Chapter 2

Test Generation from Requirements

The purpose of this chapter is to introduce techniques for the generation of test data from informally and formally specified requirements. Some of these techniques can be automated while others may require significant manual effort for large applications. Most techniques presented here fall under the category "black-box" testing in that tests are generated without access to the code to be tested.

2.1. Introduction

Requirements serve as the starting point for the generation of tests. During the initial phases of development, requirements may exist only in the minds of one or more people. These requirements, more aptly ideas, are then specified rigorously using modeling elements such as use cases, sequence diagrams, and statecharts in UML. Rigorously specified requirements are often transformed into formal requirements using requirements specification languages such as Z, S, and RSML.

While a complete formal specification is a useful document, it is often the case that aspects of requirements are captured using appropriate modeling formalisms. For example, Petri nets and its variants are used for specifying timing and concurrency properties in a distributed system, timed input automata to specify timing constraints in a real-time system, and finite state machines to capture state transitions in a protocol. UML is an interesting formalism in that it combines into a single framework several different notations used for rigorously and formally specifying requirements.

A requirement specification can thus be informal, rigorous, formal, or a mix of these three approaches. Usually, requirements of commercial applications are specified using a mix of the three approaches. In either case, it is the task of the tester, or a test team, to generate tests for the entire application regardless of the formalism in which the specifications are available. The more formal the specification, the higher are the chances of automating the test generation process. For example, specifications using finite state machines, timed automata, and Petri nets can be input to an programmed test generator and tests generated automatically. However, a significant manual effort is required when generating test cases from use cases.

Often, high level designs are also considered as part of requirement specification. For example, high level sequence and activity diagrams in UML are used for specifying interaction amongst high level objects. Tests can also be generated from such high level designs.

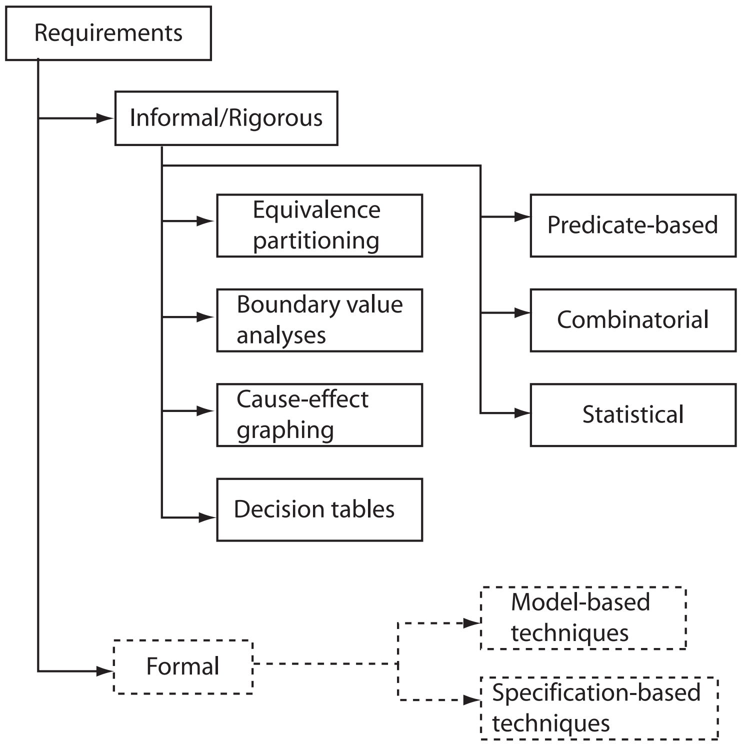

Figure 2.1: Techniques listed here for test selection from informal and rigorously specified requirements, are introduced in this chapter. Techniques from formally specified requirements using graphical models and logic based languages, are discussed in other chapters.

In this chapter we are concerned with the generation of tests from informal and rigorously specified requirements. These requirements serve as a source for the identification of the input domain of the application to be developed. A variety of test generation techniques are available to select a subset of the input domain to serve as test set against which the application will be tested.

Figure 2.1 lists the techniques described in this chapter. The figure shows requirements specification in three forms: informal, rigorous, and formal. The input domain is derived from the informal and rigorous specifications. The input domain then serves as a source for test selection. Various techniques, listed in the figure, are used to select a relatively small number of test cases from a usually very large input domain.

The remainder of this chapter describes each of the techniques listed in the figure. All techniques described here fall under the “black-box” testing category. However, some could be enhanced if the program code is available. Such enhancements are discussed in Part 3 of this book.

2.2. The test selection problem

Let D denote the input domain of program p. The test selection problem is to select a subset T of tests such that execution of p against each element of T will reveal all errors in p. In general there does not exist any algorithm to construct such a test set. However, there are heuristics and model based methods that can be used to generate tests that will reveal certain type of faults. The challenge is to construct a test set $T \subseteq D$ that will reveal as many errors in p as possible. As discussed next, the problem of test selection is difficult due primarily to the size and complexity of the input domain of p.

An input domain for a program is the set of all possible *legal* inputs that the program may receive during execution. The set of legal inputs is derived from the requirements. In most practical problems, the input domain is large, i.e. has many elements, and often complex, that is the elements are of different types such as integers, strings, reals, Boolean, and structures.

The large size of the input domain prevents a tester from exhaustively testing the program under test against all possible inputs. By “exhaustive” testing we mean testing the given program against every element in its input domain. The complexity makes it harder to select individual tests. The following two examples illustrate what is responsible for large and complex input domains.

EXAMPLE 2.1. Consider program P that is required to sort a sequence of integers into ascending order. Assuming that P will be executed on a machine in which integers range from -32768 to 32767, the input domain of p consists of all possible sequences of integers in the range $[-32768, 32767]$.

If there is no limit on the size of the sequence that can be input, then the input domain of P is infinitely large and P can never be tested exhaustively. If the size of the input sequence is limited to, say $N_{max} > 1$, then the size of the input domain depends on the value of N. In general, denoting the size of the input domain by S, we get

$$S = \Sigma_{i=0}^{N} v^i$$

where v is the number of possible values each element of the input sequence may assume, which is 65536. Using the formula given above it is easy to verify that the size of the input domain for this simple sort program is enormously large even for small values of N. Even for small values of N, say $N = 3$, the size of the input space is large enough to prevent exhaustive testing. ■

EXAMPLE 2.2. Consider a procedure P in a payroll processing system that takes an employee record as input and computes the weekly salary. For simplicity, assume that the employee record consists of the following items with their respective types and constraints: