Book Outline

Software Testing and Analysis: Process, Principles, and Techniques

Mauro Pezzè and Michal Young

Working Outline as of March 2000

Software test and analysis are essential techniques for producing dependable software. While one cannot "test quality into" a badly constructed software product, neither can one build quality into a product without test and analysis. Achieving adequate quality through testing is often a major cost in software development, as well as a major factor in product cycle time. Failing to obtain adequate quality can be an even greater cost. This book addresses software test and analysis in the context of an overall effort to achieve an acceptable level of quality at acceptable cost. It assumes that the reader's aim is not ultra-high dependability without regard to cost, nor cutting cost without regard to quality, but rather maximizing cost-effectiveness while balancing cost, schedule, and quality. It assumes, moreover, that one is not artificially constrained to isolated improvements only in a well-defined testing group or testing phase, but is instead motivated to integrate testing and analysis activities as effectively as possible across the entire process of software development and evolution.

Software testing and software analysis techniques are treated here together, which is unusual in the current book-length literature but is consistent with a goal of achieving quality as cost-effectively as possible, using whatever combination of techniques is most appropriate. Moreover, software testing and software analysis depend on much of the same fundamental technical background. One must understand both to choose and combine techniques, and combinations of testing and analysis techniques can often be made to interact synergistically. The traditionally strong distinction between software testing and formal techniques for software analysis is an artifact of historical development of the fields, and is no longer useful.

Part 1 of this book presents fundamental principles of software test and analysis techniques in a coherent framework, laying the groundwork for understanding strengths and weaknesses of individual techniques and their application in an effective software process. Part 2 brings together basic technical background of many testing and analysis methods and techniques. Part 3 presents families of solutions to common test and analysis problems. Part 4 discusses how to design a systematic testing and analysis process and graft it into an overall development process. Part 5 presents specializations of testing and analysis techniques for some important application domains with unique requirements.

Part I
Fundamentals of Testing and Analysis

1 Software Test and Analysis in a Nutshell

Software analysis and test is multi-faceted. This chapter walks through a sample test and analysis process to illustrate how a combination of methods, tools, and techniques can be applied in the context of a disciplined approach to software development to produce quality products at acceptable cost.

2 A Framework for Test & Analysis

The purpose of software test and analysis is either to assess software qualities or else to make it possible to improve the software by finding defects. Of the many kinds of software qualities, those addressed by the analysis and test techniques discussed in this book are the *dependability* properties of the software *product*.

There are no perfect test or analysis techniques, nor a single "best" technique for all circumstances. Rather, techniques exist in a complex space of tradeoffs, and often have complementary strengths and weaknesses. This chapter describes the nature of those tradeoffs and some of their consequences, and thereby a conceptual framework for understanding and better integrating material from later chapters on individual techniques.

3 Basic principles

This chapter presents some basic, recurring principles that are widely applicable to testing and analysis as well as design for testability.

4 Test and Analysis Activities Within a Software Process

Quality cannot be added after software development, but results from the interaction of many techniques to be applied during the whole life cycle. Identifying which technique must be applied when, and understanding interactions with the development process is crucial to tradeoff costs and quality issues.

This chapter discusses the complex intertwining between development and test and analysis activities and illustrates tradeoffs for integrating the different activities within a coherent process.

5 Overview of Testing and Analysis Techniques

This chapter overviews the main test and analysis problems and indicates how they can tackle with different techniques.

Part II
Basic Techniques

6 Finite Models

Many test and analysis techniques are based on a finite model of the software. This Chapter present different flow graph representations of the software control structure.

7 Data Flow Analysis

Data flow analysis is a well known technique that finds many applications in computer science, e.g., in compiler design, program optimization, slicing. Data flow analysis provides interesting information about the structure of the code that can be used for deducing static properties of the code and for deriving coverage information. This chapter introduces the technique.

8 Symbolic Execution

Symbolic execution builds a set of predicates that characterize a set of executions corresponding to a specific path in the program. Predicates produced with symbolic execution describes the conditions under which the path can be executed and the effect of the execution on the outputs.

9 Formal proof of properties

This chapter illustrates mathematical techniques to reason about the correctness of the code. Such techniques reduce the code to a set of theorem whose demonstration imply the correctness of the code.

10 Model checking

In contrast to formal proof of properties that tries to reason on an infinite space, model checking is tries proof the validity of properties of concurrent systems reasoning on a finite, although usually very large representation of the execution space. This chapter illustrates the methods to build a finite representation of the execution space and to automatically reason on top of it.

Part III
Problems and Methods

11 Test Case Selection and Adequacy

One of the first problems in software testing is the selection of test cases and their evaluation. This chapter introduces the different approaches to test case selection and the corresponding adequacy, that will be illustrated through this part. The chapter serves as a general introduction of the problem and proposes a conceptual frame for the various functional and structural approaches described in subsequent chapters.

12 System and Acceptance testing

System and acceptance testing consider the system as a whole and not as composed of different parts. This chapter identifies the problems that arise in this context and describes the possible approaches.

13 Functional Testing

Functional specifications of software are an important source of information for deriving test cases. This chapter discusses how specifications influence testing, identifies the main steps that characterize functional testing, and illustrates them by presenting the main approaches to functional testing.

14 Structural Testing

The structure of the software itself is a valuable source of information for selecting test cases and determining whether a set of test cases has been sufficiently thorough. We can ask whether a test suite has "covered" a control flow graph or other model of the program.[1] It is simplest to consider structural coverage criteria as addressing the test adequacy question: "Have we tested enough." In practice we will be interested not so much in asking whether we are done, but in asking what unmet obligations with respect to the adequacy criteria suggest about additional test cases that may be needed, i.e., we will often treat the adequacy criterion as a heuristic for test case selection or generation. For example, if one statement remains unexecuted despite execution of all the test cases in a test suite, we may devise additional test cases that exercise that statement. Structural information should not be used as the primary answer to the question, "How shall I choose tests," but it is useful in combination with other test selection criteria (particularly specification-based testing) to help answer the question "What additional test cases are needed to reveal faults that may not become apparent through black-box testing alone."

15 Data Flow Testing

Data flow information about code allow a fine grain selection of test cases. This chapter discusses how to use data flow analysis for steering test selection and compares the resulting criteria with criteria based on the control flow structure of the code.

16 Testing Complex Data Structures

Much of the complexity of a program may be in its data structures, particulary when the logic of a complex procedure is encoded in tables rather than program code. Testing of complex data structures includes testing data abstractions and systematically exercising logic that is encoded in constant data.

17 Testing Object-Oriented Software

The object oriented paradigm introduces new constructs that cannot be always address with testing techniques defined for imperative code. This chapter discusses the validity of traditional testing techniques for object oriented systems, identifies the new problems and illustrates the approaches that can help in solving these problems.

[1] In this chapter we use the term "program" generically for the artifact under test, whether that artifact is a complete application or an individual unit together with a test harness. This is consistent with usage in the testing research literature.

18 Fault-Based Testing

Fault taxonomies represent an additional source of information for test case selection. this chapter illustrates fault based techniques with particular attention to mutation analysis and indicates advantages, limitations and possible application domains for these techniques.

19 Integration Testing

Practitioners are well aware of the problems of integrating smaller components to build a larger system. No mater how well the components have been designed and tested, integration can result in new unforeseen problems. This chapter explain why new classes of errors can raise during integration and describes the related testing strategies. This chapter generalize the problem of integration testing by discussing the increasingly important problem of testing COTS, libraries and frameworks, the problem of testing software architectures and the problem of testing systems built with reusable components. Finally this chapter discusses the relation between design and test illustrating the important concept of design for testability.

20 Test Case Generation

Most test selection and adequacy criteria indicate which test have to be selected in terms of elements of code or specifications to be exercised, but do not provide the actual data that constitute the test cases. This chapter describes techniques for test case generation and discusses the possibility of automating this important step.

21 Run-Time Support for Testing

Unit and integration testing require the execution of parts of the final code that cannot be executed independently. This chapter describes different tradeoffs and techniques for generating the run time support required for executing unit and integration test, also known as scaffolding.

22 Oracles

Each test execution must be characterized as a success (the program executed as expected specified) or failure. This may be left to the judgment of a human if there are only a handful of test cases and the expected behavior is very simple and observable, but for larger numbers of test cases and

for complex behaviors and specifications, the "eyeball oracle" is neither efficient nor dependable. Automated test oracles can be part of test harnesses, or programs can be made self-checking through annotation with run-time assertions. Often a combination of oracle techniques is most attractive, combining a small number of special-case oracles with a more general procedure that can quickly but imperfectly classify the results of a large number of test outcomes.

23 Capture & Replay

Ideally one would have a test driver that systematically drives software through a set of test cases, and an oracle that checks correctness, all without human intervention. When that level of automation is not feasible, a capture/replay system may be a good compromise. With a capture/replay system, a test case is first exercised and judged manually (e.g., by interacting with a graphical user interface), but thereafter it can be replayed. Problems addressed in this chapter include what must be captured, at what level of abstraction, and how it can be systematically varied during replay.

24 Regression Testing

Software products are often developed iteratively, and they typically continue to evolve through many versions after the initial product release. Each change requires re-testing (regression testing), which can easily become the dominant cost of software evolution. The challenge for regression testing, then, is to limit cost while obtaining adequate assurance against unintended effects. Approaches to controlling the cost of regression testing have two main aspects, limiting the number of tests that must be re-executed by determining and controlling the possible effects of each change, and reducing the cost of re-running tests. Test documentation and test automation are crucial to the second aspect.

25 Inspection

Software inspections are manual, collaborative review activities. Inspections can be applied to any software artifact from requirements documents to source code to test plans. This flexibility makes inspections particularly valuable when other, more automated analyses are not applicable.

26 Code Analysis

A number of automated analyses can be applied to program source code. None of them are capable of showing that the code is functionally correct, but they may cost-effectively locate some common defects as well as producing auxiliary information that is useful in inspections and testing. Usually, an automated source code analysis technique for detecting faults is applicable or cost-effective in detecting faults only if it is applied together with programming standards, often including some annotation or additional description of intent. Some techniques require the support of sophisticated program analysis tools, but others are extremely simple.

27 Testing and Analysis of Specifications

Many product defects are ultimately the effects of faults in specifications, rather than coding errors. It is more efficient to find and remove these defects early rather than waiting for them to be revealed in product testing. While specifications cannot be "tested" in quite the same way as code, a number of techniques can be applied, ranging from inspections to construction and validation of scenarios to automated consistency checks.

28 Tools

Many software test and analysis techniques require the repetition of long and tedious activities, e.g., execution of thousands of test cases during regression testing. Other techniques are based on activities hardly completed manually, e.g., computing testing coverage requires the identification of which parts of the program are actually executed during testing. Some test and analysis techniques can be applied manually and it is possible to improve the test and analyis process without tools. However, tools are essential to take advantage of many methods and techniques and achieve the degree of efficiency often required in industrial settings. Many test and analysis tools are appearing on the market, witnessing the awareness of their importance, but also the maturity of the technology and the market.

This chapter aims at identifying which methods can or should be automatized, and what is the maturity of the required technology. The chapter does not present an overview of commercially available tools, but summarizes testing and alnaysis activites that are or can be automated and illustrates the added value of automation.

Part IV
Process

29 Fitting Test and Analysis to a Software Process

One does not have the luxury of devising an ideal software process for software test and analysis. Approaches to test and analysis interact with other aspects of software processes, and are to some extent constrained by them. In this chapter we discuss how effective processes for software test and analysis can be "fit" to overall software processes with a minimum of disruption and risk.

30 Improving the Process

There is no a priori "best" quality process that can be selected once and for all. Rather, quality is best approached through a feedback-driven process in which cost and quality issues are continuously monitored and tuned. This chapter presents process improvement specifically with respect to testing and analysis, and considering cost and schedule issues as well as the quality of the delivered product.

31 Fault Analysis

A cost-effective process for test and analysis depends on information about the cost and quality impacts of particular classes of faults. Fault occurrence frequency and impact differs from organization to organization, and changes over time as the organization and its products evolve. This chapter describes an approach to classifying and prioritizing faults and selecting appropriate counter-measures using root cause analysis.

Part V
Domain Specific Approaches to Test & Analysis

32 Testing Embedded Systems

Software is embedded in many devices, ranging from industrial equipment to telephones to automobiles to toys. Current trends suggest that an increasing

portion of software development will be targeted to embedded systems. This chapter addresses problems posed by testing embedded systems, including observability and controllability of the target system; consistency between host development system, instrumented test-bed, and the ultimate target environment; and resource constraints in the target environment.

33 Testing Concurrent & Distributed Systems

An underlying assumption of most testing is that a program is deterministic: If presented with the same input, it will always perform the same computations and produce the same output. Behaviors of concurrent (multi-threaded) and distributed systems are influenced by process scheduling, communication details, and other factors that are difficult to fully capture or control, with the result that they appear to be non-deterministic. We review special-purpose techniques that have been developed for deriving test oracles and test coverage criteria from formal models of concurrent and distributed systems.

34 Testing Graphical User Interfaces

Graphical user interfaces (GUIs) present several obstacles to cost-effective testing. This chapter describes approaches to factoring design and specification of GUIs for more cost-effective testing, development of test scaffolding to improve GUI test automation, and application of capture/replay specifically to GUI testing.