1 Making tests executable

This chapter deals with the practical issues of how to take a suite of abstract tests, generated from an abstract model, and make them executable on the real SUT. This concretization phase is an important part of the model-based testing process, and it can take a significant amount of effort. In some applications of model-based testing, the amount of time spent on concretization can be the same as the time spent on modeling; in other applications concretization is less time-consuming (for example, between 25 and 45 percent of the modeling time in the case studies in Section 2.6.3). Section 8.1 discusses the challenges that can arise, gives an overview of the various approaches, and briefly discusses some examples. Section 8.2 gives an extended example of how these approaches work. It shows how the test suite generated for the eTheater case study in Section 7.2 can be trans- formed into executable test scripts, written in the Ruby scripting language, and executed on an implementation of the eTheater system.

1.1 Principles of test adaptation

Having used model-based testing to generate a nice test suite, we almost always want to automate the execution of that test suite. The Qui-Donc example in Chapter 5 was one exception to this, where we decided it was better to execute the generated tests manually because the test execution involved interacting with a telephone system with voice output. But in most cases, we want to automate the execution of the generated tests so that our whole test process is automated and so that we can execute more tests, execute tests more often for regression testing, reduce our test execution costs, reduce the overall testing time, and so on. The problem that we face in trying to execute tests generated from a model is that the generated tests are highly abstract, just like the model, so they usually do not contain enough concrete detail to be directly executable. In other words, the API of the model does not exactly match the API of the SUT. Recall from Section 3.1 some of the kinds of abstraction that are commonly used when designing a model for testing purposes:

Common abstractions

- Model only one aspect of the SUT, not all its behavior.
- Omit inputs and outputs that are not relevant to the test goals.
- Take a simplified view of complex data values, such as enumerating a few typical values.

- Assume that the SUT has already been initialized to match a particular testing scenario.
- Define a single model operation that corresponds to a sequence of SUT operations, or to just one part of an SUT operation.

One important dimension of testing is whether to do online or offline testing.

Online vs offline testing

Online testing is where tests are executed as they are generated, so the model-based testing tool is tightly coupled to the SUT.

Offline testing decouples the generation and execution phases, so the test execution can be completely inde- pendent of the model-based test generation process. This book contains several examples of each of these. Online testing is

To execute the generated tests, we must first initialize the SUT so that it is ready for our test suite, add the missing details into the tests, and fix any mismatches between the API of the model and the SUT so that our tests can connect to the SUT interface. We also have to manage the relationships among abstract values in the model and real-world values or objects in the SUT. This requires expanding the abstract model values into more complex concrete values that can be used as SUT inputs. For example, an enumerated constant in the model might have to be expanded into a complex data value such as a record that contains several fields. To be able to check SUT outputs against the model, we must either transform the expected outputs from the model into concrete values or get concrete outputs from the SUT and transform them back into abstract values so that we can check them against the model. If the model is deterministic, we can use either approach; but if it is nondeterministic, then the latter approach is better. If the SUT creates new objects during the testing process, it is often necessary to keep track of the identity of those objects, not just their values. This requires maintaining a mapping table from the abstract objects in the model to the corresponding concrete objects that have been created by the SUT. Each time the model creates a new abstract value A, the SUT performs the corresponding operation and creates a concrete object C; then we add the pair (A, C) into the mapping table so that future uses of A in the model can be translated into uses of C in the SUT. These techniques are all examples of the difference in abstraction levels between the model and the SUT. Figure 8.1 shows the main approaches to bridging this abstraction gap.

- The adaptation approach, (a), is to manually write some adapter code that bridges the gap. This is essentially a wrapper around the SUT that provides a more abstract view of the SUT to match the abstraction level of the model.
- The transformation approach, (b), is to transform the abstract tests into concrete test scripts.

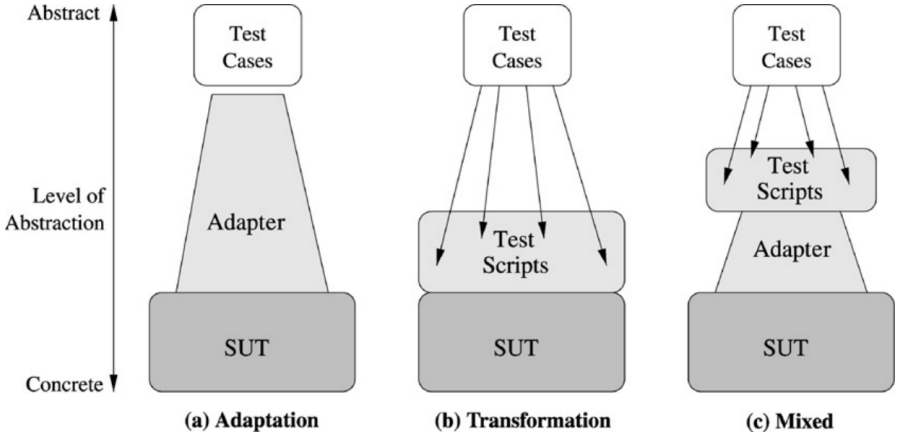

Figure 1.1: Figure 8.1 Three approaches to bridging the semantic gap between abstract tests and the concrete SUT.

- The mixed approach, (c), is a combination of the other two approaches. It is sometimes useful to add some adapter code around the SUT to raise its abstraction level part of the way toward the model and make testing easier, and then transform the abstract tests into a more concrete form that matches the adapter interface. Some benefits of this mixed approach are that the transformation can be easier, since the levels of abstraction are closer, and the adapter can be less model-specific, which may allow it to be reused for several different models or different testing scenarios.

We explore the adaptation and transformation alternatives more in the next two subsections and then discuss which approach is best for each kind of application. As an analogy, the transformation approach is like compiling the abstract tests down into some lower-level language, while the adaptation approach is like interpreting the abstract test sequences. The mixed approach is like a bytecode system, which compiles the input language down to byte- codes, and then interprets those bytecodes.

1.1.1 The Adaptation Approach

The adaptation approach involves writing some adapter code that wraps around the SUT, manages the low-level details of interacting with the SUT, and presents a more abstract view of the SUT to the model. Essentially, this adapter code acts as an interpreter for the abstract operation calls of the model, executing them with the help of the SUT. More specifically, the adapter code is responsible for the following tasks:

Adaptation

Setup: Set up the SUT so that it is ready for testing. This involves configuring and initializing the SUT so that it reflects the test scenario assumed by the model.

Concretization: Translate each model-level abstract operation call and its abstract input values into one or more concrete SUT calls with and its the appropriate input values.

Abstraction: Obtain the SUT results from those concrete calls, and translate them back into abstract values, and then pass them back to the model for comparison with the expected results in order to produce the test verdict.

Teardown: Shut down the SUT at the end of each test sequence or at the end of each batch of tests.

There are many possible architectures for connecting the model-based testing tool, the adapter, and the SUT. Most of the test harness architectures invented over the last decades for automating the execution of manually designed tests can be applied to model-based testing as well. Earlier in this book, we discussed two examples of the adaptation approach that nicely illustrate some typical solutions.
****

1.1.2 The Transformation Approach

The transformation approach involves transforming each abstract test into an executable test script. The resulting test scripts are typically in one of the following notations:

- A standard programming language, such as Java or C
- A scripting language, such as TCL, JavaScript, or VBScript
- A standard test notation, such as TTCN-3 [WDT+ 05]
- A proprietary test notation, such as the TSL (Test Script Language) of Mercury WinRunner or some company-specific test notation

The transformation process may be performed by a single-purpose translation program that always produces output in a particular language or by a more generic engine that is capable of transforming abstract tests into several languages because it is parameterized by various templates and mappings for each output language. For example, we might define a TCL template for each abstract operation, which contains the TCL code for calling that operation in the SUT, with placeholders for the parameter values. This would allow a generic transformation engine to build a concrete test script by transforming the calls in an abstract test sequence into a sequence of these TCL templates, with actual parameter values replacing the placeholders. In reality, the transformation process is likely to be more complex than this because the following factors must be taken into account.

- Some concrete setup and teardown code will be needed. This could be written manually (for example, in TCL) and then inserted automatically by the transformation engine at the beginning and end of each test sequence or test run.

- The template for each operation may be quite complex because there is not necessarily a one-to-one mapping between the signature of the abstract operation and the SUT operations. For example, it may have to call several SUT operations to implement the abstract operation; it typically has to generate any missing input parameters; it must trap exceptions produced by the SUT and check if they were expected or not; and it may have to call some SUT query operations to check that the internal state of the SUT agrees with the expected state from the model. Note that some of the oracle checking may be done within the template for each operation, and other parts of the oracle checking may be done by separate query operations in the abstract test sequence.

- The model uses abstract constants and values—these must be translated into the concrete values used by the SUT. This can often be done by defining a mapping from each abstract value to a corresponding concrete value or into a concrete expression that generates a suitable value. For example, we might map the enumeration type {BadPIN , GoodPIN } into the two concrete PINs 1111 and 2593 or map BadPIN into an expression like rand.Next(2000), which randomly generates a different incorrect PIN on each call.

- When testing nondeterministic SUTs, an abstract test may have a tree structure rather than being a simple sequence. This requires the transformation engine to be more sophisticated to handle such structures. It must generate an executable test script that includes conditional statements to check the SUT outputs and then take the appropriate branch through the tree to the next part of the test.

- Traceability between the concrete test scripts and the abstract tests must be maintained, and it is often desirable to directly record within each generated test script the traceability links back to the model and the informal requirements. This can be done by inserting these traceability links as comments within the test scripts or as executable statements so that the traceability information can be displayed whenever a test fails.

The structure of the transformed test suite is almost as important as the code within each test script. When you have hundreds or thousands of tests, their naming conventions and organization into hierarchies of folders become very important. Most companies that have a mature testing process have standards for the organization and version management of their test suites, and it is desirable for the executable test suites generated via model- based testing to follow the same standards.

For example, in addition to generating the executable code of the test scripts, it may be desirable to generate a test plan, which describes, in English, the structure of the test suite, the rationale for each test in the suite, the settings used to generate the tests, who generated the tests, when they were generated, and so on.

Another important issue is version management of the tests. Typically, each generated test suite is put into a version control system so that it can be recorded, managed, and used for regression testing. But with the trans- formation approach, we have the abstract

test suite as well as the concrete (executable) test scripts, so it is necessary to decide which artifacts to put under version control. Just the executable test scripts? Or the abstract test suite as well? Or should we instead store the model and the test selection criteria that were used to generate the tests? The answers depend on the culture and practice of your company, but deciding on the answers to these questions is an important part of adopting model-based testing into an existing testing process.

Key Point The transformation approach can produce test scripts that fit smoothly into your existing test management practices, with similar language, structure, and naming conventions as manually written test scripts.

We will see some examples of the transformation approach later in this chapter and other examples in Sections 9.5 and 10.4. Before looking at those examples, we give some guidelines for choosing between the transformation and adaptation approaches.

1.1.3 Which Approach Is Better?

For online testing, it is almost always better to use the adaptation approach because online testing requires a tightly integrated, two-way connection be- tween the model-based testing tool and the SUT. This is easiest to achieve when the model-based testing can directly connect to the adapter API and the adapter is in turn directly connected to the SUT interface.

For offline testing, we can choose between the adaptation approach or the transformation approach, or use a mixture of the two. If we use the transformation approach, then we obtain a suite of executable test scripts that can be executed directly on the SUT. If we use the adaptation approach, then our suite of abstract tests effectively becomes executable because the adapter acts as an interpreter, mapping each abstract call into SUT operations and translating the SUT results back to the abstract level. With the mixed approach, we transform the abstract tests into executable test scripts that call an adapter layer to handle the low-level details of SUT interaction.

The transformation approach has the advantage that it can produce test scripts in the same language and with the same naming and structuring conventions that are already used for manually written tests. This can make the adoption of model-based testing less disruptive on the overall testing process by allowing existing test management tools, test repositories, test execution platforms, and test reporting processes to remain unchanged. Essentially, model-based testing is replacing the test design and test scripting stages of the testing process by new techniques and tools, but the rest of the testing process remains the same. For these reasons, we suggest that you generally consider the adaptation approach first when you are doing online testing and consider the transformation approach first when doing offline testing. However, this is not a hard-and-fast rule—other combinations are possible, and it is quite common to use the adaptation approach for offline testing as well as for online testing. The best approach for your project depends on political, technical, and human factors as well as on the available tools and the nature of the project.

Key Point For online testing, use the adaptation approach. For offline testing, the transformation approach has some advantages (less disruption), and it is often useful to combine it with the adaptation approach.