How to design extended finite state machine test models in Java

Link to publication record in USC Research Bank:
http://research.usc.edu.au/vital/access/manager/Repository/usc:13381

Document Version:
Author accepted manuscript (postprint)

Citation for published version:

Copyright Statement:
Copyright © 2012 From How to design extended finite state machine test models in Java by Utting, Mark. Reproduced by permission of Taylor and Francis Group, LLC, a division of Informa plc. This material is strictly for personal use only. For any other use, the user must contact Taylor & Francis directly at this address: permissions.mailbox@taylorandfrancis.com. Printing, photocopying, sharing via any means is a violation of copyright.

General Rights:
Copyright for the publications made accessible via the USC Research Bank is retained by the author(s) and / or the copyright owners and it is a condition of accessing these publications that users recognize and abide by the legal requirements associated with these rights.

Take down policy
The University of the Sunshine Coast has made every reasonable effort to ensure that USC Research Bank content complies with copyright legislation. If you believe that the public display of this file breaches copyright please contact research-repository@usc.edu.au providing details, and we will remove the work immediately and investigate your claim.
How to Design Extended Finite State Machine Models in Java

Dr Mark Utting
The University of Waikato
New Zealand

Contents

1 Introduction .................................................. 2
  1.1 What is Model-Based Testing? ......................... 3
  1.2 What are the Pros and Cons? ......................... 3
2 Different Kinds of Models .................................. 5
3 How to Design a Test Model ................................ 7
  3.1 Designing an FSM Model ............................... 8
  3.2 From FSM to EFSM: Writing Models in Java ....... 12
4 How to Generate Tests with ModelJUnit ............... 15
5 Writing your own model-based testing tool ............ 18
6 Automating the Execution of Tests ....................... 20
  6.1 Offline Testing ......................................... 21
  6.2 Online Testing ......................................... 21
  6.3 Test Execution Results ................................ 25
  6.4 Mutation analysis of the effectiveness of our testing 26
  6.5 Testing with large amounts of data ................. 27
7 Testing an Embedded System ............................... 28
  7.1 The SIM Card Model .................................. 29
  7.2 Connecting the test model to an embedded SUT ..... 33
8 Related Work and Tools ................................... 35
9 Conclusions .................................................. 36
References ..................................................... 36

Abstract

This chapter is a tutorial that teaches you how to design extended finite state machine (EFSM) test models for a system that you want to test. EFSM models are more powerful and expressive than simple finite state machine (FSM) models, and are one of the most commonly used styles of models for model-based testing, especially for embedded systems. There are many languages and notations in use for writing EFSM models, but in this tutorial we write our EFSM models in the familiar Java programming language. To generate tests from these EFSM models we use ModelJUnit, which is an open-source tool that supports several stochastic test generation algorithms, and we also show how to write your own model-based testing tool. We show how
EFSM models can be used for unit testing and system testing of embedded systems, and for offline testing as well as online testing.

**Keywords**
Model-based testing, extended finite state machines, stochastic testing, Java, ModelJUnit.

1 **Introduction**

Above all others, the key skill that is needed for model-based testing is the ability to write good test models that capture just the essential aspects of your system under test (SUT). This chapter focuses on developing the skill of modeling for MBT. After this introduction, which gives an overview of MBT and its pros and cons, Section 2 compares two of the most common styles of models used for MBT: SUT input models and state-based FSM/EFSM models and discusses their suitability for embedded systems. Then in Section 3 we develop some simple graphical FSM test models for testing a well-known kind of Java collection (Set<E>), and show how this model can be expressed as an EFSM model in Java.

Section 4 illustrates how the ModelJUnit tool (ModelJUnit 2010) can be used to generate a test suite from this model, and discusses several different kinds of test generation algorithms.

Section 5 describes one of the simplest test generation algorithms possible, and shows how you can implement a complete model-based testing tool in just a couple of dozen lines of code, using Java reflection. Section 6 turns to the practical issues of connecting the generated tests to some implementation of Set<E>, and reports on what happens when we execute those tests on a HashSet<String> object, and on an implementation of Set<String> that has an off-by-one bug. It also describes how we can estimate the strength of the generated test suite using SUT code coverage metrics and the Jumble mutation analysis tool (Jumble 2010). As well as illustrating general EFSM testing techniques, Sections 3-6 are also useful as a brief tutorial
introduction to using ModelJUnit.

Section 7 discusses the modeling and testing of a larger, embedded system example - a subset of the GSM 11-11 protocol used within mobile phones, Section 8 discusses related work and tools, and Section 9 draws some brief conclusions.

1.1 What is Model-Based Testing?

The basic idea of model-based testing is that instead of designing dozens or hundreds of test cases manually, we design a small model of the desired behavior of the SUT, and then select an algorithm to automatically generate some tests from that model (El-Far & Whittaker, 2002; Utting & Legeard, 2007). In this chapter, most of the models that we write will be state machines, which have some internal state that represents the current state of the SUT, and some actions that represent the behaviors of the SUT. We will express these state machine models in the Java programming language, so some programming skills will be required when designing the models. The open source ModelJUnit tool can then take one of these models, use reflection to automatically explore the model, visualize the model, and generate however many test cases you want. It can also measure how well the generated tests cover the various aspects of the model, which can give us some idea of how comprehensive the test suite is.

1.2 What are the Pros and Cons?

Like all test automation techniques, model-based testing has advantages and disadvantages. One of the advantages is that generating the tests automatically can save large amounts of time, compared to designing tests by hand. However, this is partially offset by the time taken to design the test model. Most published case studies show that model-based testing reduces overall costs (Bernhard, et. al., 2004; Dalal, et. al., 1999; Farchi, et. al., 2002; Horstmann, et. al., 2005; Jard, et. al., 2005), typically by 20-30%, but sometimes more dramatically – up to 90%
Another advantage of model-based testing is that it is easy to generate lots of tests, far more than could be designed by hand. For example, it can be useful to generate and execute thousands of tests overnight, with everything automated. Of course, having more tests does not necessarily mean that we have better tests. But model-based testing can produce a test suite that systematically covers all the combinations of behavior in the model, and this is likely to be less ad-hoc than a manually design test suite where it is easy to miss some cases. Case studies have shown that model-based test suites are often as good at fault-detection as manually-designed test suites (Bernhard, et. al., 2004; Dalal, et. al., 1999; Farchi, et. al., 2002; Pretschner et. al., 2005). In addition, model-based test suites can be better at detecting requirements errors than manually-designed test suites, because typically half or more of all the faults found by a model-based test suite are because of errors in the model (Stobie, 2005). Detecting these model errors is very useful, since they often point to requirements issues and misunderstandings about the expected behavior of the SUT. The process of modeling the SUT exposes requirements issues as well.

The main disadvantage of model-based testing is the time and expertise necessary to design the model. A test model has to give an accurate description of the expected SUT behavior, so precise executable models are needed. They may be expressed in some programming language, in a precise subset of UML with detailed state machines, or using some finite state machine notation such as graphs. So the person designing the model needs to have some programming or modeling skills as well as SUT expertise. It takes some experience to be able to design a test model at a good level of abstraction, so that it is not overly detailed and large, but it still captures the essence of the SUT that we want to test. This chapter will give examples of how to develop such models for several different kinds of SUT.
One last advantage that we must mention is evolution. When requirements change, updating a large manually-designed test suite can be a lot of work. But with model-based testing it is not necessary to update the tests – we can just update the test model and regenerate a new test suite. Since a good test model is much smaller than the generated test suite, this can result in faster response to changing requirements.

2 Different Kinds of Models

The term ‘model-based testing’ can be used to describe many different kinds of test generation (Utting and Legear, 2007, page 7). This is because different kinds of models are appropriate for different kinds of SUT. Two of the most widely-used kinds of models for MBT are input models and finite state models, so we shall start with a brief overview and comparison of these two kinds.

If your SUT is batch-oriented (it takes a collection of input values, processes them and then produces some output), then one simple kind of model is to just define a small set of test values for each input variable. For example, if we are testing a print function that must print several different kinds of documents onto several different kinds of printers and work on several different operating systems, we might define an input model that simply defines several important test values for each input variable:

- **document:** {plain text, rich text+images, html+images, PDF}
- **printer:** {color inkjet printer, black&white laser, postscript printer}
- **op.system:** {Windows XP, Windows Vista, Linux, Mac OS X}

Given this input model, we could then choose between several different algorithms to generate a test suite. If we want to test all combinations of these test inputs, our test suite would contain 4×3×4=48 test cases. If we want to test all pairs of test input values (Czerwonka, 2008) then 16
test cases would suffice. If we are happy with the dangerous strategy of testing *all input values* but ignoring any interactions between different choices, then 4 test cases could cover all the test input values. This is an example of how we can model the possible inputs of our SUT in a very simple way, and then choose a test generation strategy/algorithmb to generate a test suite from that model of the inputs. This kind of input-only model is useful for generating test inputs in a systematic way, but it does not help us to know what the expected output is, or to determine whether the test passes or fails. Another example of input-only models is grammar-based testing (Coppit and Lian, 2005), where various random generation algorithms are used to generate complex input values (such as sample programs to test a compiler, or SQL queries to test a database system) from a regular expression or a context free grammar.

In this chapter, we focus on testing state-based SUTs, where the behavior of the SUT varies depending upon what state it is in. For such systems, our test cases usually contain a sequence of actions that interact with the SUT, sending it a sequence of input commands and values, as well as specifying the expected outputs of the SUT. The output of the SUT depends on the current state of the SUT, as well as upon the current input value. For example, if we call the `isEmpty()` method of a Java collection object, it will sometimes return `true` and sometimes `false`, depending on whether the internal state of the collection object is empty or not.

Similarly, if we send a ‘TurnLeft’ command to a wheelchair controller, it may respond differently depending on whether the wheelchair is currently moving or stationary. Embedded systems that contain software are usually best modeled as state-based systems.

For these state-based systems, it is important to use a richer state-based model of the expected behavior of the SUT that keeps track of the current state of the SUT. This means that the model can not only be used to generate input values to send to the SUT, but it can also tell us the
expected response of the SUT, because the model knows roughly what state the SUT is in. For modeling state-based systems, it is common to use finite state machines or UML state machines (Binder, 1999; Jacky et. al., 2008; Lee and Yannakakis, 1996; Utting and Legeard, 2007). In this chapter we will see how one style of extended finite state machine can be written in Java and used to generate test sequences that send input values and actions to the SUT as well as checking the expected SUT outputs. By using this kind of rich model of the SUT, we can generate test cases from the model automatically, and those test cases can automate the verdict assignment problem of deciding whether each test has passed or failed when it is executed.

If you want the generated tests to automate the pass/fail verdict, your model must capture the current state or expected outputs of the SUT, so use a finite-state model, not an input-only model.

3 How to Design a Test Model

We will start by designing a test model for a very small system that we want to test. We will model the Java Set<E> interface, which is an interface to a collection of objects of type E. In later sections we will generate tests from this model and execute those tests on a couple of different implementations of sets.

Here is a summary of the main methods defined in the Set<E> interface. We divide them into two groups: the mutator methods that can change the state of the set, and the query methods that return information about the set but do not change its state.
The first step of modeling any embedded system is the same: identify the input commands that change the state of the SUT and the query/observation points that allow us to observe the state of the SUT without changing its state.

### 3.1 Designing an FSM Model

To understand the idea of a state-based model, let’s start by drawing a diagram of the states that a set may go through as we call some of its mutator methods. Starting from a newly constructed empty set, imagine that we add some string called `s1`, then add a second string `s2`, then remove `s2`, then remove `s1` to get an empty set again. If we draw a diagram of this sequence of states, we get:

Each circle represents one state of the set (a snapshot of what we would see if we could look inside the set object), with the contents of the set written inside the circle. Each arrow represents an action (a call to a mutator method) that changes the set from one state to another state. Of course, a moment’s thought makes us realize that the first and last states are both empty and are actually indistinguishable. All our query methods give exactly the same results for a newly constructed empty set as they do for a set that has just had all its members removed. So we
should redraw our state diagram to merge these two states into one. Similarly for the two states that contain just the $s1$ string. They are indistinguishable, so should be merged. This gives us a smaller diagram, where some of the arrows form loops.

This state diagram is a big improvement over our first state diagram, because it has several loops, and these loops give us more ways of going through the diagram and generating tests. Note that any path through the state diagram defines a sequence of method calls, and we can view any sequence as a test sequence.

The more loops, choices, and alternative paths we have in our model the better, because they enable us to generate a wider variety of test sequences.

For example, the left-most loop tells us that the $\text{remove}(s1)$ method undoes the effect of the $\text{add}(s1)$ method, because it returns to the same empty state. So no matter how many times we go around the $\text{add}(s1); \text{remove}(s1)$ loop, the set should still be empty. Similarly, the right-most loop shows that $\text{remove}(s2)$ undoes the effect of the $\text{add}(s2)$ method.

We don't want to just generate lots of test sequences, we also want to be able to execute each test sequence on a SUT and automatically determine whether the test has passed or failed. There are two ways we can do this. For methods that return results, we can annotate each transition of our state diagram with the expected result of each method call. For example, the $\text{add}(s1)$ transition from the empty state should return true, because the $s1$ string was not a member of the empty set - so we could write this transition as $\text{add}(s1)/\text{true}$ to indicate the expected result.
The other way of checking whether a test sequence has passed or failed is to check that the internal state of the SUT agrees with the expected state of the model. It is not always possible to do this, because if the internal state of the SUT is private, we may not be able to observe it. But most SUTs provide a few query methods that give us some information about the current state of the SUT, and this allows us to check if that state agrees with our model. For our Set<E> example, we can use the size() method to check that a SUT contains the expected number of strings, and we can use the contains(String) method to check if each of the expected strings is in the set. In fact, it is a good strategy to call as many of the query methods as possible after each state transition, since this helps to test all the query methods (checking that they are consistent with each other) and also verifies that the SUT state is correct. We could explicitly show every query method as a self-transition in our state diagram, but this would clutter the state diagram too much. So we will show only the mutator methods in our state diagrams here, but we will see later how the query methods can be added into the model after each transition.

Our state diagram is already a useful little test model that captures some of the expected behavior of a Set<E> implementation, but it does not really test the full functionality yet. The clear() method is never used, and we are testing only two strings so far. We need to add some more transitions and states to obtain a more comprehensive model. This raises the most important question of model-based testing:

**How big does our model have to be?**

The answer usually is, the smaller the better. A small model is quicker to write, easier to understand, and will not give an excessive number of tests. A good model will have a high level of abstraction, which means that it will omit all details that are not essential for describing the behavior that we want to test. However, we still want to meet our test goals, which in this case is
to test all the mutator methods. So we will add some clear() transitions into our model. Also, it is often a good goal to ensure that the model is complete, which means that we have modeled the behavior of every mutator method call in every state. Our state diagram above calls add(s1) from the empty state, but not from the other states, so it is currently incomplete. If we expand it to include all five actions (clear(), add(s1), add(s2), remove(s1), remove(s2)) in every state, we get the state diagram shown in Figure 1.

Figure 1. Finite state diagram for Set<E> with two strings, s1 and s2.
<insert UttingFigureFSDforSetE.pdf here>

Note how our goal of having a complete model forced us to consider several additional cases that we might not have considered if we were designing test sequences in a more ad-hoc fashion. For example, the add(s1) transition out of the s1 state models the behavior of add(s1) when the string s1 is already in the set, and checks that we do not end up with two copies of s1 in the set. Similarly, the remove(s1) transition out of the s2 state models what should happen when the member to be removed is not in the set - the remove method should leave the set unchanged and should return false. The clear() transition out of the empty state might not have occurred to a manual test designer, but serves the useful purpose of ensuring that clear() can be called
multiple times in a row without crashing. The point is that designing a model (especially a complete model) leads us to consider all the possible cases in a very systematic way, which can improve the quality of our testing, and is a good way of finding omissions and errors in the requirements (Stobie, 2005).

The remaining question about our model that we should discuss is how many different string values should we test? Why have we tested just two strings? A real implementation can handle hundreds or millions of strings, so shouldn't we test large numbers of strings too? This is another question about how abstract our model should be. To keep our model small, we want to model as few strings as possible, but still exercise the essential features of sets. Zero strings would be rather uninteresting, since the set would always be empty. One string would mean that the set is either empty or contains just that one string. This would allow us to check that the set ignores duplicate adds and duplicate removes, but would not allow us to test that adding a string leaves all other strings in the set unchanged. Two is the minimum number of strings that covers the main behaviors of a set, so this is the best number of string to use in our model. If we expanded our model to three different strings, it would have 8 states and 7 actions, with a total of 56 transitions. This would be significantly more time consuming to design, but would give little additional testing power.

One of the key skills of developing good test models is finding a good level of abstraction, to minimize the size of the model, while still covering the essential SUT features that you want to test.

3.2 From FSM to EFSM: Writing Models in Java

Embedded systems often have quite complex behavior, so require reasonably large models to accurately summarize their behavior. As models become larger, it quickly becomes tedious to
draw them graphically. Instead, we will write them as Java classes, following an EFSM style. An extended finite state machine is basically a finite state machine with some state variables added to the model to keep track of more details about the current SUT state, and actions (code that updates the state variables) added to the transitions. These features can make models much more concise, because the state variables can define many different states, and one Java method can define many similar transitions in the model. We will use the ModelJUnit style of writing the models, because it is simple and effective.

ModelJUnit is an open source tool that aims to be the simplest possible introduction to model-based testing for Java programmers (Utting and Legeard, 2007). The models are written in Java, so that you do not have to learn some new modeling language. In fact, a model is just a Java class that implements a certain interface (FsmModel). The state variables of the class are used to define all the possible states of the state machine model, and the ‘Action’ methods of the Java class define the transitions of the state machine model. Listing 2 shows some Java code that defines our two-string test model of the Set<E> interface – we model just the add and remove operations at this stage.
Listing 2. Java code for the SimpleSet model.

```java
/** A model of a set with two elements: s1 and s2. */
public class SimpleSet implements FsmModel {
    protected boolean s1, s2;
    public Object getState() {
        return (s1 ? "T" : "F") + (s2 ? "T" : "F");
    }
    public void reset(boolean testing) {
        s1 = false;
        s2 = false;
    }
    @Action public void addS1() {s1 = true;}
    @Action public void addS2() {s2 = true;}
    @Action public void removeS1() {s1 = false;}
    @Action public void removeS2() {s2 = false;}
    @Action public void clear() {s1 = false; s2 = false;}
}
```

We now discuss each feature of this class, showing how it defines our two-string test model.

Line 02 defines a class called SimpleSet and says that it implements the FsmModel interface defined by ModelJUnit. This tells us that the class can be used for model-based testing, and means that it must define the getState and reset methods. Line 04 defines a Boolean variable for each of the two strings that we are interested in. The programmer realized that the two strings can be treated independently, and that all we need to know about each string is whether it is in the set or not. So when the variable s1 is true, it means that the first string is in the set, and when the variable s2 is true, it means that the second string is in the set. (We will decide on the precise contents of the two strings later). Choosing the state variables of the model is the step that requires the most insight and creativity from the programmer.

Lines 06-07 define the getState() method, which allows ModelJUnit to read the current state of the model at any time. It returns a string that shows the values of the two Boolean variables, with each Boolean converted to a single 'T' or 'F' character to make the state names shorter.

Lines 09-10 define the reset method, which is called each time a new test sequence is started. It sets both Boolean variables to false, meaning that the set is empty.
The remaining lines of the model give five action methods. These define the transitions of the state machine, because the code inside these methods changes the state variables of the model.

For example, the addS1 method models the action of adding the first string into the model, so it sets the s1 flag to true to indicate that the first string should now be in the set. These action methods are marked with a @Action annotation, to distinguish them from other auxiliary methods that are not intended to define transitions of the model.

4 How to Generate Tests with ModelJUnit

ModelJUnit provides a GUI that can load a model class, explore that model interactively or automatically, visualize the state diagram that the model produces, generate any number of tests from the model, and analyze how well the generated tests cover the model. If we compile our SimpleSet model (using a standard Java compiler) and then load it into the ModelJUnit GUI, we see something like
Figure 3, where the 'Test Design' panel shows several test generation options that we can choose between.
If we accept the default options and use the ‘Random Walk’ test generation algorithm to generate the default size test suite of 10 tests, the small test sequence shown in the left panel of
Figure 3 is generated. Each triple \((Sa, Action, Sb)\) indicates one step of the test sequence, where \(Action\) is the test method that is being executed, starting in state \(Sa\) and finishing in state \(Sb\). For example, the first line tells us to start with an empty set (state='FF'), add the second string, and then check that the set corresponds to state FT (that is, it contains the second string but not the first string). Then the second and third lines check that adding then removing the first string brings us back to the same FT state.

Since this test sequence is generated by a purely random walk, it is not very smart (it tests the \texttt{addS2} action on the full set four times!). However, even such a naive algorithm as this will test every transition (that is, every action going out of every state) if we generate a long enough test sequence. On average, the random walk algorithm will cover every transition of this small model if we generate a test sequence of about 125 steps. More sophisticated algorithms can cover every transition more quickly. For example, ModelJUnit also has a 'Greedy Random Walk' algorithm that gives priority to unexplored paths, and this takes about 55 steps on average to cover every transition. There is also the 'Lookahead Walk' algorithm that does a lookahead of several transitions (3 by default) to find unexplored paths, and this takes only 25 steps to test all 25 transitions. This happens to be the shortest possible test sequence that ensures \textit{all-transitions coverage} of this model. Such minimum-length test sequences are called \textit{Chinese Postman Tours} (Gua, 1962; Thimbleby, 2003), because postmen also have the goal of finding the shortest closed circuit that takes them down every street in their delivery area.

The ModelJUnit GUI is convenient but not necessary. We can also write code that automates the generation of a test suite from our model. For example, the code shown in Listing 4 will generate and print a random sequence of 1000 tests. We put the test generation code inside a main method so that we can execute it from the command line. Another common approach is to
put it inside a JUnit test method so that it can be executed as part of a larger suite of tests.

Listing 4. ModelJUnit code to generate tests by a random traversal of a model.

```java
01:  /** An example of generating tests from the set model. */
02:  public static void main(String[] args)
03:  {  
04:      Tester tester = new RandomTester(new SimpleSet());
05:      tester.addListener(new VerboseListener()); // print the tests
06:      tester.generate(1000); // generate a long sequence of tests
07:  }
```

The code in Listing 4 generates a random sequence of 1000 add, remove, and clear calls.

First we create a 'tester' object and initialize it to use a RandomTester object, which implements a 'Random Walk' algorithm. We pass an instance of our SimpleSet model to the RandomTester object, and it uses Java reflection facilities to determine what actions our model provides. The next line (Line 05) adds a VerboseListener object to the tester, so that some information about each test step will be printed to standard output as the tests are generated. The final line asks the tester to generate a sequence of 1000 test steps. This will include a random mixture of add, remove, and clear actions, and will also perform a reset action occasionally, which models the action of creating a new instance of the Set<E> class that starts off in the empty state. The reset actions also mean that we generate lots of short understandable test sequences, rather than one long sequence.

Although the usual way of generating tests is via the ModelJUnit API, as in Listing 4, for simple testing scenarios the ModelJUnit GUI can write this kind of test generation code for you. As you modify the test configuration options, it displays the Java code that implements the currently-chosen options, so that you can see how to use the API, or cut and paste the code into your Java test generation programs.
5 Writing your own model-based testing tool

ModelJUnit provides a variety of useful test generation algorithms, model visualization features, model coverage statistics, and other features. However, its core idea of using reflection and randomness to generate tests from a Java model is very simple and can easily be implemented in other languages or in application-specific ways.
Figure 5 shows the code for a simple MBT tool that just performs random walks of all the 
@Action methods in a given Java model, with a 1% probability of doing a reset at each step
instead of an action. This occasional reset helps to prevent the test generation from getting stuck
within one part of the model when the model contains irreversible actions. Many variations and
improvements of this basic strategy are possible, but this illustrates how easy it can be to develop
a simple MBT tool that is tailored to your testing environment.
public class SimpleMBT {
    public static final double RESET_PROBABILITY = 0.01;
    protected FsmModel model;
    protected List<Method> methods_ = new ArrayList<Method>();
    protected Random rand_ = new Random(42L); // use a fixed seed

    SimpleMBT(FsmModel model) {
        this.model_ = model;
        for (Method m : model.getClass().getMethods()) {
            if (m.getAnnotation(Action.class) != null) {
                methods_.add(m);
            }
        }
    }

    public String generate() throws Exception {
        if (rand_.nextDouble() < RESET_PROBABILITY) {
            model_.reset(true);
            return "reset";
        } else {
            int i = rand_.nextInt(methods_.size());
            methods_.get(i).invoke(model_, new Object[0]);
            return methods_.get(i).getName();
        }
    }
}

public static void main(String[] args) throws Exception {
    FsmModel model = new SimpleSet();
    SimpleMBT tester = new SimpleMBT(model);
    for (int length = 0; length < 100; length++) {
        System.out.println(tester.generate() + ": " + model.getState());
    }
}

6 Automating the Execution of Tests

We have now seen how we can generate tests automatically from a model of the expected behavior of the SUT. The generated test sequences have been printed in a human-readable format. If our SUT has a physical interface or a GUI, we could manually execute these generated test sequences by pushing buttons, and looking to see if the current state of the SUT
seems to be correct. This can be quite a useful approach for embedded systems that are difficult to connect to a computer.

But it would be nice to automate the execution of the tests, as well as the generation, if possible. This section discusses two alternative ways of automating the test execution: offline and online testing. Both approaches can be used on embedded systems. They both require an API connection to the SUT, so that commands can be sent to the SUT, and its current state can be observed. For embedded SUTs, this API often connects with some hardware, such as digital to analogue converters, which connect to the SUT.

6.1 Offline Testing

One simple, low-tech approach to executing the tests is to write a separate adaptor program that reads the generated test sequence, converts each action in a call to a Set<E> implementation, and then checks the new state of that implementation after the call to ensure that it agrees with the expected state, and report a test failure when they disagree. This adaptor program is essentially a little interpreter of the generated test commands, sending commands to the SUT via the API and checking the results. It plays the same role as a human who interfaces to the SUT and executes the tests manually.

This approach is called offline testing, because the test generation and the test execution are done independently, at separate times and perhaps on separate computers. Offline testing can be useful if you need to execute the generated tests in many different environments or on a different computer to the test generator, or you want to use your existing test management tool to manage and execute the generated tests.

6.2 Online Testing

Online testing is when the tests are being executed on the SUT at the same time as they are being
generated from the model. This gives immediate feedback and even allows a test generation algorithm to observe the actual SUT output and adapt its test generation strategy accordingly, which is useful if the model or the SUT is non-deterministic (Hierons, 2004; Miller et. al., 2005). Online testing creates a tighter, faster connection between the test generator and the SUT, which can permit better error reporting and fast execution of much larger test suites, so it is generally the best approach for embedded systems, unless there are clear reasons why offline testing is preferable.

In this section, we shall extend our SimpleSet model so that it performs online testing of a Java SUT object that implements the Set<E> interface.
Listing 6 shows a Java Model class that is similar to SimpleSet, but also has a pointer (called \textit{sut}) to a \textit{Set<String>} implementation that we want to test. Each of the \texttt{@Action} methods is extended so that as well as updating the state of the model (the \texttt{s1} and \texttt{s2} variables), it also calls one of the SUT methods. For example, after the \texttt{addS1} action sets \texttt{s1} to true (to indicate that string \texttt{s1} should be in the set after this action), it calls \texttt{sut.add(s1)} to make the corresponding change to the SUT object. Then it calls various query methods to check that the updated SUT state is the same as the state of the model (since all of the \texttt{@Action} methods do the same state checks in this example, we move those checks into a method called \texttt{checkSUT()}, and call this at the end of each \texttt{@Action} method).
Listing 6. An extension of SimpleSet that performs online testing.

```java
public class SimpleSetWithAdaptor implements FsmModel {

    protected boolean s1, s2;
    protected Set<String> sut; // the implementation we are testing

    // our test data for the SUT
    protected String str1 = "some string";
    protected String str2 = ""; // empty string

    /** Tests a StringSet implementation. */
    public SimpleSetWithAdaptor(Set<String> systemUnderTest) {
        this.sut = systemUnderTest;
    }

    public Object getState() {
        return (s1 ? "T" : "F") + (s2 ? "T" : "F");
    }

    public void reset(boolean testing) {
        s1 = false; s2 = false; sut.clear(); checkSUT();
    }

    @Action public void addS1() {
        s1 = true; sut.add(str1); checkSUT();
    }

    @Action public void addS2() {
        s2 = true; sut.add(str2); checkSUT();
    }

    @Action public void removeS1() {
        s1 = false; sut.remove(str1); checkSUT();
    }

    @Action public void removeS2() {
        s2 = false; sut.remove(str2); checkSUT();
    }

    /** Check that the SUT is in the expected state. */
    protected void checkSUT() {
        Assert.assertEquals(s1, sut.contains(str1));
        Assert.assertEquals(s2, sut.contains(str2));
        int size = (s1 ? 1 : 0) + (s2 ? 1 : 0);
        Assert.assertEquals(size, sut.size());
        Assert.assertEquals(!s1 && !s2, sut.isEmpty());
        Assert.assertEquals(!s1 && s2,
            sut.equals(Collections.singleton(str2)));
    }
}
```

We have written this online testing class as a standalone class so that you can see the model updating code and the SUT updating code next to each other. An alternative style is to use inheritance to extend a model class (like SimpleSet) by creating a subclass that overrides each action method and adds the SUT actions and the checking code.
A checking method such as `checkSUT()` typically calls one or more of the SUT query methods to see if the expected state (of the model) and the actual state of the SUT agree. For this example, we have decided to test a set of strings, using the two sample strings defined as `str1` and `str2` in
Listing 6. So we can see if the first string is in the set by calling `sut.contains(str1)`, and we expect that this should be true exactly when our model has set the Boolean variable `s1` to true. So we use standard JUnit methods to check that `s1` is equal to `sut.contains(str1)`. We check that relationship between `str1` and `s2` in the same way. We add several additional checks on the `size()`, `isEmpty()`, and `equals(_)` methods of the SUT, partly to gain more confidence that the SUT state is correct, and partly so that we test those SUT query methods. They will be called many times, in every SUT state that our model allows, so they will be well tested. Finally, note that in
Listing 6: we are not checking the return value of `sut.add(_)` but we can easily do this by checking that the return value equals the initial value of the `s1` flag.

Note how each action method updates the model, then updates the SUT in a similar way, then checks that the model state agrees with the SUT state. So as we execute a sequence of these action methods, the model and the SUT are evolving in parallel, each making the same changes, and the checkSUT() method is checking that they agree about what the next state should be.

This nicely illustrates the essential idea behind model-based testing:

```
implement your system twice and run the two implementations
in parallel to check them against each other.
```

But of course, no one really wants to implement a system twice! The trick that makes model-based testing useful is that those two ‘implementations’ have very different goals:

1. the SUT implementation needs to be efficient, robust, scale to large data sets and it must implement all the functionality in the requirements.

2. the model ‘implementation’ can be a vastly simplified system that implements only one or two key requirements, handles only a few small data values chosen for testing purposes, and does not need to be efficient or scalable.

This difference means that it is usually practical to ‘implement’ (design and code) a model in a few hours or a few days, whereas the real SUT takes months of careful planning and coding. We repeat, the key to cost-effective modeling is finding a good level of abstraction for the model.

*Abstraction: deciding which requirements are the key ones that must be tested and which ones can be ignored or simplified for the purposes of testing.*

### 6.3 Test Execution Results

We can use this model to test the `HashSet` class from the standard Java library simply by passing `new HashSet<String>()` to the constructor of our `SimpleSetWithAdaptor` class and then using that to generate any number of tests, either by using the ModelJUnit GUI or by
executing some test generation code similar to Listing 4. When we do this, no errors are detected. This is not surprising, since the standard Java library classes are widely used and thoroughly tested. If we write our own simple implementation of `Set<String>`, and insert an off-by-one bug into its `equals` method (see the `StringSetBuggy` class in the ModelJUnit distribution for details), we get the following output when we try to generate a test sequence of length 60 using the Greedy Random Walk algorithm.

done (FF, addS2, FT)
done (FT, addS1, TT)
done (TT, removeS1, FT)
done (FT, removeS2, FF)
done (FF, removeS2, FF)
FAILURE: failure in action addS1 from state FF due to
  AssertionError: expected:<false> but was:<true>
  ...
Caused by: AssertionError: expected:<false> but was:<true>
  ...
   at junit.framework.Assert.assertEquals(Assert.java:149)
   at SimpleSetWithAdaptor.checkSUT(SimpleSetWithAdaptor.java:123)
   at SimpleSetWithAdaptor.addS1(SimpleSetWithAdaptor.java:87)
   ... 10 more

This pinpoints the failure as being detected by the `sut.equals` call on line 41 of
Listing 6, when checkSUT method was called from the addS1 action with the set being empty. Interestingly, the test sequence shows us that checkSUT had tested the equals method on an empty set several times previously, but the failure did not occur then – it required a removeS1 followed by an addS2 to detect the failure. A manually designed JUnit test suite may not have tested that particular combination, but the random automatic generation will always eventually generate such combinations, and detect such failures, if we let it generate long enough sequences.

If we fix our off-by-one error, then all the tests pass, and ModelJUnit reports that 100% of the transitions of the model have been tested. If we measure the code coverage of this StringSet implementation, which just implements a set as an ArrayList with no duplicate entries, we find that the generated test suite has covered 93.3% of the code (111 out of 119 JVM instructions, as measured by the EclEmma plugin for Eclipse (Emma 2009)). The untested code is the iterator() method, which we did not call in our checkSUT() method, and one exception case to do with null strings.

6.4 Mutation analysis of the effectiveness of our testing

It is also interesting to use the Jumble mutation analysis tool (Jumble 2010) to measure the effectiveness of our automatically generated test suite. Jumble analyses the Java bytecode of a SUT class, creates lots of mutants (minor modifications that cause the program to have different behavior) and then runs our tests on each mutant to see if they detect the error that has been introduced. On this SUT class, StringSet.java, Jumble creates 37 different mutants and reports that our automatically generated tests detect 94% (35 out of 37) of those mutants.
Mutation points = 37, unit test time limit 2.58s
M FAIL: modeljunit.examples.StringSet:44: changed return value
.M FAIL: modeljunit.examples.StringSet:56: 0 -> 1
............................
Score: 94%

This is a high level of error detection, which indicates that our automatically generated tests are testing our simple set implementation quite thoroughly and that our model accurately captures most of the behavior of the Set<E> interface. One of the two mutants that were not detected is in the iterator() method, which we did not test in our model. The other mutant indicates that we are not testing the case where the argument to the equals method is a different type of object (not a set). This is a low-priority case that could be ignored or could easily be covered by a manually-written JUnit test.

6.5 Testing with large amounts of data

What if we wanted to do some performance testing, to test that sets can handle hundreds or thousands of elements? For example, we might know that a SUT like HashSet<E> expands its internal data structures after a certain number of elements have been added, so we suspect that testing a set with only two elements is inadequate.

One approach would be to expand our model so that it uses a bit vector to keep track of hundreds of different strings and knows exactly when each string is in or out of the set. It is not difficult to write such a model, but when we start to generate tests we quickly find that the model has so many states to explore that it will be impossible to test all the states or all the transitions. For example, with 100 strings, the model would have $2^{100}$ states and even more transitions. Many of these states would be similar, so many of the tests that we generate would be repetitive and uninteresting.

A more productive style is to keep our model small (e.g., two or three Boolean flags), but change
our interpretation of one of those Boolean flags \texttt{s2} so that instead of meaning that ‘\textit{str2 is in the set}’ it now means ‘\textit{all the strings "x1", "x2".. "x999" are in the set}’. This leaves the behavior of our model unchanged, and means that all we need to change is the code that updates the SUT.

For example, the \texttt{addS2()} action becomes:

```java
23:   @Action public void addS2()
24a:  {
24b:     s2 = true;
24c:     for (int i=1; i<1000; i++)
24d:          { sut.add("x"+i); }
24e:     checkSUT();
24f:  }
```

With this approach, we can generate the same short test sequence as earlier, and easily cover all the states and transitions of our small model, while the tests can scale up to any size of set that we want. This is another good example of using \textit{abstraction} when we design the model – we decided that even though we want to test a thousand strings, it is probably not necessary to test them all independently – testing two groups of strings should give the same fault-finding power.

> **When possible, it is good to keep the model small and abstract, and make the adaptor code do the donkey work.**

7 Testing an Embedded System

In this section we shall briefly see how this same model-based testing approach can be used to model and test an embedded system such as the \textit{Subscriber Identification Module} (SIM) card embedded in GSM mobile phones. The SIM card stores various data files that contain private data of the user and of the Telecom provider, so it protects these files via a system of access permissions and PIN codes. When a SIM card is inserted into a mobile phone, the phone communicates with the SIM card by sending small packets of bytes that follow the GSM 11.11
standard protocol (Bernard 2004). A summary of some key features of this GSM 11.11 protocol is given in Utting & Legeard (2007, Chapter 9), together with use cases, UML class diagrams and UML state machine models, plus examples of generating tests from those UML models. In this section, we give a brief overview of how the same system can be modeled in Java, and show how we can generate tests that send packets of bytes to the SIM and check the correctness of its responses. We execute the generated tests on a simulator of the SIM card so that we can measure the error detection power using Jumble. The generated tests could equally well be executed on real hardware, if we have a test execution platform with the hardware to connect to the physical SIM and send and receive the low-level packets produced by the tests.

7.1 The SIM Card Model

Our test model of the SIM card is defined in a Java class called SimCard (420 source lines of code), which contains six enumerations, 12 data variables, and 15 actions, plus the usual reset and getState methods. There is also a small supporting class called SimFile (24 source lines of code) that models the relevant aspects of the File objects stored within the SIM – we do not model the full contents of each file - a couple of bytes of data is sufficient to test that the correct file contents are being retrieved. The full source code of this SIM card model is included as one of the example models in the ModelJUnit distribution.

Figure 7 shows all the data variables of the model. The files map models the contents of all the files and directories on the SIM - these are constant throughout testing since this model does not include any write operations. The DF and EF variables model the currently selected directory file and the currently selected elementary file within that directory, respectively. The PIN variable corresponds to the correct PIN number, which is set to 11 by the reset method of the
model, then may be set to 12 or back to 11 by changePIN actions during testing (two PIN numbers are sufficient for testing purposes).

Figure 7. Data variables of the SimCard model.

```java
public class SimCard implements FsmModel {
    public enum E_Status {Enabled, Disabled};
    public enum B_Status {Blocked, Unblocked};
    public enum Status_Word {sw_9000, sw_9404, sw_9405, sw_9804,
                            sw_9840, sw_9808, sw_9400};
    public enum File_Type {Type_DF, Type_EF};
    public enum Permission {Always, CHV, Never, Adm, None};
    public enum F_Name {MF, DF_GSM, EF_LP, EF_IMSI, DF_Roaming, EF_FR, EF_UK};

    // These variables model the attributes within each Sim Card.
    protected static final int GOOD_PUK = 1223; // the correct PUK code
    public static final int Max_Pin_Try = 3;
    public static final int Max_Puk_Try = 10;

    /** This models all the files on the SIM and their contents */
    protected Map<F_Name,SimFile> files = new HashMap<F_Name,SimFile>();
    /** The currently-selected directory (never null) */
    protected SimFile DF;
    /** The current elementary file, or null if none is selected */
    protected SimFile EF;
    /** The correct PIN (can be 11 or 12) */
    protected int PIN;
    /** Say whether PIN-checking is Enabled or Disabled */
    protected E_Status status_en;
    /** Number of bad PIN attempts: 0 .. Max_Pin_Try */
    protected int counter_PIN_try;
    /** True means a correct PIN has been entered in this session */
    protected boolean perm_session;
    /** Set to Blocked after too many incorrect PIN attempts */
    protected B_Status status_PIN_block;
    /** Number of bad PUK attempts: 0 .. Max_Puk_Try */
    protected int counter_PUK_try;
    /** Set to Blocked after too many incorrect PUK attempts */
    protected B_Status status_PUK_block;
    /** The status word returned by each command */
    protected Status_Word result;
    /** The data returned by the Read_Binary command */
    protected String read_data;
    /** The adaptor object that interacts with the SIM card */
    protected SimCardAdaptor sut = null;
```
The next four variables \((\text{status_en}, \text{counter_PIN_try}, \text{perm_session}, \text{and status_PIN_block})\) model all the PIN-related aspects of the SIM security, and the following two variables \((\text{counter_PUK_try} \text{ and status_blocked})\) model the PUK (Personal Unblocking Key) checking – entry of a correct PUK code allows a user to unblock a card that had \text{status_PIN_block} set to Blocked because of three incorrect PIN attempts. However, after 10 incorrect PUK attempts, \text{status_PUK_block} will be set to Blocked, which means that all future attempts to unblock the SIM by entering a correct PUK will fail. When testing a real SIM chip, this effectively destroys the SIM chip, since there is no way of resetting the SIM to normal functionality once it has blocked PUK entry.

Figure 8 shows one of the more interesting methods in the model, Unblock_PIN. This models the user trying to enter a PUK code in order to set the SIM to use a new PIN number, which is typically done after the old PIN is blocked due to three incorrect PIN attempts. The Unblock_PIN method takes the PUK code and the new PIN code as inputs, and these are typically eight digit and four digit integers, respectively. If we chose input values at random, there would be \(10^8 \times 10^4 = 10^{12}\) possible combinations of inputs, most of which would have the same effect. So to focus the test generation on the most interesting cases, we decide to define just two actions that call Unblock_PIN – one with the correct PUK number and a new PIN code of 12, and one with an incorrect PUK number. Repeated applications of the latter action will test the PUK blocking features of the SIM. This illustrates a widely-used test design strategy called equivalence classes (Copeland 2004): when testing a general-purpose method that has many possible combinations of input values, we manually choose just a strategic few of those input combinations – one for each different kind of behavior that is possible. In our Unblock_PIN
model method, the choice of a good or bad PUK code determines the outcome of the second if condition (puk == GOOD_PUK), and since all the other if conditions are determined by the state variables of the model, these two PUK values are sufficient for us to test all the possible behaviors of this model method. This style of having several @Action methods that all call the same method, with carefully chosen different parameter values, is often used in ModelJUnit to reduce the size of the state space that is explored during testing, while still ensuring that the important different behaviors are tested.

Figure 8. The Unblock_PIN actions of the SimCard model.

```java
@Action public void unblockPINGood12() { Unblock_PIN(GOOD_PUK,12); }
@Action public void unblockPINBad()   { Unblock_PIN(1223446,11); }
public void Unblock_PIN(int puk, int newPin) {
    if (status_block == B_StatusBlocked) {
        result = Status_Word.sw_9840; /*@REQ: Unblock_CHV1 @*/
    } else if (puk == GOOD_PUK) {
        PIN = newPin;
        counter_PIN_try = 0;
        counter_PUK_try = 0;
        perm_session = true;
        status_PIN_block = B_Status.Unblocked;
        result = Status_Word.sw_9000;
        if (status_en == E_StatusDisabled) {
            status_en = E_StatusEnabled; /*@REQ: Unblock5 @*/
        } else {
            // leave status_en unchanged
        } /*@REQ: Unblock7, Unblock2 */
    } else if (counter_PUK_try == Max_Puk_Try - 1) {
        System.out.println("BLOCKED PUK!!! PUK try counter="+counter_PUK_try);
        counter_PIN_block = Max_PIN_Try;
        status_PIN_block = B_Status.Unblocked;
        result = Status_Word.sw_9000;
        if (status_en == E_StatusDisabled) {
            status_en = E_StatusEnabled; /*@REQ: Unblock5 @*/
        } else {
            // leave status_en unchanged
        } /*@REQ: Unblock7, Unblock2 */
    } else { 
        System.out.println("BLOCKED PUK!!! PUK try counter="+counter_PUK_try);
        counter_PIN_block = Max_PIN_Try;
        status_PIN_block = B_Status.Unblocked;
        result = Status_Word.sw_9000;
        if (status_en == E_StatusDisabled) {
            status_en = E_StatusEnabled; /*@REQ: Unblock5 @*/
        } else {
            // leave status_en unchanged
        } /*@REQ: Unblock7, Unblock2 */
    }

    if (sut != null) {
        sut.Unblock_PIN(puk, newPin, result);
    }
}
```
7.2 Connecting the test model to an embedded SUT

The last two lines of the Unblock_PIN method in Figure 8 show how we can connect the model to an implementation of the SIM, via some adapter code (shown in Figure 9) that handles the low-level details of assembling, sending, and receiving packets of bytes.

Figure 9. Adapter class that connects SimCard model to a SIM implementation.

```java
public class SimCardAdaptor {
    protected byte[] apdu = new byte[258];
    protected byte[] response = null;
    protected GSM11Impl sut = new GSM11Impl();

    /** Sets up the first few bytes of the APDU, ready to send to the SIM. */
    protected void initCmd(int cmdnum, int p1, int p2, int p3) {
        for (int i = 0; i < apdu.length; i++)
            apdu[i] = 0;
        apdu[0] = (byte)0xA0;
        apdu[1] = (byte)(cmdnum & 0xFF);
        apdu[2] = (byte)(p1 & 0xFF);
        apdu[3] = (byte)(p2 & 0xFF);
        apdu[4] = (byte)(p3 & 0xFF);
    }

    public void Unblock_PIN(int puk, int newPin, SimCard.Status_Word result) {
        initCmd(0x2C, 0x00, 0x00, 0x10);
        setChv(5, puk); // pack the PUK into bytes 5..12
        setChv(13, newPin); // pack the PIN code into bytes 13..20
        response = sut.cmd(apdu);
        checkStatus(result, 0);
    }
    ...
```

The SimCard model defines quite a large finite state machine. If we analyse the state variables of the model and think about which combinations of values are possible, we find that there are 10 possible directory/file settings (DF and EF), two PIN values, 4 values for counter_PIN_try and 11 values for counter_PUK_try, plus several other flags (but their values are generally correlated with other data values), so there are likely to be around 10×2×4×11=880 states in the
model and up to 15 times that number of transitions (since there are 15 @Action methods in the model). This is too large for us to want to test exhaustively, but it is easy to use the various random walk test generation algorithms of ModelJUnit to generate test suites of any desired length. If we are doing online testing, we can just keep generating and executing tests until we find an error, or until a certain number of seconds or hours has elapsed. The randomness aspect of the generation means that the longer we test, the more thoroughly we cover all the possible sequences of actions.

*Figure 10 shows how effective a simple random walk test generation algorithm can be at finding errors.* To measure the error detection power of the generated tests, we wrote a software simulation of a SIM chip in Java, and used Jumble to generate 298 different mutants of that simulation. Each mutant is like a potential bug in the SUT. Then we generate different size test suites and measure what percentage of the mutants/bugs is detected by the test suite, and what percentage of the lines of code in the SUT is executed by the test suite.

Figure 10 shows that the bug-detection rate rises rapidly up to a test length of 1000, and then more slowly to the maximum of 85% of mutants detected by a test suite with 100,000 steps (the remaining mutants were mostly modifying data in parts of the sample SIM files that were outside the scope of the model, so the mutations were not detectable with this model). The SUT code coverage follows a similar pattern and reaches a maximum of 95.8% of lines executed by the generated tests. It might seem impractical to execute test suites this large, but the total generation and online execution time (using the simulated Java SIM as the SUT) of a million test steps takes less than 7 seconds on an Intel Core 2 Duo 2.5GHz, and 100,000 tests takes less than one second, so test suites this large are quite practical. Our GSM model is reasonably large (more than 570 states and 8000 transitions), so it is worthwhile to generate lots of tests so that we cover lots of different scenarios. For example, an average test suite of length 100,000 tests the
blocked PIN situation about 3000 times, uses the PUK about 6000 times and tests the blocked-PUK situation (which destroys the SIM card) about 3 times.

Figure 10. How SUT coverage and error detection increases with test sequence length.

8 Related Work and Tools

There are many other languages and tools that can also be used for EFSM model-based testing. In this chapter we have written EFSM models in Java, but the design strategies are transferable to most kinds of state-based models and tools. For example, the NModel tool (Jacky, et. al. 2008) uses a similar style of model but in C#. Spec Explorer (Veanes, et. al. 2008) is a model-based testing tool from Microsoft that uses EFSM models written in C#, but can also combine these with scenarios written in a regular-expression style. Recent versions of Spec Explorer are integrated with Visual Studio and have GUI facilities for visualizing the EFSMs and the generated tests. There are also several MBT tools that use EFSM models written as UML state
machines, with the actions written in Java (Conformiq 2009), or in OCL (Smartesting 2009). The EFSM modeling principles are similar across all these tools, but they differ in their algorithms for generating tests, and their facilities for visualizing models and tests. ModelJUnit differs by taking a very simple random-exploration approach to test generation (and Section 5 shows how you build a similar test generation tool in any language that supports reflection), and by providing an API for generating tests, whereas the other tools use a GUI or command line program to generate tests.

9 Conclusions

We have explored the key ideas of model-based testing: creating a small model of the system that you want to test, and then using various kinds of tools to automatically generate a test suite from that model. The ModelJUnit philosophy is to use Java as the modeling language, because it is familiar, and to use reflection plus some simple random choice algorithms to generate the test suites. This is a simple approach that can be implemented quite easily in any language that supports reflection. It can be used to generate offline test suites, or for online testing. Model-based testing allows you to automatically generate and execute a large number of tests. With careful design of the model, it can give good coverage of the SUT behavior and code. The problem of maintaining the test suite disappears, since it can be regenerated at any time. Instead, you must maintain the test model, but this is typically smaller and less repetitive than a test suite.

References


Springer-Verlag.


