11 Model-based testing

11.1 What is model-based testing

Model-based testing has become a bit of a buzzword in recent years, and we have noticed that people are using the term for a wide variety of test generation techniques. The following are the four main approaches known as model-based testing.

1. Generation of test input data from a domain model
2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

We will briefly describe these approaches and then explain why this book focuses mostly on the third meaning of model-based testing and covers the other meanings more briefly.

When model-based testing is used to mean the generation of test input data, the model is the information about the domains of the input values and the test generation involves clever selection and combination of a sub- set of those values to produce test input data. For example, if we are test- ing a procedure that has three inputs, A : {red,green,yellow}, B : 1..4, and C : {car,truck,bike}, then we might use a pairwise algorithm[1] to generate a minimal set of tests that exercise all possible pairs of input values. For this example, a good pairwise algorithm would generate just 12 tests[2] rather than the $3 \times 4 \times 3 = 36$ tests that we would need if we tried all possible combinations. The automatic generation of test inputs is obviously of great practical importance, but it does not solve the complete test design problem because it does not help us to know whether a test has passed or failed. The second meaning of model-based testing uses a different kind of model, which describes the expected environment of the SUT. For example, it might be a statistical model of the expected usage of the SUT [Pro03] (operation frequencies, data value distributions, etc.). From these environment models it is possible to generate sequences of calls to the SUT. However, like the previous approach, the generated sequences do not specify the

[1] See Chapter XXXX for further discussion of pairwise testing, and the Pairwise website, http://www.pairwise.org, for tools, articles, and case studies on pairwise testing.

[2] For example, the 12 triples (red,1,car), (red,2,truck), (red,3,bike), (red,4,car), (green, 1,truck), (green,2,car), (green,3,truck), (green,4,bike), (yellow,1,bike), (yellow,2,bike), (yellow,3,car), (yellow,4,truck) cover all pairs of input values. That is, all 12 combinations of color and number appear; so do all 12 combinations of number and vehicle and all 9 combinations of color and vehicle.

expected outputs of the SUT. It is not possible to predict the output values because the environment model does not model the behavior of the SUT. So it is difficult to determine accurately whether a test has passed or failed—a crash/no-crash verdict may be all that is possible.

The third meaning of model-based testing is the generation of executable test cases that include oracle information, such as the expected output values of the SUT, or some automated check on the actual output values to see if they are correct. This is obviously a more challenging task than just generat- ing test input data or test sequences that call the SUT but do not check the results. To generate tests with oracles, the test generator must know enough about the expected behavior of the SUT to be able to predict or check the SUT output values. In other words, with this definition of model- based testing, the model must describe the expected behavior of the SUT, such as the relationship between its inputs and outputs. But the advantage of this approach is that it is the only one of the four that addresses the whole test design problem from choosing input values and generating sequences of operation calls to generating executable test cases that include verdict information. The fourth meaning of model-based testing is quite different: it assumes that we are given a very abstract description of a test case, such as a UML sequence diagram or a sequence of high-level procedure calls, and it focuses on transforming that abstract test case into a low-level test script that is executable. With this approach, the model is the information about the structure and API (application programming interface) of the SUT and the details of how to transform a high-level call into executable test scripts. We discuss this process in more detail in Chapter 8. The main focus of this book (Chapters 3 to 7 and 9 and 10) is the third meaning of model-based testing: the generation of executable test cases that include oracle information, based on models of the SUT behavior. This generation process includes the generation of input values and the sequencing of calls into test sequences, but it also includes the generation of oracles that check the SUT outputs. This kind of model-based testing is more sophis- ticated and complex than the other meanings, but it has greater potential paybacks. It can automate the complete test design process, given a suitable model, and produces complete test sequences that can be transformed into executable test scripts. With this view of model-based testing, we define model-based testing as the automation of the design of black-box tests. The difference from the usual black-box testing is that rather than manually writing tests based on the requirements documentation, we instead create a model of the expected SUT behavior, which captures some of the requirements. Then the model- based testing tools are used to automatically generate tests from that model.

Model-based testing is the automation of the design of black-box tests.

That leads us to two questions: What is a model? What notation should we use to write models? Here are two illuminating definitions of the word model, from the American Heritage Dictionary [Ame00]:

- A small object, usually built to scale, that represents in detail another, often larger object.

- A schematic description of a system, theory, or phenomenon that accounts for its known or inferred properties and may be used for further study of its characteristics.

These definitions show the two most important characteristics of models that we want for model-based testing: the models must be small in relation to the size of the system that we are testing so that they are not too costly to produce, but they must be detailed enough to accurately describe the char- acteristics that we want to test. A UML class diagram or an informal use case diagram by itself is not precise or detailed enough for model-based test- ing; some description of the dynamic behavior of the system is needed. Yes, these two goals (small, detailed) can be in conflict at times. This is why it is an important engineering task to decide which characteristics of the system should be modeled to satisfy the test objectives, how much detail is useful, and which modeling notation can express those characteristics most natu- rally. Chapter 3 gives an introduction to various kinds of modeling notations and discusses guidelines for writing effective models for testing purposes. Once we have a model of the system we want to test, we can then use one of the model-based testing tools to automatically generate a test suite from the model. There are quite a few commercial and academic model-based testing tools available now, based on a variety of methods and notations. Many of the tools allow the test engineer to guide the test generation process to control the number of tests produced or to focus the testing effort on certain areas of the model. The output of the test case generator will be a set of abstract test cases, each of which is a sequence of operations with the associated input values and the expected output values (the oracle). That is, the generated test cases will be expressed in terms of the abstract operations and values used by the model. The next step is to transform (concretize) these abstract test cases into executable test scripts. This may be done by the model-based testing tool, using some templates and translation tables supplied by the test engineer. The resulting executable tests may be produced directly in some program- ming language, such as JUnit tests in Java, or in a dynamic language such as Tcl or Python, or in a dedicated test scripting language. These executable test scripts can then be executed to try to detect failures in the SUT. The execution of the tests may be controlled and monitored by a test execution tool—different varieties of these tools are available for various types of SUT. The process of transforming the abstract test cases into executable tests and executing them is covered in Chapter 8. In the next chapter, we will discuss the benefits and limitations of model-based testing and its impact on the software life cycle. But before that, let us look at a realistic example of model-based testing to get a clearer picture of what it involves.

11.2 model-based testing process

Model-based testing automates the detailed design of the test cases and the generation of the traceability matrix. More precisely, instead of manually writing hundreds of test cases (sequences of operations), the test designer writes an abstract model of the system under test, and then the model-based testing tool generates a set of test cases from that model. The overall test design time is reduced, and an added advantage is that one can generate a variety of test suites from the same model simply by using different test

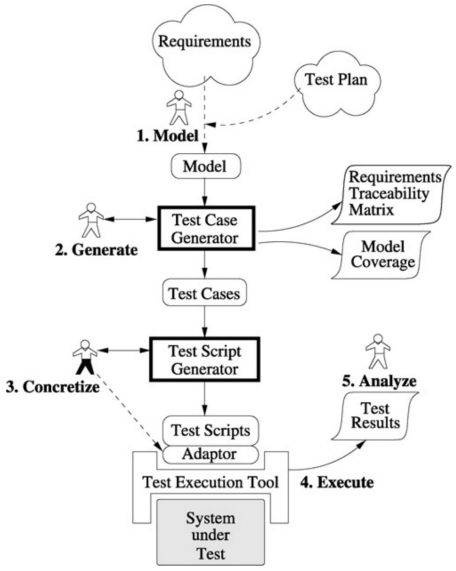

Figure 11.1: The model-based testing process (testing tools are in the boxes with very bold lines).

selection criteria. The model-based testing process can be divided into the following five main steps, as shown in Figure 2.4.

1. Model the SUT and/or its environment.
2. Generate abstract tests from the model.
3. Concretize the abstract tests to make them executable.
4. Execute the tests on the SUT and assign verdicts.
5. Analyze the test results.

Of course, steps 4 and 5 are a normal part of any testing process, even manual testing. Step 3 is similar to the "adaptor" phase of keyword-based testing, where the meaning of each keyword is defined. The first two steps distinguish model-based testing from other kinds of testing. In online model-based testing tools, steps 2 through 4 are usually merged into one step, whereas in offline model-based testing, they are usually separate. But it is still useful to explain the steps separately to ensure a clear understanding of the model- based testing process. We will now give a more detailed description of each step of the process and mention some of the practical issues about using model-based testing in large projects.

The first step of model-based testing is to write an abstract model of the system that we want to test. We call it an abstract model because it should be much smaller and simpler than the SUT itself. It should focus on just the key aspects that we want to test and should omit many of the details of the SUT. Later chapters give detailed guidelines

for how to perform this modeling step using various modeling notations. While writing the model, we may also annotate it with requirements identifiers to clearly document the relationship between the informal requirements and the formal model.

After writing the model, it is advisable to use tools to check that the model is consistent and has the desired behavior. Most modeling notations provide some automated verification tools (such as typecheckers and static analysis tools), as well as some interactive tools (such as animators) that allow us to explore the behavior of the model and check that it is what we expect. The use of an animator is highly recommended for novice modelers, but experienced modelers may prefer to save time by omitting animation—any errors that remain in the model will be noticed during later steps, and the animator can be used then to pinpoint the problem in the model.

The second step of model-based testing is to generate abstract tests from the model. We must choose some test selection criteria, to say which tests we want to generate from the model, because there are usually an infinite number of possible tests. For example, we might interact with the test generation tool to focus on a particular part of the model or to choose a particular model coverage criterion, such as all-transitions, or we might write some test case specifications in some simple pattern language to specify the kinds of test we want generated.

The main output of this step is a set of abstract tests, which are sequences of operations from the model. Since the model uses a simplified view of the SUT, these abstract tests lack some of the detail needed by the SUT and are not directly executable. Most model-based testing tools also produce a requirements traceability matrix or various other coverage reports as additional outputs of this step. The requirements traceability matrix traces the link between functional requirements and generated test cases. This is generally a many-to-many relation: a requirement can be covered by several test cases, and a single test case may exercise several requirements. The coverage reports give us some indications of how well the generated test set exercises all the behaviors of the model. Note that these reports are talking about coverage of the model, not coverage of the SUT—we have not even executed the tests on the SUT yet! For example, a coverage report may give us some coverage statistics for the operations or transitions of the model, tell us what percentage of boolean decisions in the model have been tested with true and false values, or give the coverage results for many other kinds of coverage measure (see Chapter 4). We can use such coverage reports simply for statistical feedback about the quality of the generated test set, or we can use them to identify parts of the model that may not be well tested and investigate why this has happened. For example, if a particular path through the model has no tests generated for it, we could try changing some of the test generation parameters and repeat the test generation step. This is where it can be useful to use an animation tool to investigate the behavior of the path in the model and decide whether the lack of tests is due to an error in the model, a normal feature of the model, or inadequate test generation. In the latter case, if we want to improve the coverage of our test set, we could add an abstract test for this path by hand or give the test generation tool some explicit hints about how to find the desired test.

The third step of model-based testing is to transform the abstract tests into executable concrete tests. This may be done by a transformation tool, which uses various templates

and mappings to translate each abstract test case into an executable test script. Or it may be done by writing some adaptor code that wraps around the SUT and implements each abstract operation in terms of the lower-level SUT facilities. Either way, the goal of this step is to bridge the gap between the abstract tests and the concrete SUT by adding in the low-level SUT details that were not mentioned in the abstract model. One advantage of this two-layer approach (abstract tests and concrete test scripts) is that the abstract tests can be quite independent of the language used to write tests and of the test environment. By changing just the adaptor code or the translation templates, we can reuse the same set of tests in different test execution environments.

The fourth step is to execute the concrete tests on the system under test. With online model-based testing, the tests will be executed as they are produced, so the model-based testing tool will manage the test execution process and record the results. With offline model-based testing, we have just generated a set of concrete test scripts in some existing language, so we can continue to use our existing test execution tools and practices. For example, we might use Mercury TestDirector[3] to manage the tests, execute them regularly, and record the results. The fifth step is to analyze the results of the test executions and take corrective action. For each test that reports a failure, we must determine the fault that caused that failure. Again, this is similar to the traditional test analysis process. As usual, when a test fails, we may find that it is due to a fault in the SUT or we may find that it is due to a fault in the test case itself. Since we are using model-based testing, a fault in the test case must be due to a fault in the adaptor code or in the model (and perhaps also the requirements documents)[4]. So this is another place where we get feedback about the correctness of the model. In our experience, the first execution of the test set usually sees a high percentage of the tests fail—typically because of some minor errors in the adaptor code. Once these are fixed, the remaining failures are more interesting and require deeper analysis to find the fault. Perhaps roughly half of these failures will result from faults in the SUT and the other half from faults in the model and the requirements. However, this ratio can vary widely, depending upon the experience of the testers, the kind of project, the rate of change in the requirements and the model, and the rate of change in the SUT. To finish this section, let us step back and take a more philosophical view of model-based testing. It is always the case that test design is based on some kind of model of expected behavior, but with manual test design, this model is usually just an informal mental model. By making the model explicit, in a notation that can be used by model-based testing tools, we are able to generate tests automatically (which decreases the cost of testing), generate an arbitrary number of tests, as well as obtain more systematic coverage of the model. These changes can increase both the quality and quantity of our test suite.

11.3 Modelling your system

[3]2 A trademark of Mercury Interactive Corporation; see http:// www.mercury.com.

[4]3 We should also mention the possibility of an error in the model-based testing tools themselves. Of course, this is unlikely, since they are surely well tested!

11.4 Partition and Combinatorial testing

11.5 Test generation from FSMs