2. dial/WELCOME, wait/WELCOME, #/-, 2/-, 1/-, 18/-, bad/-,
   num2/-, num1/-, dial/-, */ENTER, #/-, */-, 2/-, 1/-, dial/-,
   num1/SORRY, bad/ERROR, wait/ENTER, wait/ENTER, wait/BYE
3. dial/WELCOME, wait/WELCOME, wait/WELCOME, #/-, 2/-, 1/-, 18/-,
   bad/-, num2/-, num1/-, dial/-, */ENTER, num2/NAME, */ENTER,
   18/FIRE, #/-, 2/-, 1/-, 18/-, bad/-, num2/-, num1/-,
   dial/-, wait/BYE
4. dial/WELCOME, wait/WELCOME, wait/WELCOME, wait/NOTALLOW
```

our model—the three Enter$_i$ states should each have a transition #/ERROR that goes back to the Enter1 state.

- The input 18, when in state Info$_i$, is not ignored because it contains a 1, which triggers the 1/SPELL transition. This is a consequence of our concrete test values—one input happens to be a prefix of another.
- The input bad, when in state Info$_i$, also triggers the 1/SPELL transition because the bad number is 81123456, which contains a 1. Similarly for the num1 and num2 inputs in these states.

We mention these "surprises" to illustrate that it is often very dangerous and error-prone to make a general assumption about the behavior of all omitted transitions in an FSM model, such as assuming that all unspecified inputs are ignored. Binder discusses this in more detail [Bin99, 223–228] and recommends that an explicit response matrix, similar to Table 5.2, be used to record the precise response to every input event.

In conclusion, a transition tour of the explicit FSM generates a useful test suite that gives 100 percent transition and state coverage (and usually input and output coverage too). A transition tour of the complete FSM generates a larger test suite that also does robustness testing by trying every possible input in each state. This is useful for detecting *sneak paths* (extra transitions) in the SUT.

We recommend that you treat transition coverage as your basic coverage metric when doing model-based testing from FSMs; you should expect to achieve 100 percent transition coverage of your explicit FSM model and preferably 100 percent transition coverage of the complete FSM model.

Of course, even 100 percent transition coverage is no guarantee that you have found all the errors in the SUT! It is quite possible that a particular sequence of three transitions would have exposed an SUT fault, but that particular sequence did not happen to be exercised during the transition tour. Also, the FSM model usually just describes part of the SUT functionality, so even a huge test suite generated from the FSM is unlikely to help in finding errors that are outside the scope of the model.

Key Point Full transition coverage is a good minimum to aim for when generating tests from FSM models.

EXPLICIT TEST CASE SPECIFICATIONS

It may seem a bit cold and mechanical, or perhaps just naive, to assume that a simple algorithm like transition tour is capable of generating a good test suite. Is there no longer any need for the skill, the knowledge, the passion, and the domain expertise of an experienced test engineer?

Well, of course, the test engineer remains central to the whole process of model-based testing. He or she designed the model, must decide which test generation algorithms and heuristics to apply, and should inspect the results of testing (faults found, code coverage levels, model coverage levels, etc.) to decide how much more testing is necessary.

But it can also be useful for test engineers to directly use their domain knowledge and testing intuition to help the model-based testing tools to choose certain interesting tests. One way to do this is for the test engineer to supply *explicit test case specifications*, which ask for a particular kind of test to be generated. This means that the high-level design and the rationale for the test come from the engineer, while the low-level details of the test and the expected SUT outputs come from the model.

For example, we may have the goal of testing whether the Qui-Donc system can be used by disabled or elderly people, who are slow at entering

numbers and frequently fail to complete the input before the timeout expires. We could specify these kinds of tests by asking a tool to generate a test that traverses the `Star3`, `Enter3`, and `Info3` states. We could specify this as a regular expression [Fri97] over the sequence of states that will be visited (here, * is a wildcard that means "any sequence of transitions"):

```
*,Star3,*,Enter3,*,Info3,*
```

The shortest test case that this would generate is

```
dial/WELCOME, wait/WELCOME, wait/WELCOME, */ENTER,
wait/ENTER, wait/ENTER, num2/NAME,
wait/INFO, wait/INFO, wait/BYE.
```

Or we might decide that we want to test the emergency numbers more thoroughly than other numbers, so we ask for tests that go through the `Emerg` state many times and exercise all the transitions in and out of that state. For example, we could specify this with the regular expression (over the states visited)

```
*,Enter1,Emerg,*,Enter2,Emerg,*,Enter3,Emerg,*
```

The shortest test that satisfies this explicit test case specification is

```
dial/WELCOME, */ENTER, 18/FIRE, */ENTER,
wait/ENTER, 18/FIRE, */ENTER,
wait/ENTER, wait/ENTER, 18/FIRE, wait/BYE.
```

The same specification would give a larger, more varied test suite if we added the other emergency numbers to our model because it would explore the alternative paths between the $\texttt{Enter}_i$ and `Emerg` states. As models become larger, explicit test case specifications become more useful for focusing attention on particular parts of the model. Regular expressions are not a very expressive notation for explicit test case specifications, so it is common to use more sophisticated notations to give more control over the generated test sets.

This illustrates that model-based testing does not have to be fully automatic. It is sometimes useful for the test engineer to direct the tools explicitly to generate a certain style of test, to follow a specific path through the model, or to test one part of the model more intensively than others. Model-based testing is not intended to be a tool that replaces human skill and ingenuity; rather it should be a tool that amplifies human skills.

Key Point Explicit test case specifications give an engineer precise, low-level control over which tests are generated from a model.

5.1.4 Complete Testing Methods

In this section we look at more powerful FSM test generation techniques that can guarantee to find all SUT errors.

Researchers have been working on FSM test generation algorithms for 50 years, since Moore started the field with his *Gedanken experiments on sequential machines* [Moo56]. During the 1960s through the 1980s, several *complete* test generation methods for FSMs were invented, such as the D-method [Hen64], W-method [Vas73, Cho78], the Wp-method [FvBK+91], the U-method [SD88, YU90], as well as the transition-tour or T-method [NT81], which we used earlier (which can be a complete method for some SUTs, as we shall see).

These algorithms are quite impressive in that they generate a set of tests that *guarantees* that the SUT implements the identical FSM as the model. However, to make this possible they must make some strong assumptions: they assume that the specification FSM model is *deterministic, minimal, complete, strongly connected* (or initially connected with a *reset* operation), and *has the same complexity as the SUT*. This last assumption means that if we view the actual behavior of the SUT as also being a finite state machine, then that finite state machine must have the same number of states as the FSM model. This is a very strong assumption, which is usually impossible to verify in real-world situations, especially when the SUT is a black box. However, in the Further Reading section of this chapter we discuss how some of these assumptions can be weakened.

The basic idea behind these methods is that, for every transition $s \xrightarrow{i/o} s'$ in the FSM model, we want to:

1. force the SUT into state s (by resetting and starting from the initial state, if necessary),
2. send the input i to the SUT and check that it produces the expected output o, and
3. check that the SUT is now in the expected state s'.

Step 3 is the difficult one—how can we check what state the SUT is in if the SUT is a black box?

The transition-tour method takes a simplistic approach to this: it assumes that the SUT provides a reliable[5] `status()` operation that tells us its

[5] If the `status()` operation is not known to be reliable, then it may be useful to call it *twice* after each transition to check that the `status()` operation itself does not change the state of the SUT.

current state. This means that it is sufficient to test all the transitions, calling the `status()` operation after each transition. If you are lucky enough to be testing an SUT that does have a trustworthy `status()` operation, then a transition tour of your complete FSM model is guaranteed to find all SUT errors (though we still must assume that the SUT and our model FSM have the same number of states). Of course, many SUTs do *not* provide a `status()` operation, like our Qui-Donc example. In this case, the transition-tour method is not guaranteed to find all *transfer* faults (where the SUT moves into the wrong state), but if there are no transfer faults, then it will find all *output* faults in the SUT.

If we do not have a `status()` operation, could we perhaps use one of the other methods to test the SUT, such as the D-/W-/Wp-/U-methods? They check that the destination state is correct by applying various cleverly chosen sequences of transitions and observing the outputs of the SUT.

Yes, in principle we could use one of those algorithms, but in practice they are not always useful because they produce very long test cases that contain up to $O(pn^3)$ transitions in the worst case, where n is the number of states and p is the size of the input alphabet. Our small Qui-Donc example has 11 states and 10 input symbols, so these methods could generate test suites containing up to $10 \times 11^3 = 13{,}310$ transitions, which is not really practical. Assuming that each transition takes an average of 15 seconds, a test suite of this magnitude would take more than 55 hours of nonstop testing, or about 1.5 person-weeks. This would be an excessive test length for such a simple system. In practice, most FSMs are better than the worst case and the D-method and U-method can generate a reasonable size test suite for many FSM models (usually the size of the test suites is T-method < U-method < D-method < Wp-method < W-method [Yu90]). The D-method and U-method test suites are usually much smaller than the Wp-method and W-method test suites.

Unfortunately, the D-method is not applicable to our Qui-Donc FSM because it does not have a *distinguishing sequence* (a sequence of inputs that gives different output behavior when applied to each state of the FSM). The U-method is also not applicable because the `Info3` state does not have a UIO sequence (a sequence of inputs that gives different results when we are in the `Info3` state than when we are in any other state). If it did have a UIO sequence, then the U-method would generate a test suite of about 110 tests (one for each transition in the complete FSM), with an average length of around 6 transitions.[6] So this test suite would contain 600 to 700

[6]This estimate is based on an average of 3 transitions to get to a given state, followed by a test of one transition and then applying the UIO sequence (average length 2) for the resulting state.

transitions, which is five times larger than our transition tour but still small enough to be practical. Optimized versions of the U-method could reduce the test suite size further by overlapping test sequences [ADLU91].

A comparative study by Sidhu [SkL89] shows that when the SUT does not provide a reliable `status()` method, the transition-tour method (over the complete FSM) has weaker fault-detection power than the U-method, D-method, and W/Wp-methods, which all have similar fault-finding power. The transition-tour method over just the explicit FSM is weaker still and may fail to detect sneak paths (extra transitions) in the SUT. However, transition tours are still very useful, because they exercise every state and every transition of the FSM model and produce a test suite whose total length is proportional to the number of transitions.

In the second case study of this chapter, we will explore further test generation techniques and metrics (e.g., random walks, greedy random walks, and transition-pair coverage) that can produce practical test suites that are stronger than just transition coverage.

5.2 EFSMS AND THE MODELJUNIT LIBRARY

In this section, we introduce *extended finite state machines* (EFSMs), which make it possible to model more complex SUTs than are possible with FSMs. Then, we describe the ModelJUnit library, which can be used to write EFSMs in Java and for simple kinds of test generation.

5.2.1 Extended Finite State Machines

An EFSM looks similar to an FSM (states and transitions), but it is more expressive because it has internal variables that can store more detailed state information. For example, in the Qui-Donc model, rather than have three separate $\texttt{Enter}_i$ states as in Figure 5.1, an EFSM model might have just one `Enter` state, plus a `timeouts` variable that counts how many times a timeout has occurred.

So an EFSM can appear to have a small number of visible states, while it actually has a much larger number of internal states. Mapping the large set of internal states down into the smaller set of visible states is a kind of *abstraction*. Deciding how to do this abstraction (i.e., how to partition the internal state space) is an important design decision because it strongly influences the kinds of test that can be generated from the EFSM.

Figure 5.2 illustrates this abstraction process. Imagine that we have an SUT with an infinite state space (lots of integer variables), and we have decided to model it by an EFSM whose internal state space contains just two

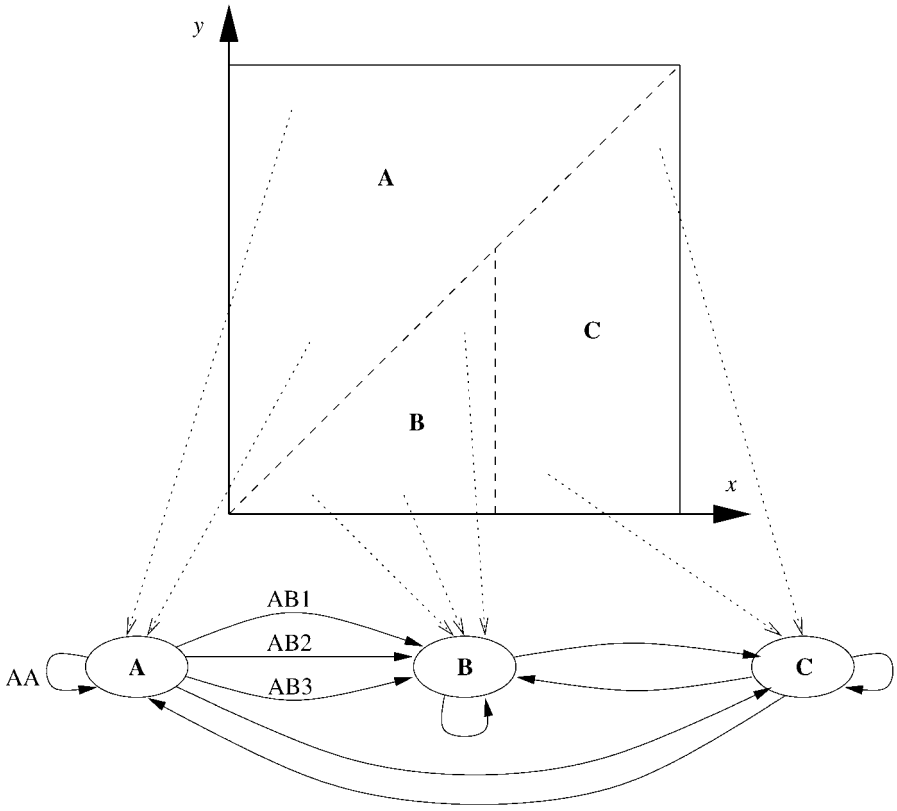

FIGURE 5.2 Abstraction of the large internal state space of an EFSM into three visible states.

integer variables, $x, y \in 0..9$. This means that the EFSM has $10 \times 10 = 100$ internal states, which we decide is still more than we need for testing the SUT. Based on our test objectives and on what we know about the SUT behavior, we decide to partition this state space into three areas: area **A** is all the states where $y \geq x$, area **B** is all the states where $y < x$ and $x < 5$, and area **C** is all the states where $y < x$ and $x \geq 5$. The three visible states of the EFSM represent these three areas, respectively.

The transitions of an EFSM sometimes need to update the EFSM state variables (e.g., x and y in Figure 5.2, or the `timeouts` variable in a Qui-Donc EFSM), so we often attach some code to each transition to perform these state updates. For example, the

$$\texttt{Enter1} \xrightarrow{\texttt{wait/ENTER}} \texttt{Enter2}$$

and

$$\texttt{Enter2} \xrightarrow{\texttt{wait/ENTER}} \texttt{Enter3}$$

transitions in Figure 5.1 would both be modeled by a single EFSM transition, from state `Enter` to itself, that contained code to increment the `timeouts` variable.

A transition in an EFSM can also have a *guard*, which is a boolean expression that can enable or disable the transition. It must evaluate to true for the transition to be taken. For example, the transition just discussed that increments `timeouts` would also need the guard `[timeouts<2]` to ensure that the `timeouts` variable is not incremented too many times. A separate transition, with the guard `[timeouts==2]` would handle the `wait/BYE` transition from state `Enter` to state `Start`.

Returning to the example in Figure 5.2, the transition labeled AA models an operation that, when called in a state within region A (where $y \geq x$), will stay in region A. For example, it might be an operation that increments y. The transitions labeled AB1, AB2, and AB3 model three operations that change the (x, y) variables from region A to region B. For example, they might be defined as follows:

AB1: $x, y := 1, 0$ (with no guard)

AB2: $y := 0$ with the guard $[x < 5]$

AB3: $y := y - 1$ with the guard $[x = y$ and $0 < x < 5]$

The advantage of using an EFSM model rather than a simple FSM model is that it allows us to take a complex SUT (with billions of states or infinite states) and build a more detailed model of it (perhaps with hundreds or thousands of states) than would be possible with an FSM. Then by defining the visible states of the EFSM carefully (typically just a few dozen of them), we can reduce the test generation task to something practical and focus on the interesting transitions between different areas of the model. The two levels of abstraction give us better control than one level would because they are used for different purposes.

- The medium-size state space of the EFSM and the code in its transitions are used to model the SUT behavior more accurately than is possible with just an FSM, and thus generate more accurate inputs and oracles for the SUT.
- The smaller number of *visible* states of the EFSM defines an FSM that can be used to drive the test generation. For example, we can use graph

algorithms such as transition tours to generate tests. These would not be practical on a larger EFSM state space that has thousands of states and more transitions.

Key Point An EFSM can model an SUT more accurately than an FSM, and its visible states define a second layer of abstraction (an FSM) that drives the test generation.

5.2.2 The ModelJUnit Library

The ModelJUnit library is a set of Java classes that is designed to be used as an extension of JUnit for model-based unit testing of Java classes. JUnit[7] is a widely used Java library for writing unit tests for Java classes. ModelJUnit was developed by one of the authors (Utting) as a *simple*, *open-source* framework for exploring FSM-based testing. It allows the FSM models to be written in Java, which is familiar to programmers, and because it is an extension of JUnit, the tests are run in the same way as other JUnit tests.

Most commercial and research model-based testing tools use sophisticated notations and theories that take some time to learn, whereas with the ModelJUnit library, one can start with an extremely simple FSM model and begin testing immediately, and then progress to slightly more sophisticated EFSM models as desired. The ModelJUnit library is based on ideas from Harry Robinson[8] and from Spec Explorer [VCST05], but it has only a small subset of the functionality of Spec Explorer. ModelJUnit is available from the website associated with this book.

The basic philosophy of ModelJUnit is to take advantage of the expressive power of Java (procedures, parameters, inheritance, annotations, etc.) to make it easier to write EFSM models, and then provide a collection of common traversal algorithms for generating tests from those models. It is typically used for *online* testing, which means that the tests are executed while they are being generated. The EFSM usually plays a dual role: it defines the possible states and transitions that can be tested, and it acts as the *adaptor* that connects the model to the SUT (which is usually another Java class).

Each EFSM model is written as a Java class, which must have at least the following methods:

[7] See *http://www.junit.org*.

[8] The C# login model from his STAR East 2005 presentation [Rob05] shows one way to express an FSM as a C# class. See *http://www.geocities.com/harry_robinson_testing/stareast2005.htm*.

`Object getState()`: This method returns the current visible state of the EFSM. So this method defines an *abstraction function* that maps the internal state of the EFSM to the visible states of the EFSM graph. Typically, the result is a string, but it is possible to return any type of object.[9]

`void reset(boolean)`: This method resets the EFSM to its initial state. When online testing is being used, it should also reset the SUT or create a new instance of the SUT class. The boolean parameter can be ignored for most unit testing applications.[10]

`@Action void` *`name`*$_i$`()`: The EFSM must define several of these *action* methods, each marked with an `@Action` annotation. These action methods define the transitions of the EFSM. They can change the current state of the EFSM, and when online testing is being used, they also send test inputs to the SUT and check the correctness of its responses.

`boolean` *`name`*$_i$`Guard()`: Each action method can optionally have a *guard*, which is a boolean method with the same name as the action method but with "`Guard`" added to the end of the name. When the guard returns true, then the action is enabled (so may be called), and when the guard returns false, the action is disabled (so will not be called). Any action method that does not have a corresponding guard method is considered to have an implicit guard that is always true.

Each action method typically defines a short, straight-line sequence of JUnit code that tests one aspect of the SUT by calling one or more SUT methods and checking the correctness of their results. The effect of applying model-based testing to the EFSM is to make a traversal through the EFSM graph, and this weaves those short sequences of test code into longer sequences of more sophisticated tests that dynamically explore many aspects of the SUT.

Using Java as the notation for writing EFSMs has benefits and limitations. The benefits include the familiarity of Java, having the expressiveness of a full programming language available, and the ability to quickly change the structure of the EFSM graph simply by redefining the `getState()` abstraction function or by modifying the guards and actions.

Some of the limitations are that the guards and transitions are defined as executable methods rather than as symbolic formulae. So graph exploration

[9] The objects that are returned must correctly implement the Java `equals` and `hashCode` methods, since these are used to compare states.

[10] The boolean parameter can be set to false in order to explore the FSM model without testing the underlying SUT. This can be useful if the SUT operations are very slow and we want an algorithm to quickly explore the FSM before testing starts.

and test generation algorithms can execute guards and transitions and inspect their results (true/false from a guard or a new EFSM state after a transition), but they cannot inspect the internal structure of the guards or transitions. To create the EFSM graph, ModelJUnit is limited to exploring it dynamically by executing enabled transitions. This means that it can be difficult to obtain the whole graph if some guards are rarely true. On the other hand, even if the EFSM graph is too large to explore completely, some forms of test generation are still possible, so the EFSM approach is still useful.

Another limitation is that the SUT interactions are handled internally within each transition, so the SUT input and output values are not explicitly represented in the EFSM graph as they are in a Mealy machine FSM model.[11] This places some small limitations on the test generation algorithms and coverage metrics that we can use in ModelJUnit. For example, we can measure action coverage and state coverage but not input coverage or output coverage. We can use transition-tour test generation algorithms but not some other test generation methods, such as the W-method, that analyze the output part of transitions. However, in practice this limitation is outweighed by the benefit of being able to generate rich SUT inputs dynamically and perform more sophisticated checking of the SUT outputs than the simple equality check of a Mealy machine FSM.

5.2.3 An EFSM Model of Qui-Donc

As an example of writing an EFSM in the ModelJUnit style, let's convert our Qui-Donc model into an EFSM. Since we cannot automate the execution of the Qui-Donc tests (JUnit is not yet smart enough to understand the spoken responses from Qui-Donc), the tests that we generate with ModelJUnit will simply print out test sequences (input-output pairs) that a human can execute later. That is, here we are using ModelJUnit to generate offline tests, whereas in the next section we use it for online testing.

Listing 5.3 shows the state-related parts of the model and Listing 5.4 shows the actions. We use the same set of input symbols as in Figure 5.1, but we have put the full output messages into our model rather than just their names, to show how we can use Java string concatenation to reduce the duplication of strings. The "`wait/...`" output message means that the previous message is repeated.

The interesting thing about this EFSM model is that it has only 5 states (`Start`, `Star`, `Enter`, `Emerg`, `Info`) rather than the 11 states of our

[11] Technically, this means that ModelJUnit models are *labeled transition systems* (LTS) rather than traditional FSMs.

Listing 5.3 Qui-Donc EFSM model in Java: the internal state variables and the `getState()` and `reset()` methods.

```
/** A simple EFSM model of the Qui-Donc service.
 *  Qui-Donc is a service of France Telecom that allows you
 *  to ring up, enter a telephone number, and find who
 *  owns that telephone number and what is their address.
 */
public class QuiDonc implements FsmModel
{
  public enum State
    { Start, // not yet connected to the Qui-Donc service.
      Star,  // waiting for the initial '*'.
      Enter, // waiting for a telephone number to be entered.
      Emerg, // after explaining the emergency number, waiting for '*'.
      Info   // ready to give information about the subscriber.
    };
  private State currState; // the current state of the system.
  private int timeouts;    // on the third timeout, we hang up.

  public String WELCOME  = "Welcome to Qui-Donc.  Please ...";
  public String NOTALLOW = "Your telephone does not allow...";
  public String ENTER    = "Please enter the 10-digit tel...";
  public String ERROR    = "Invalid number entered.  Plea...";
  public String FIRE     = "18 is the emergency number fo...";
  public String SORRY    = "Sorry.The number 03 81 12 3..."+ENTER;
  public String INFO     = "Press 1 to spell the name, pr...";
  public String NAME     = "The number 03 81 12 34 56 cor..."+INFO;
  public String SPELL    = "Renard is spelled R, E, N, A, R..."+INFO;
  public String ADDR     = "The address of Renard, K. J. ..."+INFO;
  public String BYE      = "Thank you for using the Qui-D....";

  public QuiDonc()
  {
    timeouts = 0;
    currState = State.Start;
  }

  public String getState()
  {
    return currState.toString(); // + (timeouts+1);
  }

  public void reset(boolean testing)
  {
    timeouts = 0;
    currState = State.Start;
  }
```

LISTING 5.4 Qui-Donc EFSM model: actions and guards.

```
public boolean dialGuard()
{return currState==State.Start;}
public @Action void dial()
{
  out.println("dial/"+WELCOME);
  currState = State.Star;
  timeouts = 0;
}

// No guard -- always enabled.
//  We call this wait_, to avoid
//  a clash with Object.wait().
public @Action void wait_()
{
  timeouts++;
  if (timeouts >= 3
      || currState==State.Emerg
      || currState==State.Start)
  {
    if (currState==State.Star)
     out.println("wait/"+NOTALLOW);
    else
     out.println("wait/"+BYE);
    currState = State.Start;
    timeouts = 0;
  }
  else
    out.println("wait/...");
}

public boolean starGuard()
{return currState==State.Star
      || currState==State.Emerg
      || currState==State.Info;
}
public @Action void star()
{
  out.println("*/"+ENTER);
  currState = State.Enter;
  timeouts = 0;
}

public boolean badGuard()
{return currState==State.Enter;}
public @Action void bad()
{
  out.println("bad/"+ERROR);
  // state is unchanged.
  timeouts = 0;
}

public boolean num18Guard()
{return currState==State.Enter;}
public @Action void num18()
{
  out.println("18/"+FIRE);
  currState = State.Emerg;
  timeouts = 0;
}


public boolean num1Guard()
{return currState==State.Enter;}
public @Action void num1()
{
  out.println("num1/"+SORRY);
  // state is unchanged.
  timeouts = 0;
}


public boolean num2Guard()
{return currState==State.Enter;}
public @Action void num2()
{
  out.println("num2/"+NAME);
  currState = State.Info;
  timeouts = 0;
}


public boolean key1Guard()
{return currState==State.Info;}
public @Action void key1()
{
  out.println("1/"+SPELL);
  // state is unchanged.
  timeouts = 0;
}


public boolean key2Guard()
{return currState==State.Info;}
public @Action void key2()
{
  out.println("2/"+ADDR);
  // state is unchanged.
  timeouts = 0;
}
```

original Qui-Donc model because our `getState()` method returns only the `currState` variable and ignores the `timeouts` counter. For example, this collapses the original three states (`Star1`, `Star2`, and `Star3`) into a single state, `Star`. This smaller set of states means that some of the transitions in the original model (e.g., the three `*/ENTER` transitions from the $\texttt{Star}_i$ states to the `Enter1` state) also collapse into a single transition. So our EFSM model is significantly smaller, even though it models exactly the same functionality. Figure 5.3 shows the EFSM graph that is generated by ModelJUnit with the `getState()` method shown in Listing 5.3.

If we redefine the `getState()` method so that it also puts the value of `timeouts` into the state string (as shown in the comment), then we get a larger EFSM graph that is almost identical to the original Qui-Donc graph in Figure 5.1. The only significant difference is that we have an extra `wait_` loop on the `Start` state because our `wait_` action has no guard, so it is enabled in *every* state. This extra transition could easily be removed by adding a guard

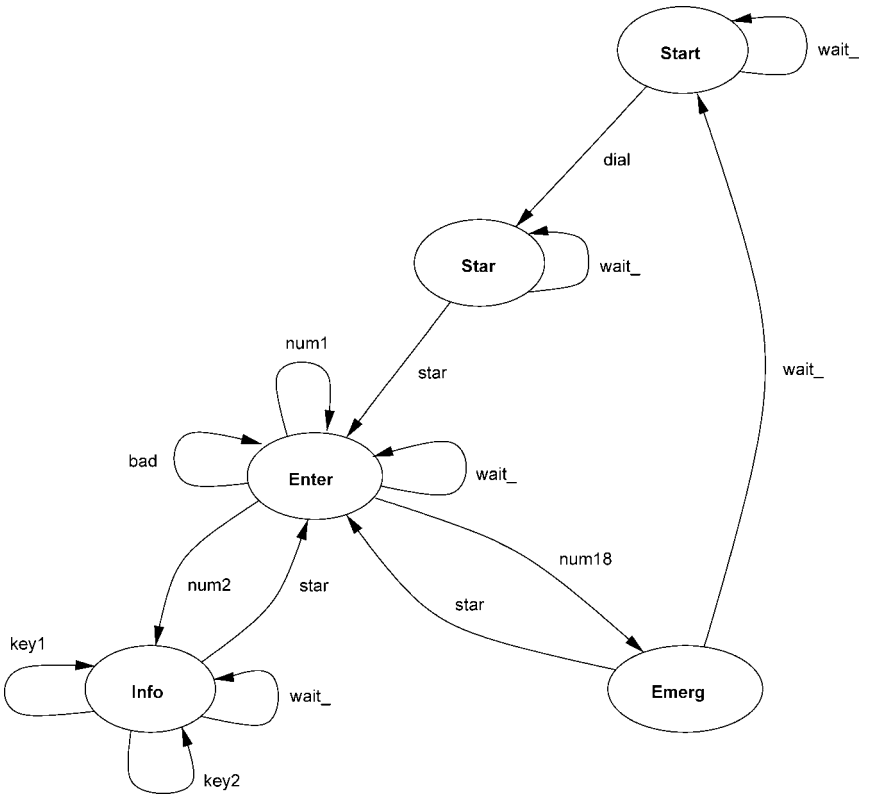

FIGURE 5.3 Qui-Donc EFSM, generated from the Qui-Donc Java model.

(`currState!=State.Start`) to the `wait_` action, but we wanted to leave it without a guard to illustrate that a missing guard is interpreted as always true.

An interesting aspect of the FSM in Figure 5.3 is that it does not include all the `wait` transitions back to the `Start` node. This is because the `wait` transitions are actually nondeterministic and the random graph exploration did not follow the wait loops enough times to find the alternative path. In this case, the missing transitions could be discovered automatically if we traversed the EFSM graph using an *all-loops* heuristic, going around each loop at least three times. But in general, the EFSM notation is expressive enough to define large or infinite graphs, so when we try to take a finite projection of such a graph (to create an FSM) by traversing it randomly or using heuristics, it is quite common that we miss some transitions. This can slightly weaken the test suites generated from that FSM. It illustrates the fact that testing is often incomplete.

One last point about this EFSM model is that although we wrote only 9 action methods, these generate the 15 transitions in Figure 5.3, or the 36 transitions in the larger EFSM of Figure 5.1 if we use the alternative `getState()` method. This illustrates the power of using Java and the EFSM approach—a single EFSM method can define many transitions, and the full power of Java coding can be used within that action to update the state appropriately. This can dramatically reduce the time required to write a complex model. But it also means that writing the model becomes a programming task, which requires programming skills, whereas the simpler FSM models can sometimes be designed by nonprogrammers using a graphical drawing tool or by writing a transition table in a spreadsheet.

VALIDATING THE MODEL

To validate this model, we wrote a simple `main` method that allowed methods to be called interactively; then we did a manual traversal through the model, using the transition tour from Figure 5.1 as a guide. This exposed three errors in the model: one typo (the `dial` action ended in the `Start` state rather than the `Star` state, and two cut-and-paste errors (the `key1` and `key2` guards were true in the `Enter` state rather than in the `Info` state). Using the transition tour was a bit excessive for the validation of this simple model; a shorter informal tour that just covered all 9 actions would have exposed the same errors (which is just as well, because we don't usually have an existing transition tour before the model is built, and it would not be useful to use a tour generated from the model to validate the same model!). In fact, it would not have been a disaster if we had done *no* validation of the model—the errors might have remained in the model and we would have generated some

incorrect tests, but those errors would almost certainly be exposed when the tests were executed; the chance of having an identical error in the model and in the SUT is quite low.

GENERATING TESTS FROM THE MODEL

If we take this Qui-Donc model and generate a *random walk*, which randomly calls any enabled action method, we get a test sequence that starts like this:

```
dial/Welcome to Qui-Donc.  Please press the star key.
wait/...
*/Please enter the 10-digit telephone number of the
  subscriber you are looking for, followed by the hash key.
wait/...
bad/Invalid number entered.  Please enter another number,
  followed by the hash key.
18/18 is the emergency number for the Fire Brigade.
  If you want to do another search, press star.
wait/Thank you for using the Qui-Donc service.
etc.
```

This output could be used as a manual test script. We could generate a random walk test sequence of a fixed length to match our testing budget (say 240 transitions, which would be about 1 hour of manual test execution), and then measure the model coverage to see what percentage of states, actions, or transitions we have tested. Alternatively, we might choose to use a more sophisticated generation algorithm, such as a transition tour, that achieves a certain kind of coverage (e.g., transition coverage) with a minimal test suite.

5.3 UNIT TESTING ZLIVE WITH EFSMS

In this section, we go through another case study using a parameterized EFSM model to do model-based unit testing of a hierarchy of Java classes. The concepts that we illustrate include:

- EFSM models in Java, with state variables, and guards and state updates within the transitions.
- *online* testing, where the SUT is tested while the tests are being generated, rather than later.
- a simple kind of *adaptor*, which links the model to the SUT.
- several structural model coverage criteria for FSMs/EFSMs.

5.3.1 The System under Test: ZLive `FlatPred`

ZLive is an animator for the Z specification language, developed by one of the authors (Utting). It is part of the open-source Community Z Tools project,[12] which has developed support tools for ISO Standard Z [ISO02] and several extensions of Z. ZLive can evaluate simple Z expressions and predicates and can execute some Z specifications that have small state spaces (in general, the Z specification language is not executable [HJ89, Spi92]). It is typically used to test parts of specifications, to generate instances of schemas, or to search for counterexamples. The implementation of ZLive contains about 8000 lines of commented Java.

Inside ZLive, the Z specification is translated into a sequence of `FlatPred` objects, reordered for more efficient evaluation, then evaluated using backtracking search. There are over 20 kinds (subclasses) of `FlatPred` objects, and each `FlatPred` object represents a constraint among several variables. The `FlatPred` class hierarchy is the most significant part of ZLive and accounts for about two-thirds of its source code. Here are a few examples of `FlatPred` objects and the constraints that they represent:

`FlatMult(x,y,z)`	:	$x * y = z$
`FlatPlus(x,y,z)`	:	$x + y = z$
`FlatConst(x,k)`	:	$x = k$ (k is a constant)
`FlatMember(s,x)`	:	$x \in s$
`FlatRangeSet(a,b,s)`	:	$s = (a..b)$
`FlatSetComp(decls,pred,expr,s)`	:	$s = \{decls \mid pred @ expr\}$

For example, the following sequence of `FlatPred` objects will return all the factors of 24, one by one, in the variable `answer`.

```
FlatRangeSet(1,24,range),        // range = (1..24)
FlatMember(range, factor1),      // factor1 ∈ range
FlatMult(answer, factor1, 24)    // answer * factor1 = 24
```

How does it do this? The `FlatMember` constraint iterates through the set `range`, setting the `factor1` variable first to 1, then to 2, then to the successive integers up to 24. For each value of `factor1`, the `FlatMult` constraint tries to find a value of `answer` that satisfies `answer` $*$ `factor1` $= 24$; if it fails then we backtrack to the previous constraint and try the next value of `factor1`; but if it succeeds, then the correct value is assigned to the `answer` variable and a solution is returned to the caller.

[12] See *http://czt.sourceforge.net*.

This illustrates that each `FlatPred` object can be used in several different *modes*, with inputs or outputs. The example used `FlatMult` in the mode `OII`, meaning that its first parameter was an output, while the second and third parameters were inputs. But `FlatMult` can also be used in the mode `IIO` (which multiplies $x \times y$ and sets z to the result, so it never fails), in mode `IOI` (which sets y to a number that satisfies $x \times y = z$ or fails if there is no such number), or in mode `III` (which takes all three parameters as inputs and fails if their values do not satisfy $x \times y = z$).

The behavior of a typical `FlatPred` object is reasonably sophisticated, so it needs to be tested thoroughly. We decided to apply model-based unit testing to the `FlatPred` subclasses, to try to improve the quality of the unit tests and to see how convenient it was to use an FSM approach to test a hierarchy of Java classes.

ZLive has a system test suite of over 600 tests, which tests the `FlatPred` subclasses in many ways but does not generally exercise all modes. In addition, 17 out of 23 of the `FlatPred` subclasses already had a reasonable set of manually designed JUnit tests (2500 lines of commented Java), which tested every mode of those subclasses. The remaining six `FlatPred` implementations had no unit tests (prior to adopting model-based testing), mostly because they implemented only one mode (`III`), so they were tested reasonably well by the system tests.

5.3.2 A Family of Models

In this section, we gradually develop a parameterized EFSM model for `FlatPred`, using `FlatMult` as an example but taking care to keep the model general enough that we can use it for all the subclasses of `FlatPred`. This means that the overhead of creating the model can be amortized over the testing of more than 20 classes.

Before we can design a model, we must describe the `FlatPred` API a little more to see which methods change the state of the `FlatPred` object under test, when each method can be called, and so on. So here is a brief summary of the main `FlatPred` methods (we ignore the constructors because they are different for each subclass).

`Mode chooseMode(Envir)`: This is the key method that allows us to find out which evaluation modes a `FlatPred` object supports. For example, if we have a `FlatMult(x,y,z)` object and call its `chooseMode(env)` method with `env` containing just `x`, the result will be null because it is not possible to solve the $x \times y = z$ constraint knowing only x. But if `env` contains `x` and `y`, then a `Mode` object will be returned, indicating that mode `IIO`

is allowed. The returned `Mode` object contains an updated environment which includes the output variable `z`, plus some other statistical information about the expected number of solutions of this constraint.

`void setMode(Mode):` After we have called `chooseMode(..)` one or more times to find out what modes are possible, we can use this method to select one of those modes. This fixes it as the mode that will be used during evaluation.

`void startEvaluation():` After an evaluation mode has been set with `setMode`, this method can be called to start a new evaluation. This is similar to requesting an iterator from a collection object in order to iterate through all the solutions. Note that we typically perform a sequence of several evaluations using the same mode—each evaluation usually has different values for the input variables.

`boolean nextEvaluation():` This calculates the next solution and returns true if there is a valid solution, false otherwise. When it returns true, it updates the values of the output variables in the current environment so that following `FlatPred` constraints can access those new values. This method can be called only after `startEvaluation()`.

`boolean inferBounds(Bounds):` This is an optional static analysis method for inferring information about integer bounds and allowing more efficient evaluation. It is usually called before `chooseMode()`. The `Bounds` parameter is a repository for the currently known lower and upper bounds of all the integer variables in scope. This method returns true if it has tightened those bounds. Some of the bounds information may be saved within the `FlatPred object`, so calling this method may change the state of the `FlatPred` object, as well as change the `Bounds` repository.

So this is the typical life cycle of a `FlatPred` object:

1. [*optional*] Call `inferBounds` to do some static analysis.
2. Call `chooseMode` one or more times to see which modes are possible, and then call `setMode` to select one of those modes.
3. Call `startEvaluation`, followed by a series of calls to `nextEvaluation` to get each solution until it returns false.

The last step is usually repeated several times for each evaluation that we want to perform. This life cycle shows that the `FlatPred` object changes state quite a few times and that some methods can be called only in certain states.

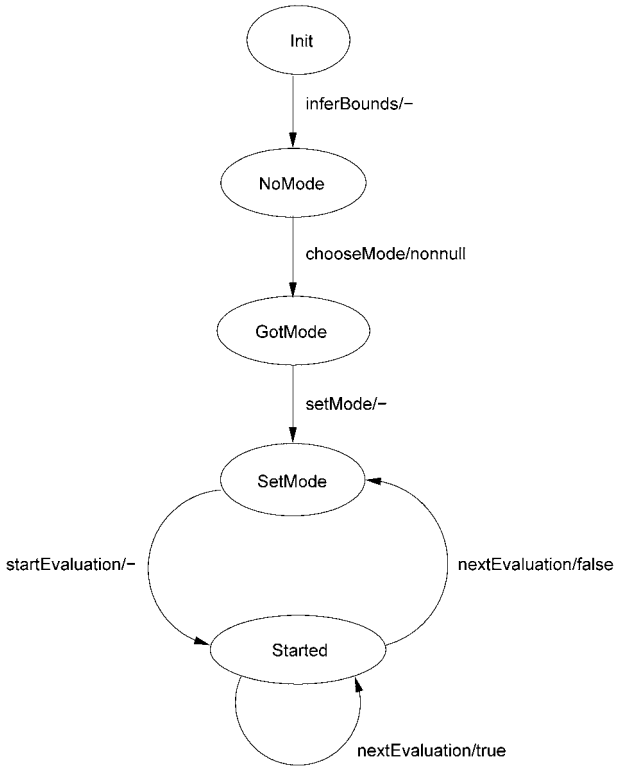

FIGURE 5.4 FSM model for `FlatMult`, version 1. The labels on the transitions are the names of the `FlatMethods` to call, with the expected result of the method shown after the slash (– means void or "don't care").

It is this state-based behavior of `FlatPred` that makes it suitable to model with an FSM/EFSM.

Key Point If you have a class that changes state and has different behavior in each state, then an FSM/EFSM model is a good choice for testing that class.

This typical life cycle suggests that we could use an FSM model of `FlatPred` like the one shown in Figure 5.4.

But in fact, this is not a good model, for the following reasons:

Not enough loops and choices: The model is too simplistic! The power of model-based testing comes from having lots of paths to explore. But this

model has only two loops and very few alternative paths. We need to add more paths, to explore all the different modes that are possible, to try restarting a new evaluation before the last one is complete, to try choosing a new mode after or during an evaluation, and so on. Without these alternatives, the generated tests will repeat the same sequences of methods over and over again, with little variation.

Not deterministic: Once we reach the `Started` state, there are two transitions that leave that state, but both of them are labeled with the same input value, `nextEval`. This means that both transitions involve calling the `nextEvaluation()` method, but we expect a response of true from the SUT in one case and a response of false in the other case. This makes it impossible for the FSM traversal algorithms to *control* which transition will be followed. In this chapter, we require our FSM models to be deterministic so that the test generation algorithms can control the SUT.

Not a strong enough oracle: This model describes the behavior of almost every `FlatPred` and does not have strong enough checks to ensure that each subclass has the correct behavior. For example, if a `FlatMult(x,y,z)` object returned *two* solutions for z (say, 12 and 13) when $x = 3$ and $y = 4$, the model would happily follow the `nextEval/true` transition two times, then the `nextEval/true` transition once, and would agree that this is the correct behavior! We need some way of having stronger checking on the behavior of each `FlatPred` object. Since the behavior varies between `FlatPred` subclasses, and even between different input values, this means we need to *parameterize* our generic `FlatPred` model to tell it the specific behavior to expect for each particular SUT.

Key Point Whenever possible, make your FSM model deterministic. This simplifies test generation and means that the generated tests can control the SUT.

To fix the first of these problems, we add more transitions to our model. For example, we add some separate transitions to try each mode, plus some `newMode` transitions that go back from the evaluation states and start using a new mode. This allows us to test the effect of jumping out of an evaluation and switching to a different mode. Figure 5.5 shows some of these new transitions (it shows only the `OOO` and `III` modes).

To fix the second and third problems, we switch from a simple FSM model to a more sophisticated extended finite state machine (EFSM) model. We write our EFSM as a Java class, following the `ModelJUnit` style. So the

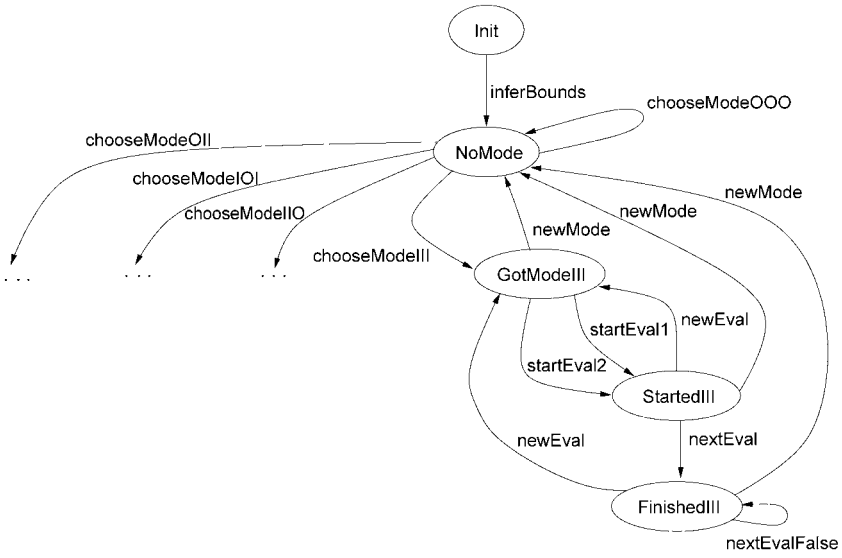

FIGURE 5.5 EFSM model for `FlatMult`, showing only the `OOO` and `III` modes. The labels on the transitions are the names of the `@Action` methods in the `ModelJUnit` class.

names on the transitions of Figure 5.5 are actually the names of the Java `@Action` methods in our `ModelJUnit` class, which we develop over the next few pages. Each of these methods is responsible for updating the current state of the EFSM as well as interacting with the SUT.

We will use the internal variables of the EFSM to record more precise information about the expected behavior of the current `FlatPred` that is being tested, such as the mode that we are testing and the values of each input variable. Also, to permit more accurate testing of each subtype of `FlatPred`, we will parameterize our general `FlatPred` model with some values that are specific to the particular `FlatPred` object under test so that our EFSM can use those parameters to check the correctness of the SUT behavior. For example, here is the Java code that constructs an EFSM model for `FlatMult(x,y,z)` (the full graph of this model is shown in Figure 5.6):

```
FlatPredModel model = new FlatPredModel(sut,
    new ZRefName[] {x,y,z},
    "OII,IOI,IIO,III",
    new Eval(1, "???", i3, i4, i12),
    new Eval(0, "I?I", i2, i5, i11) // 11 is prime
  );
```

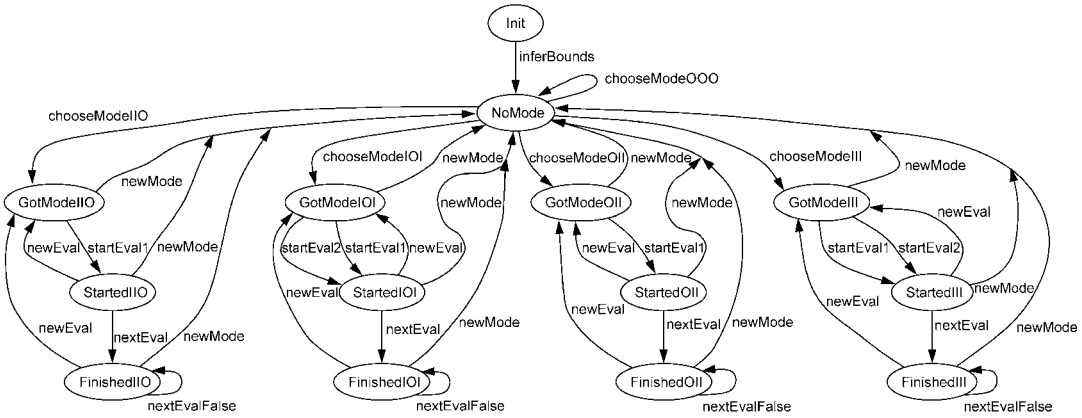

FIGURE 5.6 FSM model for `FlatMult`, final version.

Let us explain these parameters to give some examples of how it is possible to parameterize an EFSM. We will always create the same type of model, a `FlatPredModel`, but the parameters to its constructor can change the structure and the meaning of the EFSM model dramatically.

The first parameter, `sut`, will be discussed in Section 5.3.3—it has to do with test execution, not with the EFSM model. The second parameter informs the EFSM model about all the variables used by the SUT. In this case, our SUT implements the constraint $x \times y = z$, so the free variables are x, y, and z.

The third parameter lists all the modes that are supported by the SUT. Modes that are in this list are expected to give a non-null result when `chooseMode` is called, while any modes that are not in the list are expected to give a null result. The `FlatPredModel` class uses this parameter to determine the gross structure of the EFSM: each mode in the list generates a transition that leads to a separate subgraph of the EFSM that tests evaluations using that mode (e.g., the `chooseModeIII` transition); each mode not in the list generates a transition that loops back to the `NoMode` state (e.g., the `chooseModeOOO` transition).

The fourth and fifth parameters are two evaluation examples to use when testing the SUT. The first of these examples is used by the `startEval1` transition to start an evaluation, and the second example is used by the `startEval2` transition. We use two examples rather than one because we want to swap between *different* evaluations during the test runs to check that one evaluation does not corrupt the other. It would be possible to parameterize the model with more than two examples, but we usually do this by creating multiple models instead.

Each of these evaluation examples is encapsulated into an object that contains a value for each of the free variables (e.g., the first example sets $x = 3$, $y = 4$, and $z = 12$), plus the number of solutions expected. This enables us to strengthen the oracle checking within the `startEval1` and `startEval2` transitions so that they check that `nextEvaluation()` returns true the correct number of times, as well as telling us what values to use for the inputs and what values to expect in the outputs. It turns out that the number of solutions sometimes depends on the mode used, which is why each evaluation example has a string like `I?I` to say which parameters must be inputs (`I`), which must be outputs (`O`), and which can be either inputs or outputs (`?`). Since the second example restricts the evaluation modes to `I?I`, the `startEval2` transition is disabled in the `OII` and `IIO` modes, where x and z are not inputs, respectively. This is another example of how these parameters change the shape of the EFSM graph.

Having designed our EFSM model, the next section describes how this model is written as a Java class, and then we look at how we can use the ModelJUnit library to explore the model, generate tests from it, and measure the coverage of the generated tests.

5.3.3 Encoding `FlatPredModel` in Java

The ZLive `FlatPred` example is a more complex EFSM model than the Qui-Donc model. The `FlatPredModel` class contains around 350 lines of commented Java, which is 230 noncomment source lines of Java, or 120 executable statements. In this section we briefly make a few comments about some interesting features of this model before we show the results of using it for test generation.

DEFINING THE EFSM STATES

Like the Qui-Donc model, the `FlatPred` model defines an enumeration of its five main states.

```
/** The possible main states of the FlatPred. */
enum State {Init, NoMode, GotMode, Started, Finished};
private State state_;
```

However, we want to distinguish the states of the different modes (to encourage the test generation algorithms to explore all the modes), so we define our `getState()` method to add the mode string (`III`, `IIO`, etc.) to the end of the state, but only when we are in one of the `GotMode`, `Started`, or `Finished` states. Note that the following `getState()` method makes use of several other private variables of the EFSM: `names_` is the array of free variables of the SUT, and `env_` is the current evaluation environment, which

is null in the `Init` and `NoMode` states and contains the values of the input variables in the other states.

```
public String getState()
{
  StringBuffer result = new StringBuffer();
  result.append(state_.toString());
  if (env_ != null) {
    // add the mode to the end of the state.
    for (int i=0; i<names_.length; i++)
      if ( ! env_.isDefined(names_[i]))
        result.append('O');
      else
        result.append('I');  // an input
  }
  return result.toString();
}
```

Note that if we have an SUT with five free variables, we will have EFSM states like `GotModeIIIIO` and `GotModeIIIII`, whereas if we have an SUT with just one free variable, we will have EFSM states like `GotModeI` and `GotModeO`. The `getState()` method gives us tremendous power to control the set of states in our EFSM, and this in turn helps to determine how many states of the SUT will be tested and which states will be tested. Effectively, the `getState()` method is performing the crucial *abstraction* function, mapping the huge or infinite number of possible SUT states down into a small set of interesting EFSM states that will be tested.

Key Point The abstraction function (the `getState()` method in ModelJUnit) is the key to controlling the number of states and transitions in your EFSM.

PARAMETERIZED EFSM ACTIONS

Figure 5.6 shows that we have several transitions going out of the `NoMode` state, each trying a different mode. The Java code for these transitions is very similar, so we define a parameterized helper method that contains all the common code. Then each transition can be defined in just a few lines:

```
/** Tries chooseMode with all names except the last being inputs. */
public boolean chooseModeIIOGuard() {return state_==State.NoMode;}
@Action public void chooseModeIIO() {chooseMode("IIO");}
```

The helper method (`chooseMode(String)`) is written to handle SUTs with any number of free variables—its parameter is always a three-character string, but the first and last characters control the first and last inputs, respectively, while the middle character controls all the remaining inputs (when *freevars* ≥ 3). This is another example of abstraction or simplification—we decided that it would be sufficient to test all the middle inputs in the same way. These kinds of decision are easy to code up into the EFSM because the full power of Java (loops, conditionals, parameterized methods, etc.) is available for use in each action method.

DEFINING THE EFSM GRAPH STATICALLY AND DYNAMICALLY

The `chooseMode(String)` helper method also uses a `validModes_` list, which is the second parameter to the `FlatPredModel` constructor, to determine the destination state of each `chooseMode` transition. This is an example of how a construction-time parameter can change the static structure of the EFSM.

It is always safe to use parameters and conditionals to define the static structure of the EFSM, but one must be careful when using conditionals that change the structure of the EFSM dynamically (that is, on-the-fly, during a traversal of the EFSM). It is okay to use guards to disable or enable transitions (though this can make it more difficult to discover the shape of the EFSM graph). But it is undesirable to define a transition that sometimes leads from state A to state B and at other times leads from state A to state C. This is a *nondeterministic* transition, and it can make the model coverage metrics misleading and confuse some of the graph traversal algorithms.

A better approach is to split that nondeterministic transition into two or more deterministic transitions with different names. If necessary, this can be done by introducing a new state A2, then writing a transition from A to A2 that stores a flag in an internal EFSM variable, followed by two separate transitions, $A2 \xrightarrow{AB} B$ and $A2 \xrightarrow{AC} C$, that have guards to enable them when the flag is true or false, respectively. This is the technique that we used to handle the nondeterministic `nextEval` transition in Figure 5.4—it has been split into separate `nextEval` and `nextEvalFalse` transitions in Figure 5.5.

ONLINE TESTING WITH MODELJUNIT

We now briefly review the responsibilities of the various parts of the EFSM model when it is being used for online testing with ModelJUnit. Recall that with online testing, the EFSM model has two roles: it defines the model and it acts as the adaptor that connects the model to the SUT.

The EFSM model must maintain a pointer to the SUT object as part of its internal state so that it can communicate with the SUT as testing

progresses. In fact, the SUT is the first parameter to our `FlatPredModel` constructor, so the `sut` parameter shown on page 173 would actually be an instance of `FlatMult` class, such as:

```
new FlatMult(x,y,z)
```

The guards of the action methods are responsible for saying when each action method is enabled; this may require querying the SUT to determine whether it is ready to perform some operation.

Each action method is responsible for:

1. changing the state of the EFSM (or leaving it unchanged, in the case of a loop transition);
2. sending test inputs to the SUT;
3. checking any SUT outputs to make sure they are correct (this is typically done via JUnit assert methods, which check some given condition and generate a test failure if it is false); and
4. (if possible) checking that the SUT is now in the correct state that agrees with the new EFSM state.

Responsibility 1 defines the shape of the EFSM graph. Responsibilities 2 and 3 are the *adaptor* part of the EFSM, which links the model to the SUT and implements the oracle checking. Responsibility 4 gives a stronger form of oracle checking that is not always possible to achieve if the SUT is a black box with not enough query methods to determine its state. However, when the classes that are being tested are well designed, they will often provide query methods that allow the SUT state to be determined. In this case, note that the transition-tour method is as powerful as the D-/W-/Wp-/U-methods of test generation. That is, in theory a simple transition tour is sufficient to find all differences between the model and the SUT (assuming that the SUT does not have more states than the model).

EXAMPLES OF ACTION METHODS

To illustrate how action methods can fulfill these responsibilities, here are some simple action methods from the `FlatPredModel`. The `nextEvalFalse` action calls the SUT `nextEvaluation()` method and checks its result, but it does not need to change the state of the EFSM (it loops back to the same state). The `println` message is just for debugging the model—it is sometimes useful to see the internal state of the EFSM as it is traversed.

```
public boolean nextEvalFalseGuard()
{ return state_ == State.Finished; }
```

```
/** Checks that nextEvaluation() returns false. */
public @Action void nextEvalFalse()
{
  boolean result = pred_.nextEvaluation();
  System.out.println("nextEvalFalse gives "
      +result+" with env="+env_);
  Assert.assertFalse(result);
}
```

The `newMode` action is the opposite. It does no testing of the SUT; it just changes the state of the EFSM and resets some of the EFSM internal variables. Note that this action is enabled in all the `GotModeXXX`, `StartedXXX`, and `FinishedXXX` states, so this one action method generates the 12 `newMode` transitions in Figure 5.6.

```
public boolean newModeGuard()
{ return state_ == State.Started
  || state_ == State.Finished
  || state_ == State.GotMode;
}
/** Go back and try a new mode. */
@Action public void newMode()
{
  System.out.println("newMode with env="+env_);
  mode_ = null;
  env_  = null;
  data_ = null;
  state_ = State.NoMode;
}
```

Other actions in `FlatPredModel` are more complex in that they test the SUT as well as change the EFSM state. Some of them use conditionals and loops to perform more comprehensive testing of the SUT. Now let's use this model to do some testing.

5.3.4 Test Results

We will take the simplest possible approach to test generation—a random walk through the model. This will allow us to see how the various model coverage metrics change as we generate longer and longer tests. We have also recorded the SUT code coverage metrics so that we can see how they are correlated with the model coverage.

Since we are using online testing, each transition that we take in the model is immediately testing the SUT. Note that a random walk of the

model is more powerful than just generating random SUT inputs because each transition of the model not only generates SUT inputs but also runs the oracle code that checks the correctness of the outputs. So with random input generation, the only oracle you get is crash/no-crash, whereas with a random walk of the model, you get the powerful oracle checking that you have (we hope!) included in the action methods of your model adapter.

What happens when an error is detected by this oracle code? Usually, JUnit just gives you a one-line message that says how the expected and actual values differed and which assert statement detected the error. ModelJUnit gives that same information, but it can also display the current state of the EFSM model and the complete path through the model that led to the error, if desired.

One interesting possibility of online testing, which ModelJUnit does not yet support, is the BeeLine approach to minimizing the test sequence that leads to a given error.[13] Let's say that we find an error when we are deep into a random walk through the model. The BeeLine algorithm repeatedly cuts out random segments of that sequence (replacing them with shorter paths through the EFSM) until it finds the shortest sequence that leads to the error. This can make it easier to analyze the cause of the failure.

For our `FlatMult` application, we just set the random walk going and let it do a few thousand transitions. One of the nice features of random walks is that you can ask for any length test suite. We use a fixed seed for the random number generator so that the random walk takes the same path each time we run our unit tests and we get repeatable results from successive runs.

Figure 5.7 shows how several common model coverage metrics increase as we let the random testing continue for longer and longer. Notice that the simple state coverage metric increases very quickly and reaches 90 percent after just 30 transitions and 100 percent after 86 transitions. The action coverage metric is similar and reaches 100 percent after 104 transitions. Transition coverage takes a little longer—it reaches 90 percent after 121 transitions and 100 percent after 299 transitions. The transition-pair coverage rises even more slowly—it reaches 50 percent after 132 transitions, 90 percent after 628 transitions, and 100 percent after 1211 transitions.

Recall that before we did this model-based testing of the `FlatMult` implementation, we already had a set of manually written JUnit tests for `FlatMult`. These were a typical set of 8 JUnit tests that tested all the modes of `FlatMult` and contained 72 executable statements, of which 43 were JUnit assert statements. These JUnit tests had achieved 68 percent branch coverage and 89

[13] This is another idea taken from Harry Robinson's STAREast 2005 Model-Based Testing Tutorial.

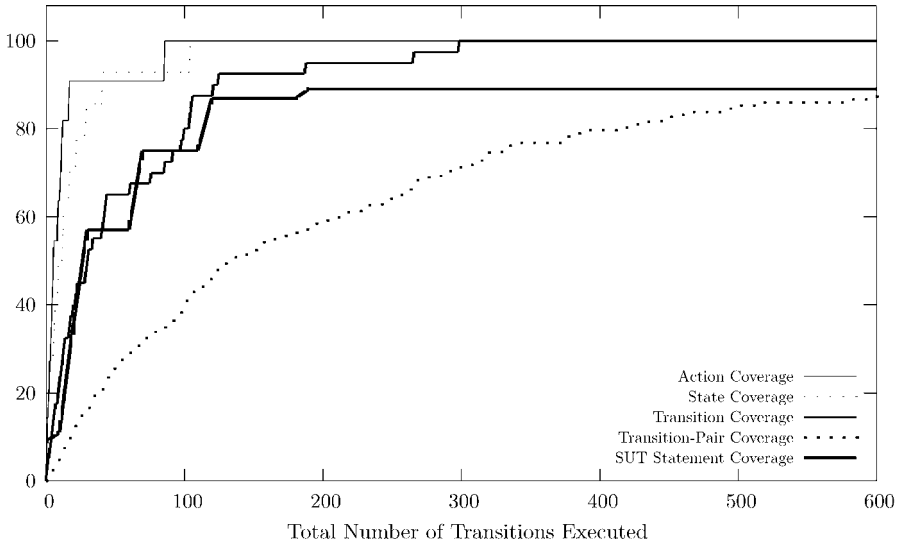

FIGURE 5.7 Coverage metrics for one random walk over the `FlatPredModel`, testing the `FlatMult` implementation.

percent statement coverage of the `FlatMult` code (which contained 62 executable statements). The random walk of the model-based testing improved the SUT branch coverage slightly. It reached the same level of coverage as the JUnit tests after 190 transitions and then increased the branch coverage to 73 percent after 370 transitions. Figure 5.7 shows that the statement coverage remained the same at 89 percent.

Figure 5.8 shows a logarithmic view of how these coverage metrics increase with the length of the test sequence. Rather than show percentages, this graph shows the raw numbers of items (States, Actions, Transitions, and Transition-Pairs) that have been covered as the length of the test increases.

Overall, the use of model-based testing on the `FlatPred` classes was very successful. It took about 8 hours to design (and redesign twice!) `FlatPredModel` and apply it to the `FlatMult` implementation. This is probably four to eight times longer than the time required to develop the original JUnit tests, but they were just for `FlatMult`, whereas the model was general enough to test most `FlatPred` subclasses. When we reused the model to test another `FlatPred` subclass, it took less than 15 minutes to parameterize the model and invoke a random walk, which is much less than the time required to write six to ten JUnit tests, so the overhead of the model development is cost-effective when spread across more than 20 subclasses.

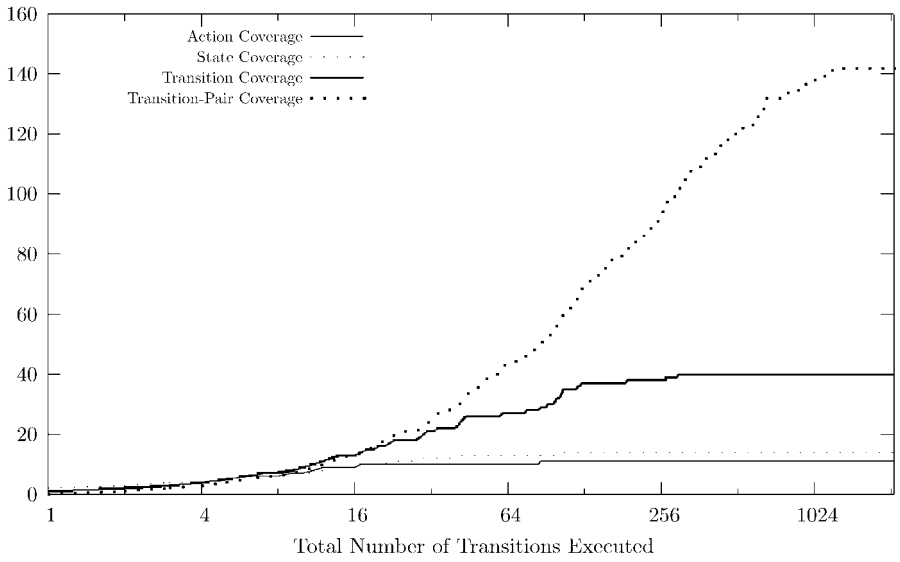

FIGURE 5.8 Raw coverage numbers for one random walk over the `FlatPredModel`, testing the `FlatMult` implementation (logarithmic view).

During the development of `FlatPredModel`, the random walk testing of `FlatMult` found 4 adaptor errors (errors in the parts of the `FlatPredModel` actions that communicate with the SUT), 1 model error (which was obvious when we looked at the graph of the EFSM), and 1 SUT error (to our surprise!). The error was in the `IOI` mode of `FlatMult(x,y,z)`, with the data $x = 2$ and $z = 11$. The implementation of that mode correctly did a division to calculate $y = z/x = 11/2 = 5$, but then it immediately assigned 5 to y and returned true rather than rechecking the multiplication first, and finding that $2 \times 5 \neq 11$, so returning false. This error was detected by the random walk test generation, but it had not been detected by the JUnit tests or by the 600 ZLive system tests, 36 of which used multiplication.

5.4 LABELED TRANSITION SYSTEMS MODELS

Finite state machines and extended finite state machines can be viewed as special cases of *labeled transition systems* (LTS). The main difference between LTS and FSM is that an LTS is allowed to have an infinite set of states and/or an infinite set of labels, whereas an FSM must have a finite set of states and a finite input alphabet.

LTSs were defined by Keller [Kel76] and have become widely used for modeling data-intensive systems, such as sequential and concurrent programs and hardware circuits [BJK+05, 113]. There are many kinds of LTS, including the Input/Output Automata of Lynch and Tuttle [LT87] and the Input/Output Transition Systems of Tretmans [Tre96].

Most of the research on test generation from LTS models has been theoretical, but some of the research has resulted in model-based testing tools being implemented. One of the most well-known theories is Tretman's *Input/Output Conformance* relation (ioco), which has been the basis for several model-based testing tools, such as TGV [JJ05] and TorX[14] [TB03].

LTS models are good at modeling nondeterministic *reactive systems*, where the generated tests must cope with events coming from the SUT spontaneously rather than as a direct response to the test inputs. We will see examples of this approach in the next two chapters (Sections 6.5 and 7.3).

5.5 SUMMARY

An FSM model has a finite set of *states*, plus a collection of *transitions* that lead from one state to another. Each transition is labeled with an input value i and an expected output value o and is sometimes written $s \xrightarrow{i/o} s'$, where the transition goes from state s to state s'.

An EFSM is similar to an FSM but is more expressive because it has internal variables that can store more detailed state information. Its actions can update those variables and can also have guards that enable or disable the action.

FSM and EFSM models can be used for system testing or for unit testing. The generated tests can be executed manually or automatically either online or offline.

If you have a system that changes state and has different behavior in each state, then a finite state model (FSM or EFSM) is a good choice for modeling that system.

The key step in designing a finite state model is abstraction—deciding which aspects of the SUT to ignore and which aspects to view as separate states or separate transitions. These design decisions determine the size and shape of the finite state model, which in turn determines the tests that are generated.

[14]The TorX tool is available for noncommercial use from the University of Twente. See *http://fmt.cs.utwente.nl/tools/torx* or *http://www.purl.org/net/torx*.

In the ModelJUnit library, the `getState()` method defines this abstraction function. It is the key to controlling the number of states and transitions in your EFSM. Small changes to this method can produce EFSMs of radically different shapes and sizes.

When generating tests from a finite state model, full transition coverage is a recommended minimum goal. The Chinese postman algorithm [Thi03] generates a minimum-length test sequence that satisfies 100 percent transition coverage. Transition-pair coverage is much more demanding, but the percentage of transition-pair coverage is a useful measure of how many interactions between adjacent transitions you have tested.

Random walks are the easiest way to generate test suites from an FSM model, and those test suites are surprisingly effective. Random walks have the nice properties that (1) you can easily generate a test suite of any desired length, and (2) the test suite gives more and more sophisticated coverage of the model as it grows longer and longer. With online testing, random walks can generate an infinite sequence of constantly changing tests that can be useful for overnight testing, perhaps even on multiple machines, each with a different seed for the random walk.

Explicit test specifications can be useful when you want to force a particular kind of test to be generated from a model.

5.6 FURTHER READING

Chapter 7 of Binder [Bin99] contains lots of practical advice on how to design various kinds of finite state systems. His Section 7.4.5 discusses how to choose among the test generation strategies for finite state systems. For each strategy he gives examples of the size of the generated test suite (his Table 7.12) and the fault-revealing power (his Figure 7.40) of the strategy.

Harry Robinson's entertaining introduction to the use of graph theory in model-based testing [Rob99] gives an overview of Euler tours, the Chinese postman problem, the New York street sweeper problem, the de Bruijn sequence for combinations of length 2 (also called safecracker sequences), the parallel street sweeper problem, random walks, and Markov chains.

The first four chapters of [BJK+05] give a more thorough and theoretical coverage of test generation from FSMs and describe a variety of test generation algorithms and their complexity. Chapters 5 through 9 of the same book give a comprehensive survey of model-based testing techniques for various kinds of labeled transition system. Also, see [Pet01, BT01] for brief annotated bibliographies of some of the main research papers and books in the LTS field.

Bochmann and Petrenko [BP94] and Lee and Yannakakis [LY96] give surveys of FSM and EFSM testing methods. The 1989 paper by Sidhu and Leung [SL89] describes four test generation algorithms for FSMs and gives some experimental results on their fault-detection capability and the size of the test suites that they generate. The 1991 paper by Fujiwara and colleagues [FvBK+91] adds some more algorithms and discusses assumptions and limitations of the various methods.

In Section 5.1.4 we note that many of the complete FSM test generation methods such as the D-method [Hen64], the W-method [Vas73, Cho78], the Wp-method [FvBK+91], and the U-method [SD88] place strong requirements on the FSM (it must be deterministic, minimal, complete, strongly connected, and have the same number of states as the SUT). More recent research allows many of these requirements to be relaxed. The minimal, complete, and strongly connected requirements can be satisfied reasonably easily by transforming the FSM into an equivalent minimal FSM, by adding transitions, and by adding a reset operation, respectively. There has been some research on generating tests from nondeterministic FSMs [LvBP94, Hie03, Hie04, MCLH05], and there are some FSM test generation algorithms that can weaken the assumption on the number of states, but the length of the tests grows exponentially with each additional SUT state [LY96, Section 4.7.2].

At MBT 2004 [GKP05], Kervinen and Virolainen [KV05] described an experimental evaluation (on 45 variants of a distributed conference chat system) of the fault-detection speed of seven test generation algorithms for EFSMs, ranging from purely random to sophisticated adaptive algorithms. They found that a *greedy random algorithm* (which gives higher priority to unused transitions) worked surprisingly well, but the highest fault-detection was with adaptive or pessimistic algorithms that give high priority to sending new inputs or receiving new outputs from the SUT.

The ModelJUnit library is available for download from this book's website, and the Qui-Donc model is included as one of the examples.