Part I

Overview

1 Describe, Analyze, Test

Creating software is a notoriously error-prone activity. If the errors might have serious consequences, we must check the product in some systematic way. Every project uses *testing*: check the code by executing it. Some projects also *inspect* code, or use *static analysis* tools to check code without executing it. Finding the best balance among these *assurance methods*, and the best techniques and tools for each, is an active area of research and controversy. Each approach has its own strengths and weaknesses.[1]

The unique strength of testing arises because it actually executes the code in an environment similar to where it will be used, so it checks all of the assumptions that the developers made about the operating environment and the development tools.

But testing is always incomplete, so we have to use other assurance methods also. And there are other important development products besides code. To be sure that the code solves the right problem, we must have a *specification* that describes what we want the program to do. To be sure that the units of code will work together, we need a *design* that describes how the program is built up from parts and how the parts communicate. If the specification or design turns out to be wrong, code may have to be reworked or discarded, so many projects conduct *reviews* or *inspections* where people examine specifications and designs. These are usually expressed in *informal* notations such as natural language and hand-drawn diagrams that cannot be analyzed automatically, so reviews and inspections are time-consuming, subjective, and fallible.

In this book we teach novel solutions to these problems: expressing and checking specifications and designs, generating test cases, and checking the results of test runs. The methods we describe increase the automation in each of these activities, so they can be more timely, more thorough, and (we expect) more effective.

[1] Definitions for terms that are printed in italics where they first appear are collected in the Glossary (Appendix C).

We also teach a technology that realizes these solutions: the NModel modeling and testing framework, a library and several tools (applications) built on the C# language and .NET. However, this technology is not just for .NET applications. We can use it to analyze and test programs that run outside .NET, on any computer, under any operating system. Moreover, the concepts and methods are independent of this particular technology, so this book should be useful even if you use different languages and tools.

In the following sections we briefly describe what the technology can do. Explanations of how it works come later in the book.

1.1 Model programs

We express what we want the program to do – the *specification* – by writing another much simpler program that we call a *model program*. We can also write a model program to describe a program *unit* or *component* – in that case, it expresses part of the *design*. The program, component, or system that the model program describes is called the *implementation*. A single model program can represent a *distributed system* with many computers, a *concurrent system* where many programs run at the same time, or a *reactive program* that responds to events in its environment.

A model program can act as executable documentation. Unlike typical documentation, it can be executed and analyzed automatically. It can serve as a prototype. With an analysis tool, it can check whether the specification and design actually produce the intended behaviors. With a testing tool, it can generate test cases, and can act as the *oracle* that checks whether the implemenation passes the tests.

In the NModel framework, model programs are written in C#, augmented by a library of attributes and data types. Methods in the model program represent the *actions* (units of behavior) of the implementation. Variables in the model program represent the *state* (stored information) of the implementation. Each distinct combination of values for the variables in the model program represents a particular state (situation or condition) of the implementation.

Within a model program, we can identify separate *features* (groups of related variables and methods). We can then perform analysis or testing limited to particular features or combinations of features.

We can write separate model programs and then combine them using *composition*. Composition is a program-transformation technique that is performed automatically by our analysis and testing tools, which can then analyze or test from the composed program. Composition is defined (and implemented by the tools) in a way that makes it convenient to specify interacting features, or to limit analysis and testing to particular scenarios, or to describe temporal properties to check during analysis.

To see how to write a model program, we can refer to traditional, informal specifications and design documents. Sometimes there is already an implementation we can inspect or experiment with. Sometimes a designer will write the model program first, going directly from ideas to code. There is no algorithm or automated method for deriving a model program – we have to use judgment and intuition. But there are systematic methods for *validating* a model program – checking that it behaves as we intended.

Writing a model program does not mean writing the implementation twice. A model program should be much smaller and simpler than the implementation. To achieve this, we usually select just a subset of the implementation's features to model. A large implementation can be covered by several small model programs that represent different subsets of features. Within each subset, we choose a *level of abstraction* where we identify the essential elements in the implementation that must also appear in the model. Other implementation details can be omitted or greatly simplified in the model. We can ignore efficiency, writing the simplest model program that produces the required behaviors, without regard for performance. Thanks to all this, the model program is much shorter and easier to write than the implementation, and we can analyze it more thoroughly. "The size of the specification and the effort required in its construction is not proportional to the size of the object being specified. Useful and significant results about large program can be obtained by analyzing a much smaller artifact: a specification that models an aspect of its behavior."[2]

We use the term *preliminary analysis* for this preparatory activity where we select the subset of features to include, identify the state and the actions that we will represent, and choose the level of abstraction.

Writing a model program can be a useful activity in its own right. When we (the authors) write a model program, we usually find that the source materials provided to us – the informal specifications and design documents – are ambiguous and incomplete. We can always come up with a list of questions for the architects and designers. In the course of resolving these, the source materials are revised. Clarifications are made; future misunderstandings with developers and customers are avoided. Potential problems and outright errors are often exposed and corrected.

1.2 Model-based analysis

Model-based analysis uses a model program to debug and improve specifications and designs, including architectural descriptions and protocols. Model-based analysis can also help to *validate* the model programs themselves: to show that they actually

[2] The quotation is from Jackson and Damon (1996).

do behave as intended. The model program is expressed in a *formal* notation (a programming language), so it can be analyzed automatically. Analysis uses the same model programs and much of the same technology as testing.

Runs of the model program are *simulations* (or *animations*) that can expose problems by revealing unintended or unexpected behaviors. To perform a simulation, simply code a main method or a unit test that calls the methods in the model program in the order that expresses the scenario you wish to see. Then execute it and observe the results.

We can analyze the model program more thoroughly by a technique called *exploration*, which achieves the effect of many simulation runs. It is our primary technique for analyzing model programs. Exploration automatically executes the methods of the model program, selecting methods in a systematic way to maximize coverage of the model program's behavior, executing as many different method calls (with different parameters) reaching as many different states as possible. Exploration records each method call it invokes and each state it visits, building a data structure of states linked by method calls that represents a *finite state machine* (FSM).[3]

The `mpv` (Model Program Viewer) tool performs exploration and displays the results as a *state-transition diagram*, where the states appear as bubbles, the *transitions* between them (the method calls) appear as arrows, and interesting states and transitions are highlighted (see, e.g., Chapter 3, Figures 3.8–3.11).

The input to `mpv` is one or more model programs to explore. If there is more than one, `mpv` forms their composition and explores the composed program. Composition can be used to limit exploration to particular scenarios of interest, or to formulate temporal properties to analyze.

It can be helpful to view the result of exploration even when you do not have a precise question formulated, because it might reveal that the model program does not behave as you intend. For example, you may see many more or many fewer states and transitions than you expected, or you may see dead ends or cycles you did not expect.

Exploration can also answer precisely formulated questions. It can perform a *safety analysis* that identifies *unsafe* (forbidden) states, or a *liveness analysis* that identifies *dead states* from which goals cannot be reached. To prepare for safety analysis, you must write a Boolean expression that is true only in the unsafe states. Exploration will search for these unsafe states. To prepare for liveness analysis, you must write a Boolean expression that is true only in *accepting states* where the program is allowed to stop (i.e., where the program's goals have been achieved). Exploration will search for dead states, from which the accepting states cannot be reached. Dead states indicate *deadlocks* (where the program seems to stop running

[3] Exploration is similar to another analysis technique called *model checking*.

and stops responding to events) or *livelocks* (where the program keeps running but can't make progress). The `mpv` tool can highlight unsafe states or dead states.

There is an important distinction between finite model programs where every state and transition can be explored, and the more usual *"infinite"* model programs that define too many states and transitions to explore them all. Recall that a state is a particular assignment of values to the program variables. Finite programs usually have a small number of variables with finite domains: Booleans, enumerations, or small integers. The variables of "infinite" model programs have "infinite" domains: numbers, strings, or richer data types.

To explore "infinite" model programs, we must resort to *finitization*: execute a finite subset of method calls (including parameters) that we judge to be representative for the purposes of a particular analysis. Exploration with finitization generates an FSM that is an *approximation* of the huge *true FSM* that represents all possible behaviors of the model program. Although an approximation is not complete, it can be far more thorough than is usually achieved without this level of automation. Along with abstraction and choosing feature subsets, approximation makes it feasible to analyze large, complex systems.

We provide many different techniques for achieving *finitization* by *pruning* or *sampling*, where the analyst can define rules for limiting exploration. Much of the analyst's skill involves choosing a finitization technique that achieves meaningful coverage or probes particular issues.

1.3 Model-based testing

Model-based testing is testing based on a model that describes how the program is supposed to behave. The model is used to automatically generate the test cases, and can also be used as the *oracle* that checks whether the *implementation under test* (IUT) passes the tests.

We distinguish between *offline* or *a priori testing*, where the test case is generated before it is executed, and *online* or *on-the-fly testing*, where the test case is generated as the test executes. A test case is a *run*, a sample of behavior consisting of a sequence of method calls. In both techniques, test cases are generated by exploring a model program. In offline testing using the `otg` tool (Offline Test Generator), exploration generates an FSM, the FSM is *traversed* to generate a scenario, the scenario is saved in a file, and later the `ct` tool (Conformance Tester) executes the test by running the scenario. In online testing, `ct` creates the scenario on-the-fly during the test run. The `ct` tool executes the model program and the IUT in *lockstep*; the IUT executes its methods as exploration executes the corresponding methods in the model program, and the model program acts as the oracle to check the IUT.

To use ct, you must provide one or more model programs and write a *test harness* that couples your IUT to the tool. If you provide more than one model program, ct composes them and explores their composition. In this context, composition is usually used to limit exploration to particular scenarios. If you wish, you can write a custom *strategy* in C# that ct uses to maximize test coverage according to criteria you define.

We distinguish between *controllable actions* of the IUT that can be executed on demand by the test tool and *observable actions* of the IUT that the test tool can only monitor. Method calls are controllable actions, while events (including message arrival and user's input such as keystrokes and mouse clicks) are observable actions. Observable actions are usually *nondeterministic*: it is not possible for the tester to predict which of several possible observable actions will happen next. We can classify systems by their controllability and determinism. *Closed systems* are fully controllable and deterministic. *Reactive systems* have both controllable and observable actions. Some systems are uncontrollable, with only observable actions; a log file is a simple example of such a system.

Some test tools can only handle closed systems. Such tools can be used to test reactive systems by creating a *sandbox* where normally observable actions are made controllable by the test harness (which can be made to generate messages or events on demand). But a sandbox is not realistic and is not always technically feasible. The ct tool can accommodate observable events, which means, for example, that it can test an IUT at one end of a network connection. On-the-fly testing works well for reactive systems because it can deal efficiently with nondeterminism.

1.4 Model programs in the software process

Model programs can change the way we develop software. We can begin checking and testing development products earlier in the project.

To see how this works, it is helpful to represent software project activities and schedule in a V-diagram (Figure 1.1). The horizontal axis represents time, beginning with the project concept at the left and ending with the product delivery on the right. The vertical axis represents the *level of integration*, with the entire product at the top, and the smallest meaningful units of software (classes and other types in C#) at the bottom. A traditional project begins at the upper left with a product concept, then works down the left side of the V, creating a specification that describes what the product should do, and a design that describes how the product should be built up from units. At the bottom of the V, developers code and test individual units. Then the project works up the right side, integrating and testing larger collections of units, until the complete product is delivered. (In projects with frequent releases, there can be a V-diagram for each release.)

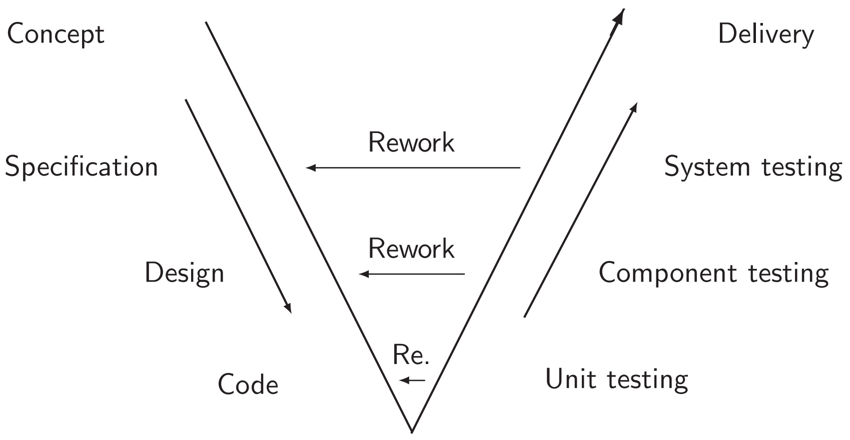

Figure 1.1. V-diagram showing traditional software project activities and schedule.

The V-diagram shows how each kind of testing activity (on the right side) is supposed to check one kind of development product (at the same level on the left side). This reveals the problem with traditional sequential development: the products that are produced first (the specification and high-level system design) are tested last (by tests on the integrated system, using scenarios suggested by customers or their representatives among the developers). Therefore, defects in these first products might not be discovered until much later, after code has been written from them. The diagram shows how the costs of rework escalate as defects are discovered later in the project.

It would be better if we could check each product as soon as it is produced. We would like to check and correct the specification and design as we work down the left side of the V, before we code the units. Then, soon after the unit tests pass, the integrated system should just work – with few unpleasant surprises.

Something like this is already being tried at the unit level (down at the point of the V). In *test-driven development*, developers write each unit test case before its code, and execute it as they code. When the code for each unit is completed, it has already passed its unit tests.

Now it is possible to apply the same principle of immediate feedback to other development products. Analysis with model programs can check specifications and designs, much like unit tests check code, so problems can be detected and fixed immediately (Figure 1.2).

Analyzing and testing models is common in many branches of engineering, where builders cannot depend on testing the end product in order to expose major defects. No one plans to crash an airplane or collapse a bridge! Instead, engineers create mathematical models – such as block diagrams for circuits and control systems, or

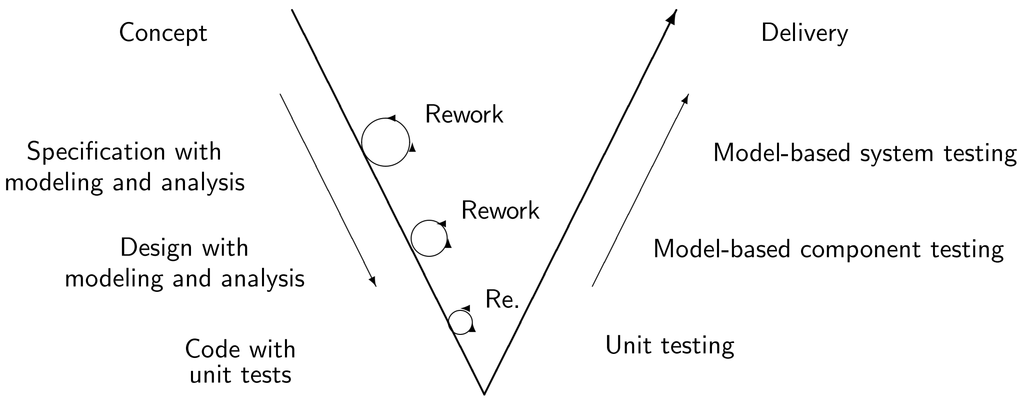
Concept
Delivery
Rework
Specification with modeling and analysis
Model-based system testing
Rework
Design with modeling and analysis
Model-based component testing
Re.
Code with unit tests
Unit testing

Figure 1.2. V-diagram showing opportunities for model-based testing and analysis.

finite element models for structures – and even build physical models (to test in wind tunnels, etc.). Our model programs are analogous to the models used in other branches of engineering.

Model-based analysis and testing are each useful in their own right. A project might use either or both. Figure 1.2 shows several opportunities for using them, but a particular project might take only one or two. We know of projects where architects used model programs just to debug protocols early in the project, and testing was performed in the usual way. We know of others where testers wrote model programs just for testing components that had been specified and designed in the usual way. Moreover, we can model specifications (by modeling the system behavior visible to users) or designs (by modeling the behavior of components only visible to developers) or both (as shown in Figure 1.2). It is not necessary to model everything; projects typically focus their modeling efforts on the system behaviors or components that are the most novel, critical, or intricate.

Although model programs can be helpful during specification and design, this book emphasizes model-based testing. Many of our examples assume that an implementation is already in hand or on the way, so the model is based on the implementation (not the other way around). We usually model subsets of features that are about the right size for a tester's work assignment (a typical test suite). We usually choose a level of abstraction where actions in the model correspond to actions in the implementation that are easy to observe and instrument (such as calls to methods in its API).

Moreover, we believe that the tester's point of view is especially helpful in keeping the models grounded in reality. In order to perform conformance testing, which involves lockstep execution of the model with the implementation, the tester must write a *test harness* that couples the implementation to the model through the test tool. This test harness makes the correspondence between the model and the implementation completely explicit. Writing this harness, and the subsequent lockstep execution itself, closes the loop from the model back to the implementation. It validates the model with a degree of thoroughness that is not easily achieved in projects that do not use the models for testing.

1.5 Syllabus

This book teaches how to write and analyze model programs, and how to use them to test implementations. Here is a brief summary of the topics to come.

The book is divided into four parts. The end of each part is an exit point; a reader who stops there will have coherent understanding and tools for modeling, analysis, and testing up to that level of complexity. Presentation is sequential through Part III, each chapter and part is a prerequisite for all following chapters and parts. Chapters in Part IV are independent; readers can read one, some, or all in any order. Each part concludes with a guide to futher reading, an annotated bibliography of pertinent literature including research papers.

Part I shows what model-based testing and analysis can do; the rest of the book shows how to do it.

Chapter 1 (this chapter) is a preview of the topics in the rest of the book.

Chapter 2 demonstrates why we need model-based testing. We exhibit a software defect that is not detected by typical unit tests, but is only exposed by executing more realistic scenarios that resemble actual application program runs. We preview our testing techniques and show how they can detect the defect that the unit tests missed.

Chapter 3 demonstrates why we need model-based analysis. We exhibit a program with design errors that cause safety violations (where the program reaches forbidden states), deadlocks (where the program seems to stop running and stops responding to events), and livelocks (where the program keeps running but can't make progress). We preview our analysis and visualization techniques and show how they can reveal the design errors, before beginning any testing.

Part II explains modeling, analysis, and testing with finite models that can be analyzed exhaustively. (The systems that are modeled need not be finite.)

Chapter 5 introduces the modeling library and explains how to write model programs.

Chapter 6 introduces our primary model-based analysis technique, exploration. We introduce the analysis and visualization tool, `mpv` (Model Program Viewer), and explain how it can reveal errors like those discussed in Chapter 3.

Chapter 7 introduces features and model composition, which are used to build up complex model programs by combining simpler ones, to focus exploration and