SOFTWARE ERROR DETECTION THROUGH TESTING AND ANALYSIS

J. C. Huang
University of Houston

WILEY
A JOHN WILEY & SONS, INC., PUBLICATION

SOFTWARE ERROR DETECTION THROUGH TESTING AND ANALYSIS

SOFTWARE ERROR DETECTION THROUGH TESTING AND ANALYSIS

J. C. Huang
University of Houston

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Library of Congress Cataloging-in-Publication Data:

Huang, J. C., 1935–
Software error detection through testing and analysis / J. C. Huang.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-40444-7 (cloth)
1. Computer software–Testing. 2. Computer software–Reliability. 3. Debugging in computer science. I. Title.
QA76.76.T48H72 2009
005.1′4–dc22

2008045493

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my parents

CONTENTS

Preface **ix**

1 Concepts, Notation, and Principles **1**

1.1 Concepts, Terminology, and Notation 4
1.2 Two Principles of Test-Case Selection 8
1.3 Classification of Faults 10
1.4 Classification of Test-Case Selection Methods 11
1.5 The Cost of Program Testing 12

2 Code-Based Test-Case Selection Methods **14**

2.1 Path Testing 16
2.2 Statement Testing 17
2.3 Branch Testing 21
2.4 Howden's and McCabe's Methods 23
2.5 Data-Flow Testing 26
2.6 Domain-Strategy Testing 36
2.7 Program Mutation and Fault Seeding 39
2.8 Discussion 46
Exercises 51

3 Specification-Based Test-Case Selection Methods **53**

3.1 Subfunction Testing 55
3.2 Predicate Testing 68
3.3 Boundary-Value Analysis 70
3.4 Error Guessing 71
3.5 Discussion 72
Exercises 73

4 Software Testing Roundup **76**

4.1 Ideal Test Sets 77
4.2 Operational Testing 80
4.3 Integration Testing 82
4.4 Testing Object-Oriented Programs 84
4.5 Regression Testing 88

4.6 Criteria for Stopping a Test 88
4.7 Choosing a Test-Case Selection Criterion 90
Exercises 93

5 Analysis of Symbolic Traces 94

5.1 Symbolic Trace and Program Graph 94
5.2 The Concept of a State Constraint 96
5.3 Rules for Moving and Simplifying Constraints 99
5.4 Rules for Moving and Simplifying Statements 110
5.5 Discussion 114
5.6 Supporting Software Tool 126
Exercises 131

6 Static Analysis 132

6.1 Data-Flow Anomaly Detection 134
6.2 Symbolic Evaluation (Execution) 137
6.3 Program Slicing 141
6.4 Code Inspection 146
6.5 Proving Programs Correct 152
Exercises 161

7 Program Instrumentation 163

7.1 Test-Coverage Measurement 164
7.2 Test-Case Effectiveness Assessment 165
7.3 Instrumenting Programs for Assertion Checking 166
7.4 Instrumenting Programs for Data-Flow-Anomaly Detection 169
7.5 Instrumenting Programs for Trace-Subprogram Generation 181
Exercises 192

Appendix A: Logico-Mathematical Background 194

Appendix B: Glossary 213

Appendix C: Questions for Self-Assessment 220

Bibliography 237

Index 253

PREFACE

The ability to detect latent errors in a program is essential to improving program reliability. This book provides an in-depth review and discussion of the methods of software error detection using three different techniqus: testing, static analysis, and program instrumentation. In the discussion of each method, I describe the basic idea of the method, how it works, its strengths and weaknesses, and how it compares to related methods.

I have writtent this book to serve both as a textbook for students and as a technical handbook for practitioners leading quality assurance efforts. If used as a text, the book is suitable for a one-semester graduate-level course on software testing and analysis or software quality assurance, or as a supplementary text for an advanced graduate course on software engineering. Some familiarity with the process of software quality assurance is assumed. This book provides no recipe for testing and no discussion of how a quality assurance process is to be set up and managed.

In the first part of the book, I discuss test-case selection, which is the crux of problems in debug testing. Chapter 1 introduces the terms and notational conventions used in the book and establishes two principles which together provide a unified conceptual framework for the existing methods of test-case selection. These principles can also be used to guide the selection of test cases when no existing method is deemed applicable. In Chapters 2 and 3 I describe existing methods of test-case selection in two categories: Test cases can be selected based on the information extracted form the source code of the program as described in Chapter 2 or from the program specifications, as described in Chapter 3. In Chapter 4 I tidy up a few loose ends and suggest how to choose a method of test-case selection.

I then proceed to discuss the techniques of static analysis and program instrumentation in turn. Chapter 5 covers how the symbolic trace of an execution path can be analyzed to extract additional information about a test execution. In Chapter 6 I address static analysis, in which source code is examined systematically, manually or automatically, to find possible symptoms of programming errors. Finally, Chapter 7 covers program instrumentation, in which software instruments (i.e., additional program statements) are inserted into a program to extract information that may be used to detect errors or to facilitate the testing process.

Because precision is necessary, I have made use throughout the book of concepts and notations developed in symbolic logic and mathematics. A review is included as Appendix A for those who may not be conversant with the subject.

I note that many of the software error detection methods discussed in this book are not in common use. The reason for that is mainly economic. With few exceptions, these methods cannot be put into practice without proper tool support. The cost of the tools required for complete automation is so high that it often rivals that of a major programming language compiler. Software vendors with products on the mass market can afford to build these tools, but there is no incentive for them to do so because under current law, vendors are not legally liable for the errors in their products. As a result, vendors, in effect, delegate the task of error detection to their customers, who provide that service free of charge (although vendors may incur costs in the form of customer dissatisfaction). Critical software systems being built for the military and industry would benefit from the use of these methods, but the high cost of necessary supporting tools often render them impractical, unless and until the cost of supporting tools somehow becomes justifiable. Neverthless, I believe that knowledge about these existing methods is useful and important to those who specialize in software quality assurance.

I would like to take opportunity to thank anonymous reviewers for their comments; William E. Howden for his inspiration; Raymond T. Yeh, José Muñoz, and Hal Watt for giving me professional opportunities to gain practical experience in this field; and John L. Bear and Marc Garbey for giving me the time needed to complete the first draft of this book. Finally, my heartfelt thanks go to my daughter, Joyce, for her active and affectionate interest in my writing, and to my wife, Shih-wen, for her support and for allowing me to neglect her while getting this work done.

J. C. HUANG

Houston

1 Concepts, Notation, and Principles

Given a computer program, how can we determine whether or not it will do exactly what it is intended to do? This question is not only intellectually challenging, but also of primary importance in practice. An ideal solution to this problem would be to develop certain techniques that can be used to construct a formal proof (or disproof) of the correctness of a program systematically. There has been considerable effort to develop such techniques, and many different techniques for proving program correctness have been reported. However, none of them has been developed to the point where it can be used readily in practice.

There are several technical hurdles that prevent formal proof of correctness from becoming practical; chief among them is the need for a mechanical theorem prover. The basic approach taken in the development of these techniques is to translate the problem of proving program correctness into that of proving a certain statement to be a theorem (i.e., always true) in a formal system. The difficulty is that all known automatic theorem-proving techniques require an inordinate amount of computation to construct a proof. Furthermore, theorem proving is a computationally unsolvable problem. Therefore, like any other program written to solve such a problem, a theorem prover may halt if a solution is found. It may also continue to run without giving any clue as to whether it will take one more moment to find the solution, or whether it will take forever. The lack of a definitive upper bound of time required to complete a job severely limits its usefulness in practice.

Until there is a major breakthrough in the field of mechanical theorem proving, which is not foreseen by the experts any time soon, verification of program correctness through formal proof will remain impractical. The technique is too costly to deploy, and the size of programs to which it is applicable is too small (relative to that of programs in common use). At present, a practical and more intuitive solution would be to test-execute the program with a number of test cases (input data) to see if it will do what it is intended to do.

How do we go about testing a computer program for correctness? Perhaps the most direct and intuitive answer to this question is to perform an *exhaustive test*: that is, to test-execute the program for all possible input data (for which the program is expected to work correctly). If the program produces a correct result for every possible input, it obviously constitutes a direct proof that the program is correct. Unfortunately, it is in general impractical to do the exhaustive test for any nontrivial program simply because the number of possible inputs is prohibitively large.

To illustrate, consider the following C++ program.

Program 1.1

```
main ()
{
int i, j, k, match;

   cin  >> i  >> j  >> k;
   cout << i << j << k;
   if (i <= 0 || j <= 0 || k <= 0
         || i+j <= k || j+k <= i || k+i <= j)
      match = 4;
   else if !(i == j || j == k || k == i)
      match = 3;
   else if (i != j || j != k || k != i)
      match = 2;
   else match = 1;
   cout << match << endl;
}
```

If, for an assignment of values to the input variables i, j, and k, the output variable match will assume a correct value upon execution of the program, we can assert that the program is correct for this particular test case. And if we can test the program for all possible assignments to i, j, and k, we will be able to determine its correctness. The difficulty here is that even for a small program like this, with only three input variables, the number of possible assignments to the values of those variables is prohibitively large. To see why this is so, recall that an ordinary integer variable in C++ can assume a value in the range −32,768 to +32,767 (i.e., 2^{16} different values). Hence, there are $2^{16} \times 2^{16} \times 2^{16} = 2^{48} \approx 256 \times 10^{12}$ possible assignments to the input triple (i, j, k). Now suppose that this program can be test-executed at the rate of one test per microsecond on average, and suppose further that we do testing 24 hours a day, 7 days a week. It will take more than eight years for us to complete an exhaustive test for this program. Spending eight years to test a program like this is an unacceptably high expenditure under any circumstance!

This example clearly indicates that an exhaustive test (i.e., a test using all possible input data) is impractical. It may be technically doable for some small programs, but it would never be economically justifiable for a real-world program. That being the case, we will have to settle for testing a program with a manageably small subset of its input domain.

Given a program, then, how do we construct such a subset; that is, how do we select test cases? The answer would be different depending on why we are doing the test. For software developers, the primary reason for doing the test is to find errors so that they can be removed to improve the reliability of the program. In that case we say that the tester is doing *debug testing*. Since the main goal of debug testing is to find programming errors, or *faults* in the Institute of Electrical and Electronics

Engineers (IEEE) terminology, the desired test cases would be those that have a high probability of revealing faults.

Other than software developers, expert users of a software system may also have the need to do testing. For a user, the main purpose is to assess the reliability so that the responsible party can decide, among other things, whether or not to accept the software system and pay the vendor, or whether or not there is enough confidence in the correctness of the software system to start using it for a production run. In that case the test cases have to be selected based on what is available to the user, which often does not include the source code or program specification. Test-case selection therefore has to be done based on something else.

Information available to the user for test-case selection includes the probability distribution of inputs being used in production runs (known as the *operational profile*) and the identity of inputs that may incur a high cost or result in a catastrophe if the program fails. Because it provides an important alternative to debug testing, possible use of an operational profile in test-case selection is explained further in Section 4.2. We discuss debug testing in Chapters 2 and 3. Chapter 4 is devoted to other aspects of testing that deserve our attention. Other than testing as discussed in Chapters 2 and 3, software faults can also be detected by means of analysis, as discussed in Chapters 5 through 7.

When we test-execute a program with an input, the test result will be either correct or incorrect. If it is incorrect, we can unequivocally conclude that there is a fault in the program. If the result is correct, however, all that we can say with certainty is that the program will execute correctly for that particular input, which is not especially significant in that the program has so many possible inputs. The significance of a correct test result can be enhanced by analyzing the execution path traversed to determine the condition under which that path will be traversed and the exact nature of computation performed in the process. This is discussed in Chapter 5.

We can also detect faults in a program by examining the source code systematically as discussed in Chapter 6. The analysis methods described therein are said to be static, in that no execution of the program is involved. Analysis can also be done dynamically, while the program is being executed, to facilitate detection of faults that become more obvious during execution time. In Chapter 7 we show how dynamic analysis can be done through the use of software instruments.

For the benefit of those who are not theoretically oriented, some helpful logico-mathematical background material is presented in Appendix A. Like many others used in software engineering, many technical terms used in this book have more than one possible interpretation. To avoid possible misunderstanding, a glossary is included as Appendix B. For those who are serious about the material presented in this book, you may wish to work on the self-assessment questions posed in Appendix C.

There are many known test-case selection methods. Understanding and comparison of those methods can be facilitated significantly by presenting all methods in a unified conceptual framework so that each method can be viewed as a particular instantiation of a generalized method. We develop such a conceptual framework in the remainder of the chapter.