# FUNCTIONAL PROGRAM TESTING & ANALYSIS

McGRAW-HILL SERIES IN SOFTWARE ENGINEERING AND TECHNOLOGY

WILLIAM E. HOWDEN

# FUNCTIONAL PROGRAM TESTING AND ANALYSIS

**William E. Howden**

*University of California at San Diego*

**McGraw-Hill Book Company**

New York St. Louis San Francisco Auckland Bogotá Hamburg
London Madrid Mexico Milan Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

This book was set in Times Roman by Publication Services.
The editors were Karen M. Jackson and Joseph F. Murphy;
the designer was Joan E. O'Connor;
the production supervisor was Denise L. Puryear.
Project supervision was done by Publication Services.
R.R. Donnelley & Sons Company was printer and binder.

**FUNCTIONAL PROGRAM TESTING AND ANALYSIS**



234567890 DOCDOC 89210987

ISBN 0-07-030550-1

Library of Congress Cataloging-in-Publication Data
Howden, William E.
Functional program testing and analysis.
(Software engineering series)
1. Computer programs—Testing. I. Title.
II. Series: Software engineering series (New York, N.Y.)
QA76.6.H7 1987 005.3'028'7 86-28220
ISBN 0-07-030550-1

# PREFACE

Testing is an unavoidable part of any responsible effort to develop a software system. Over the past 15 years more and more attention has been given to this topic, but it is still a relatively new one and has consisted primarily of an unintegrated collection of methods whose justification is intuitive rather than scientific or mathematical.

This book presents an integrated approach to program testing and analysis which has a sound mathematical basis. It describes both previous techniques, and how they fit together, as well as new methods. It provides a general approach to testing and validation that incorporates all important software life cycle products, including requirements and general and detailed designs. The theory part of the book contains mathematical results that can be used to characterize the effectiveness of different functional testing and analysis methods. The results can be used to prove that well-defined classes of faults and failures will be discovered by specific techniques. Functional testing and analysis is a general approach to verification and validation and not only integrates current techniques, but indicates fruitful directions for continued research and development.

This book is written for the practicing software engineer and for the advanced undergraduate or graduate student. Although some familiarity with basic software engineering concepts would be useful, all of the related ideas are explained. This includes material on requirements and on general and detailed design. The theoretical foundations part of the book relies on several branches of mathematics, and all of this material is also fully explained.

If used as a text, the book is suitable for courses in software engineering or in testing and validation. For a general course on software engineering it is recommended that it be used along with a good book on design or a general software engineering text that covers design methodology in depth.

The book has three major sections. The first section consists of the first three chapters. It contains a discussion of functions, states, and types in programs and illustrates their central role in requirements, design, and coding with two detailed examples. The second section contains the fourth chapter and includes the theoretical foundations which are used to put functional testing and analysis on a sound mathematical basis. This section outlines a theory of testing based on fault and failure analysis, and contains results in statistics, algebra, and graph theory. The basic concepts of the first and second sections are combined to produce the systematic approach to testing and analysis described in the third section, consisting of Chapters 5, 6, and 7.

It is not necessary for the reader to learn all of the mathematical results in the fourth chapter in order to apply functional testing and analysis. It is possible to skip most of this chapter and then only to look back at selected material referenced in Chapters 5 and 6. The exception is Sections 4.1 and 4.2, which should be read after Chapters 1, 2, and 3, and before going on to Chapters 5 through 7. Section 4.2 describes the basic integrating concepts used to build the systematic testing methodology that is detailed in the remaining chapters of the book.

Many of the methods described in the book depend on the use of software tools for support. A complete collection of functional testing and analysis tools is under construction and additional information about them is available from *Critical Software Systems*, P.O. Box 1241, Solana Beach, California 92075.

I would like to take this opportunity to thank all of my professional colleagues in software engineering, with whom I have had many insightful conversations. I would like to thank Patrick Dymond, Michael Fredman and Elias Masry at UCSD for discussions on different aspects of the theoretical material. The research upon which this book is based was funded by the Office of Naval Research, and I would like to thank the ONR for their continued support. I would also like to thank Chiquita Payne for her dedication and patience in typing the manuscript and Sue Sullivan who worked on earlier drafts of the material. Finally, I would like to acknowledge my dependence on that source of all creativity and knowledge from whom I have derived what modest inspiration guided the writing of this book.

*William E. Howden*
*Solana Beach, California*

# CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1 TESTING

Testing is a fundamental part of all branches of engineering, and it is an essential part of software development. In manufacturing processes that involve physical products, testing is carried out to detect materials defects. In software development all errors are human errors, and it is tempting to hope that errors can be eliminated with better development methods, better trained programmers, or some kind of programming language or environment that proves to be a universal panacea for human fallibility. However, there is no reason to believe that this will ever be the case, except in special circumstances, and testing will continue to play a major role in software engineering.

## 1.2 TESTING AND THE SOFTWARE LIFE CYCLE

In the modern life-cycle approach to software development, a variety of documents are produced before and during software development which make the process less error-prone and more systematic. Life-cycle software development may include requirements analysis methods such as data-flow analysis,[1] specification systems such as Problem Statement Language/Problem Statement Analysis (PSL/PSA),[2] general design methods such as Structured Design[3] and Structural Analysis and Design Technique (SADT)[4] and detailed design techniques such as Program Design Languages[5] and Nassi-Schneiderman diagrams.[6]

The advantage of systematic life-cycle development is that it reduces complexity by separating the description of different aspects of a software system into parts. This simplifies the description and makes software development easier and more systematic. It enables the detection of basic errors earlier in the development process and decreases the cost of their elimination.

The approach to testing outlined in this book uses the information about a system which is contained in a typical set of life-cycle documents. The approach can also be applied, but less easily, to a system which consists only of programs and no other documents.

## 1.3 TESTING, HIGH-LEVEL LANGUAGES, AND SPECIFICATIONS

One way to reduce errors in programs is to use high-level languages, or languages which are specialized for particular areas of application. This reduces errors by reducing the logical complexity of the object that must be created by a programmer. The use of very high-level languages is feasible in applications areas where large numbers of very similar programs are required, and new programs can be constructed by providing the system specification in some tabular format. There are a variety of commercial data processing program construction systems like this, in which all the user has to do is specify file or data-base formats. This is the only place in which testing can, foreseeably, become unnecessary, but it is because there is no programming being done.

Ideally, it may appear that a general purpose formal specifications language could be developed which would allow the specification of any program, and that testing would become unnecessary because the program would be generated mechanically from the specification and hence be correct. This is an unlikely possibility. If the specifications language were general purpose, then the programmer would have to specify as much logical information in the specification as he previously had to in a program, and the net result would be that the errors would be in the specification now instead of the program. In principle, such a specifications language would be a kind of programming language, possibly a worse one than a conventional language, and no real problem would be solved by its use. The success with high-level specifications languages for data processing systems is the result of specialization rather than specification.

If a family of specialized specification systems could be developed that would have the advantages of the data processing specification systems, then errors and testing would become less important for the applications systems analyst. But this would be because he would no longer really be a programmer. The programming would be in the construction of the translator which transformed the specifications into programs, and testing would play its traditional, necessary role here.

## 1.4 TESTING AND PROOFS OF CORRECTNESS

It has often been quoted that "testing can only be used to detect the presence of bugs, not to prove their absence." The implication is that testing should be abandoned and some form of proof method adopted. Before going into the limitations of proofs it should be pointed out that testing has some very important advantages over proof methods.

There is no such thing as an absolute proof of correctness; there are only proofs of equivalency. Program proofs of correctness are proofs that one description of a function is equivalent to another. Usually, one is a state description in a formal logical language and the other is an algorithm description in a programming language. It often happens during software development that it is necessary to invent some expression or function to carry out a computation whose full specification is not yet available. A few selected cases may be known, and a computational procedure constructed that is characterized by those cases. After the experimental period, based on testing, it will be possible to construct a more complete specification.

The dependency of proofs on formal specifications is a pervasive difficulty in the attempt to use proofs of correctness. Formal specifications for combinatorial or mathematical programs may be concise and may clearly describe the intended computational effect of a program, but for other programs this is often not the case. There are many programs for which formal specifications are artificial and obscure, and much more difficult to construct, read, and understand than the specified program. Formal specifications describe initial and final states of a system in terms of relationships between variables. Programs are transformers which change initial states to final states. For many programs there is no reasonable way to express relationships between initial and final states except in the form of the program which establishes the relationship.

Another factor that inhibits the use of proofs of correctness is the difficulty of constructing a program proof. It is a tedious, error-prone task, consisting of many cases and inductions. Techniques have been developed to simplify the proof process, but proofs may be longer and more difficult to understand than the proved programs. In addition, proofs of correctness are usually proofs of the correctness of the logic of an algorithm and not of programs. The peculiarities of fixed length word sizes, floating point numbers, and other computer dependent properties are ignored. If they were not, the proofs would become even more detailed and difficult to understand.

Although, in principle, it may be possible to prove the correctness of a program, it may simply not be cost efficient. There is often a sense of urgency in software development. A usable product must be completed within a time frame that will not allow formal specification and proof. No added value would be introduced into the system or program by formal

specifications and proofs, even if this were to increase its relative mean time between failure from 75 to 100 percent.

Continuing research in program proving will expand the scope of applicability of and ease of use of proofs. For a variety of reasons, however, it is unlikely that proofs will replace testing. Perhaps the most fruitful area of application for proofs of correctness is in the design stage, where they can be used to prove that certain general algorithmic concepts have some desired property. Such proofs may be independent of the eventual structure of a program and use traditional mathematical techniques, such as proof by contradiction, that are difficult to use when proofs are tied to program structure. Once the basic design ideas are proved, then testing and analysis can be used to confirm that a designed and implemented program conforms to those ideas. Both proving and testing are considered to have their place, and their cooperative use should continue to increase.

## 1.5 FUNCTIONAL TESTING AND ANALYSIS

Current Institute of Electrical and Electronics Engineers (IEEE) standards define a bug in a program as a *fault* and the incorrect behavior of the program induced by the fault as a *failure*. This terminology will be used throughout the book.

Testing has traditionally been unsystematic and unreliable because there was no unifying approach to combining different testing methods and no theory to characterize the classes of faults and failures that could be found by different methods. This book describes such an approach. The basic idea is that programs can be viewed as collections of functions which are synthesized from other functions, and that program faults correspond to faults in synthesis. There are two very general ways of joining functions together—functional synthesis and structural synthesis.

In *functional synthesis*, expressions and other simple programming constructs are used to construct an input-output function. The programmer may not know the specification of the new function but is expected to know what the correct output would be for selected input. General classes of functional synthesis faults are defined, along with the kinds of tests needed to reveal those faults. *Functional testing* involves the testing of functions formed by functional synthesis over fault revealing test data. It is called functional testing since the emphasis is on execution of functions and examination of their input and output data.

In *structural synthesis*, functions are joined together into graph-like structures which describe the sequences in which they should be executed. Failures correspond to wrong sequences of function invocation or to wrong transmission of data between one function and another. *Functional analysis* involves the analysis of programs for structural synthesis failures. In some

cases failures can be detected by the presence of inconsistent data interfaces, and in others the programmer is expected to have a description of correct function usage sequences.

Functional testing is a fault analysis method since it depends on the definition and detection of specified classes of faults. In functional testing the programmer exhaustively tests for classes of faults. Exhaustive testing for all possible failures is not possible since the number of different possible input-output pairs that would have to be examined is prohibitively large. Exhaustive testing for classes of faults is feasible since the number of different possible commonly occurring classes of functional synthesis faults is relatively small. Functional analysis is a failure analysis rather than a fault analysis method since it involves looking for all possible structural synthesis failures, that is, failures in function sequencing. This is feasible since it is possible to prove that if there are any such failures, they will occur in a small finite set of sequences which can be exhaustively examined. Functional analysis is more powerful in this sense, since it can detect the presence of any fault which results in a function sequence failure. Functional testing can detect only failure-causing faults for which it has been possible to construct fault-testing rules. This may leave kinds of failures not caused by those classes of faults undetected.

## 1.6 VERIFICATION AND VALIDATION

The words verification and validation are often used in discussing testing and analysis, although they are used in a variety of contradictory and confusing ways. The entire process of examining a software product to confirm that it operates as intended is often referred to as "V and V" without any attempt to explain what the differences between the two words are or why two words are even necessary. The word verification is sometimes associated uniquely with proofs of correctness.

The following definitions are consistent with the original way in which these two words were used. *Verification* refers to the activity of comparing a software development product with some description of that product that occurred earlier in the development process. General designs may be verified with respect to requirements analysis documents, detailed designs with respect to general designs, and programs with respect to detailed designs. When the only development products are specifications and programs, then programs are verified by comparing them with their specifications. This is consistent with the use of the word verification to refer to proofs of correctness, in which programs are proved equivalent to formal specifications.

*Validation* means to compare a software development product with the user's perceived requirements for that product. The term is sometimes used to refer to customer acceptance testing when the final software system is

tested in the environment for which it was intended. Other products besides code can be validated. Requirements and design products may be compared directly with user expectations through the use of prototypes or simulation.

Functional testing and analysis methods can be used for both verification and validation and in this book are not organized into separate sections based on this classification. They can be used with different life-cycle products, involving information from requirements, general design, detailed design, and coding.

## 1.7 TOOLS

Many of the testing and analysis methods which will be discussed depend on the availability of tools. The basic features of such tools are described and some of the currently available tools are referenced.

## 1.8 ORGANIZATION OF THE BOOK

The remaining part of the book begins with a discussion of functions, states, and types. Functions and states are considered to be the basic concepts in a system, and their occurrence in requirements, general and detailed designs, and code is discussed. This is followed by a chapter on the theoretical foundations of functional testing and analysis. The basic ideas of testing and analysis are discussed along with the fundamental mathematical results upon which the method is based.

The first part of the theoretical foundations chapter contains a discussion of the impossibility of proving correctness by testing. This is followed by a discussion of what can be proved. Sections on statistical testing, algebraic foundations for functional testing, and graph theory foundations for functional analysis are included.

The next two chapters in the book describe functional testing and functional analysis. The application of the basic theoretical results to practical functional testing methods is described in the first of these two chapters. The second describes module and structural interface analysis, the basic techniques of functional analysis.

The final chapter is on the planning and management of software testing. It contains advice on the selection and use of testing and analysis methods. Three software quality assurance packages are suggested, ranging from an inexpensive, manual approach to an expensive, fully automated approach.

## REFERENCES

1. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, N.Y., 1978.
2. D. Teichroew and E. Hershey, PSL/PSA—Computer aided techniques for structure and

documentation and analysis of information processing systems, *IEEE Transactions on Software Engineering*, SE-6, 1980.
3. E. Yourdon and L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, N.J., 1979.
4. D. T. Ross, Structured Analysis (SA): A language for communicating ideas, *IEEE Transactions on Software Engineering*, SE-3, January 1977.
5. S. H. Caine and K. E. Gordon, PDL—A tool for software design, *Proceedings National Computer Conference*, vol. 44, 1975.
6. I. Nassi and B. Schneiderman, Flowchart techniques for structured programming, *ACM Sigplan Notices*, 8, 1973.

# CHAPTER 2

# FUNCTIONS

## 2.1 FUNCTIONS AND DATA TYPES

Mathematically, all functions are of the form $f: a \rightarrow b$ where $f$ is a function which, for every object of type $a$, returns an object of type $b$. In the simplest cases, types are sets of simple objects, such as integers or reals, and in more complex cases they are structured objects having different components.

In defining a function, the data which it operates on and which it produces may be discussed either directly in terms of types or indirectly in terms of properties of variables and data structures used to store data of a specified type. The latter is more common in the more detailed phases of software development, when it becomes necessary to choose variables and data structures.

## 2.2 SOFTWARE DEVELOPMENT AND FUNCTIONS

Functions and their data are the basic conceptual units that are used to build software. They are used not only in programs but also to construct requirements, and general and detailed designs. They may be both formally and informally defined. The central role of functions in software development is illustrated in the following sections. In the first section, the functions in the informal requirements for a data processing system and the functions in the formal specification for a scientific program are described. This is followed by sections describing the functions in the general and detailed designs for these examples.

It is not necessary for the reader to understand the examples in complete detail. The point is to illustrate how function and data-type definitions arise in and are critical concepts in all phases of software development.

## 2.3 FUNCTIONS AND REQUIREMENTS

In the first of the following examples, Structured Analysis is used to give an informal set of requirements for a computerized dating system. In the second, mathematical formulae are used to give formal specifications for a scientific program. These two examples are representative of the variety of methods that can be used to provide preliminary descriptions of a proposed program. In general, requirements will be informal for systems and programs that are new and are not translations of previously constructed programs, or for translations from a formal noncomputational logical language, such as mathematics, to a computational one.

The purpose of the examples is to illustrate the central role of functions in requirements and to show how their presence can be easily identified in Structured Analysis documents.

### 2.3.1 Dating System Example

In Structured Analysis[1] the requirements for a system are described using data-flow diagrams and data dictionaries. The method is commonly used to describe the requirements for data processing systems. Data-flow diagrams consist of arcs, which represent data flow, and nodes, which represent data transformations. Figure 2.1 contains the data-flow diagram for a computerized dating system. The system is expected to find the best date in its database for individual clients. Clients and dates are part of the same population and each client's dating information is expected to be in the *datefile* database. The information stored for each date consists of the personal and physical characteristics of the date as well as the personal and physical

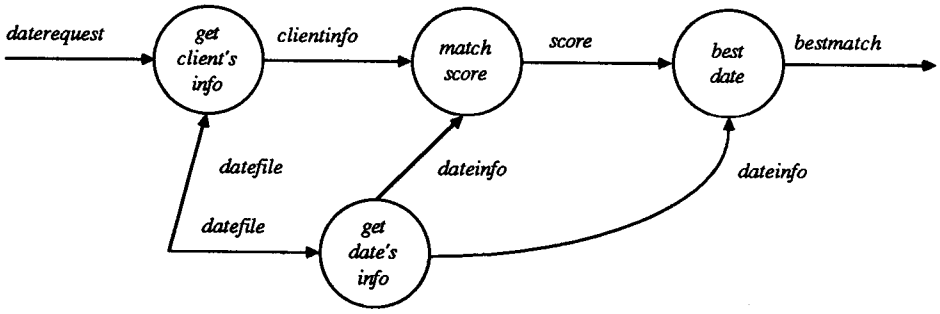


**FIGURE 2.1.** Data-flow diagram for dating system.