Software Engineering

*by **Nguyen Xuan Huy***

Institute of Information Technology,

National Center for Natural Science and Technology

Email: nxhuy@ioit.ncst.ac.vn

Duration: 45 hours

Chapter 5 Software Verification

Objectives

This chapter describes various test methods and discusses how testing and debugging should be carried out and documented. The testing process and the stages of testing and the advantages and disadvantages of top-down testing are discussed. A formal method of program verification is described.

Contents

5.1 Test methods

- 5.1.1 Verification of algorithms
- 5.1.2 Static program analysis
- 5.1.3 Dynamic testing
- 5.1.4 Black-box and white-box testing
- 5.1.5 Top-down and bottom-up testing

5.2 Mathematical program verification

5.3 Debugging

References and selected reading

5.1 Test methods

In carrying out testing we will wish to address some or all of the following questions:

- How is functional validity tested?
- What *classes* of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?
- Which activities are necessary for the systematic testing of a software system?
- What is to be tested?
  - *System specifications,*

- *Individual modules,*
- *Connections between modules,*
- *Integration of the modules in the overall system*
- *Acceptance of the software product.*

Testing the system specifications

- The system specifications provide the foundation for the planning and execution of program tests.
- The comprehensibility and unambiguity of the system specifications is the prerequisite for testing a software system.
- The goal of the specification test is to examine the completeness, clarity, consistency and feasibility of the system specifications.

Techniques of cause and effect ([Hughes 1976]).

- *Causes* are conditions or combinations of conditions that are prescribed or that have arisen
- *Effects* are results or activities.
- The specification test is intended to show whether causes (data, functions) are specified for which no effects (functions, results) are defined or vice versa.
- Prototypes of user interface and system components play an important role in testing the system specifications; they permit experimental inspection of the system specifications.
- The specifications test must always be carried out in cooperation with the user.

Testing modules

- Modules encapsulate data structures and operations that work with these data structures.
- The *goal* of the module test is to *identify all deviations* of the implementation from the module specifications.
- Module testing requires first testing the individual functions and then their interplay.
- Modules are not independently programs; their execution requires the existence of other modules (due to the networking of modules through import and inheritance relationships).
- The test can only executed if the environment of the module already exists or can be simulated.
- Part of work in module testing consists of creating a suitable test environment that permits of the module, examination of the results of its invocation, and simulation of modules yet to be developed (see Figure 5.1).
- Keep the test as simple as possible, for increasing complexity also raises the probability the environment itself contains errors.

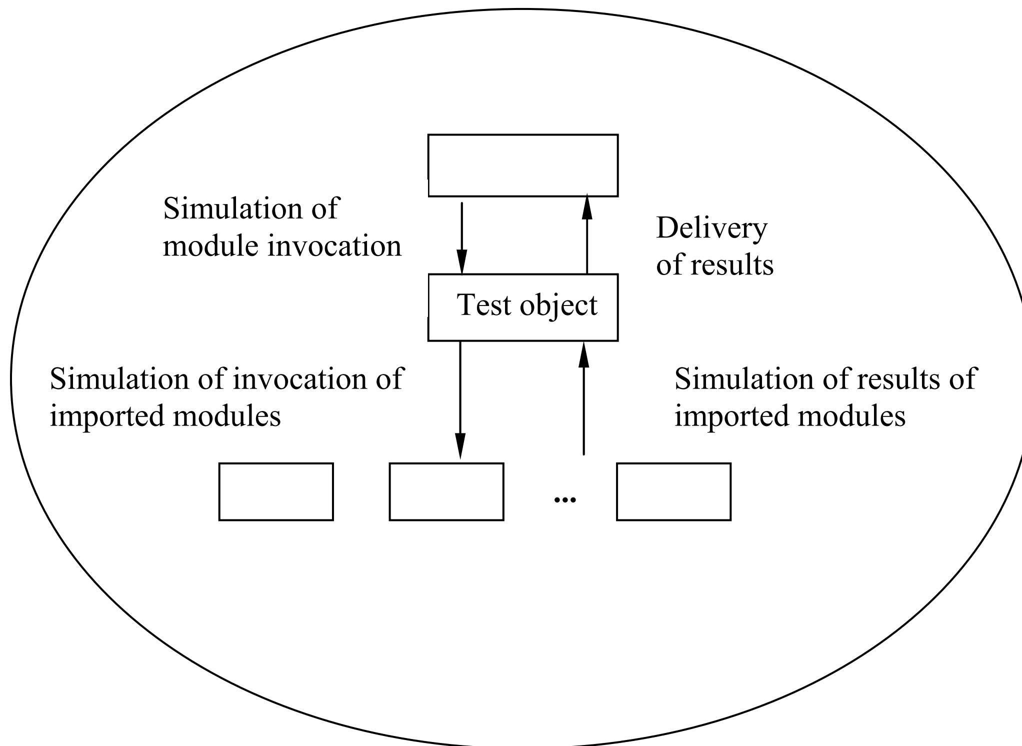

Figure 5.1 Test object and test environment

Testing module connections *(testing of subsystems)*

- A *subsystem* comprises a problem-specific collection of individual (tested) modules.
- The *goal* is to test the *connections between the individual modules* of a system.
- The test encompasses checking the correctness of module communication hardly differs in its approach from module testing.
- Subsystems require the creation of test environments. After testing subsystems, several subsystems are combined to a (hierarchically superordinate) subsystem, and the connections between subsystems are tested (*integration test).*

Test of the overall system

- Test the integration of all subsystems.
- The *goal* is to *detect all deviations* of the system behavior from that in the requirements definition.
- The goal is not only to test the completeness of the user requirements and the correctness of the results, but also to test whether the software system is reliable and robust with respect to erroneous input data.
- The system's adherence to nonfunctional requirements, e.g., the required efficiency, must also be tested.

- The scope and number of tests as well as the selection of appropriate test data vary from case to case; the present state of the art offers no universal recipe for testing software systems.
- The effort vested in testing of the overall system must be in some proportion to the amount of damage that erroneous operation of the software systems can inflict.

Acceptance test

- The development of a software product ends with the *inspection (acceptance test)* by the user.
- At inspection the software system is tested with *real data* under *real conditions of use.*
- The *goal* of the inspection is to *uncover all errors* that arose from such sources as misunderstandings in consultations between users and software developers, poor estimates of application-specific data quantities, and unrealistic assumptions about the real environment of the software system.

5.1.1 Verification of algorithms

- Two important questions:
  - Whether the chosen approach will actually lead to the required solution?
  - Whether the already developed components are correct, i.e., whether they fulfill the specifications?
- Ideally we would like to be able to prove at any time that for every imaginable input the algorithms included in the design and their interplay will deliver the expected results according to the specification.
- Since today's large software systems permit the complete description of neither the input set nor the anticipated result set (due to combinatorial explosion), such systems can no longer be completely tested. Thus efforts must be made to achieve as much clarity as possible already in the design phase about the correctness of the solution.
- Proof of correctness cannot be handled without formal and mathematical tools ([McCarthy 1962], [Naur 1966], [Hoare 1969], [Manna 1969], [Knuth 1973], [Dijkstra 1976], see overview in [Elspas 1972].)
- The use of a verification procedure forces the software engineer to reproduce all design decisions and thus also helps in finding logical errors. Verification is an extremely useful technique for early detection of design errors and also complements the design documentation. Verification itself can be fallible, however, it cannot replace testing of a software product.
- Verification can be used successfully to prove the correctness of short and simple algorithms. For the production of larger software systems, the difficulties rise so sharply that it should be clear that verification fails as a practical test aid.

5.1.2 Static program analysis

- *Static program analysis* seeks to detect errors without direct execution of the test object.
- The activities involved in static testing concern *syntactic, structural and semantic analysis of the test object* (compare [Ramamoorthy 1974] and [Schmitz 1982]).

- The goal is to localize, as early as possible, error-prone parts of the test object.
- The most important activities of static program analysis are:

  - Code inspection
  - Complexity analysis
  - Structural analysis
  - Data-flow analysis

Code inspection

- *Code inspection* is a useful technique for localizing design and implementation errors. Many errors prove easy to find if the author would only read the program carefully enough.
- The idea behind code inspection is to have the author of a program discuss it step by step with other software engineers.
- Structure of the four person team ([Fagan 1976]):

  1. An experienced software engineer who is not involved in the project to serve as moderator
  2. The designer of the test object
  3. The programmer responsible for the implementation
  4. The person responsible for testing

  The moderator notes every detected error and the inspection continues.

- The *task of the inspection team* is to *detect, not to correct,* errors. Only on completion of the inspection do the designer and the implementor begin their correction work.

Complexity analysis

- The *goal* of *complexity analysis* is to *establish metrics* for the complexity of a program ([McCabe 1976], [Halstead 1972], [Blaschek 1985])
- These metrics include complexity measures for modules, nesting depths for loops, lengths of procedures and modules, import and use frequencies for modules, and the complexity of interfaces for procedures and methods.
- The results of complexity analysis permit statements about the quality of a software product (within certain limits) and localization of error-prone positions in the software system.
- Complexity is difficult to evaluate objectively; a program that a programmer perceives as simple might be viewed as complex by someone else.

structural analysis

- The *goal* of *structural analysis* is to uncover structural anomalies of a test object.

Data-flow analysis

- Data-flow analysis is intended to help discover data-flow anomalies.
- Data-flow analysis provides information about whether a data object has a value before its use and whether a data object is used after an assignment.
- Data-flow analysis applies to both the body of a test object and the interfaces between test objects.

5.1.3 Dynamic testing

- *For dynamic testing* the test objects are executed or simulated.
- Dynamic testing is an imperative process in the software life cycle. Every procedure, every module and class, every subsystem and the overall system must be tested dynamically, even if static tests and program verifications have been carried out.
- The activities for dynamic testing include:
  - Preparation of the test object for error localization
  - Availability of a test environment
  - Selection of appropriate test cases and data
  - Test execution and evaluation

5.1.4 Black-box and white-box testing

Every test object can be tested in two principal ways to determine whether it fulfills its specifications:

1. Black-box test (functional test or exterior test): Test of the input/output behavior without considering the inner structure of the test object (interface test)
2. White-box test (structure test or interior test): Test of the input/output behavior considering the inner structures (interface and structure test)

Black-box tests

- *Black-box testing* focuses on the functional requirements of the software.
- Black-box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing attempts to find errors in the following categories:
  1. Incorrect or missing functions,
  2. Interface errors,
  3. Errors in data structures or external database access,
  4. Performance errors, and
  5. Initialization and termination errors.
- *Black-box tests* serve to detect deviations of a test object from its specification. The selection of test cases is based on the specification of the test object without

knowledge of its inner structure. It is also important to design test cases that demonstrate the behavior of the test object on erroneous input data.

- Criteria for test cases [Myers 1979]
  1. Test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing
  2. Test cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.
- The black-box test provides no information about whether all functions of the test object are actually necessary or whether data objects are manipulated that have no effect on the input/output behavior of the test object. Such indicators of design and implementation errors emanate from the white-box test.

White-box tests

- *White-box testing* is one of the most important test methods. For a limited number of program paths, which usually suffices in practice, the test permits correct manipulation of the data structures and examination of the input/output behavior of test objects.
- In white-box testing the test activities involve not only checking the input/output behavior, but also examination of the inner structure of the test object.
- The goal is to determine, for every possible path through the test object, the behavior of the test object in relation to the input data.
- Test cases are selected on the basis of knowledge of the control flow structure of the test object.
- The selection of test cases must consider the following:
  - Every module and function of the test object must be invoked at least once
  - Every branch must be taken at least once
  - As many paths as possible must be followed
- It is important to assure that every branch is actually taken. It does not suffice to merely assure that every statement is executed because errors in binary branches might not be found because not all branches were tested.
- It is important to consider that, even for well-structured programs, in practice it remains impossible to test all possible paths, i.e., all possible statement sequences of a program ([Pressman 1987], see Figure 5.2)

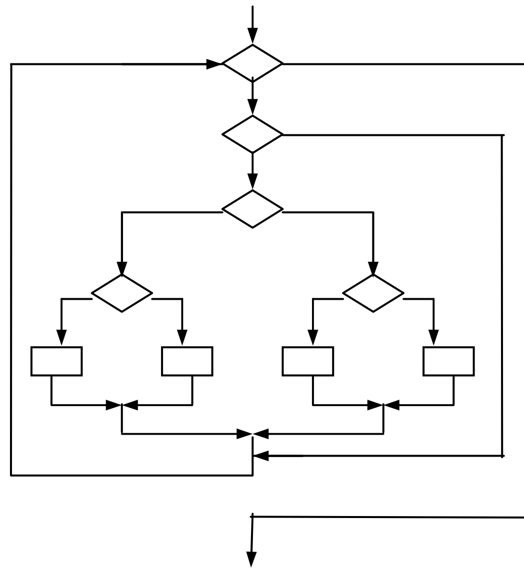

Figure 5.2 Control flowchart: for 10 iterations
there are almost one million paths

5.1.5 Top-down and bottom-up testing

Top-down testing

Method

- The control module is implemented and tested first.
- Imported modules are represented by surrogate modules.
- Surrogates have the same interfaces as the imported modules and simulate their input/output behavior.
- After the test of the control module, all other modules of the software systems are tested in the same way; i.e., their operations are represented by surrogate procedures until the development has progressed enough to allow implementation and testing of the operations.
- The test advances stepwise with the implementation. Implementation and phases merge, and the integration test of subsystems becomes superfluous.

The advantages

- Design errors are detected as early as possible, saving development time and costs because corrections in the module design an be made before their implementation.
- The characteristics of a software system are evident from the start, which enables a simple test of the development state and the acceptance by the user.

- The software system can be tested thoroughly from the start with test cases without providing (expensive) test environments.

The drawbacks

- Strict top-down testing proves extremely difficult because designing usable surrogate objects can prove very complicated, especially for complex operations.
- Errors in lower hierarchy levels are hard to localize.

Bottom-up test

Method

- *Bottom-up testing* inverts the top-down approach.
- First those operations are tested that require no other program components; then their integration to a module is tested.
- After the module test the integration of multiple (tested) modules to a subsystem is tested, until finally the integration of the subsystems, i.e., the overall system, can be tested.

The advantages

- The advantages of bottom-up testing prove to be the drawbacks of top-down testing (and vice versa).
- The bottom-up test method is solid and proven. The objects to be tested are known in full detail. It is often simpler to define relevant test cases and test data.
- The bottom-up approach is psychologically more satisfying because the tester can be certain that the foundations for the test objects have been tested in full detail.

The drawbacks

- The characteristics of the finished product are only known after the completion of all implementation and testing, which means that design errors in the upper levels are detected very late.
- Testing individual levels also inflicts high costs for providing a suitable test environment.

5.2 Mathematical program verification

- If programming language semantics are formally defined, it is possible to consider a program as a mathematical object.
- Using mathematical techniques, it is possible to demonstrate the correspondence between a program and a formal specification of that program.
- Program is proved to be correct with respect to its specification.
- Formal verification may reduce testing costs, it cannot replace testing as a means of system validation.
- Techniques for proving program correctness and axiomatic approaches: [McCarthy 1962], [Hoare 1969], [Dijkstra 1976], [Manna 1969.

The basis of the axiomatic approach

- Assume that there are a number of points in a program where the software engineer can provide assertions concerning program variables and their relationships. At each of these points, the assertions should be invariably true. Say the points in the program are *P(1)*, *P(2),...P(n)*. The associated assertions are *a(l), a(2),...,a(n)*. Assertion *a(1)* must be an assertion about the input of the program and *a(n)* an assertion about the program output.
- To prove that the program statements between points *P(i)* and *P(i+l)* are correct, it must be demonstrated that the application of the program statements separating these points causes assertion *a(i)* to be transformed to assertion *a(i+l)*.
- Given that the initial assertion is true before program execution and the final assertion is true after execution, verification is carried out for adjacent program statements.

Example 5.1

Axioms for proving program correctness

(A1) $\quad$ *if {P} A {R} and {R} B {Q}*

then {P} A; B{Q}

(A2) $\quad$ *if* $P \Rightarrow Q[x/e]$ *then {P} x:=e {Q}*

where *Q[x/e]* is received from *Q* by replacing all the occurrences of variable *x* with *e*

(A3.1) *if* $\{P \wedge E\}$ *A {Q} and* $\{P \wedge !E\}$ *B {Q}*

then {P} if E then A else B {Q}

where *!E* is *not E*

(A3.2) *if* $\{P \wedge E\}$ *A {Q} and* $P \wedge !E \Rightarrow Q$

then {P} if E then A {Q}

(A4) $\quad$ *if* $\{P \wedge E\}$ *A {P}*

then {P} while E do A; $\{P \wedge !E\}$

Example 5.2 The greatest common divisor of two natural numbers

gcd(a,b) (The Euclidian Algorithm)

```
x:=a;
y:=b;
{ Invariant: gcd(x,y) = gcd(a,b) }
while y <> 0  do
        begin
     { (gcd(x,y) = gcd(a,b)) ∧ y <> 0 }
     { gcd(x,y) = gcd(y,x mod y) = gcd(a,b) }
     r := x mod y;
        { gcd(x,y) = gcd(y,r) = gcd(a,b) }
```

$x := y;$

$\{ gcd(x,y) = gcd(x,r) = gcd(a,b) \}$

$y := r;$

$\{ gcd(x,y) = gcd(a,b) \}$

end;

$\{ (gcd(x,y) = gcd(a,b)) \wedge y = 0 \Rightarrow (gcd(x,0) = gcd(a,b) = x) \}$

$\{ x = gcd(a,b) \}$

5.3 Debugging

- Software testing is a process that can be systematically planed and specified. Test design can be conducted, a strategy can be defined, and results can be evaluated against prescribed expectations.
- *Debugging* occurs as a consequence of successful testing. When a test case uncovers an error, debugging is the process that results in the removal of the error.
- Debugging is not testing, but it always occurs as a consequence of testing.
- The debugging process begins with the execution of a test case (Figure 5.3).
- The debugging process attempts to match symptom with cause, thereby leading to error correction.
- Two outcomes of the debugging:

1. The cause will be found, corrected, and removed,
2. The cause will not be found.

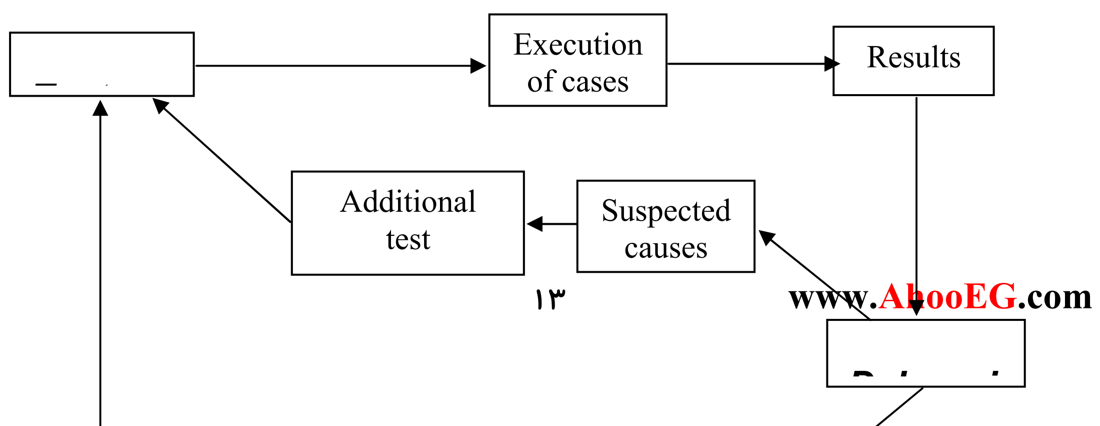

Figure 5.3 Debugging

Characteristics of bugs ([Cheung 1990])

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embeded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging approaches ([Bradley 1985], [Myers 1979])

Three categories for debugging approahes

- Brute force
- Backtracking
- Cause elimination

➢ We apply brute force debugging methods when all else fails.

➢ Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.

➢ Backtracking is a fairly common debugging

References and selected reading

[Beizer 1990] Beizer B., *Software Testing Techniques,* 2d ed., Van Nostrand Reinhold, 1990

[Blaschek 1985] Blaschek G., *Statische Programmanalyse,* Elektronische Rechenanlagen 2, 1985

[Bradley 1985] Bradley J. H., The Science and Art of Debugging, *Computer World,* August 19, 1985, 35-38

[Brilliant 1987] Brilliant S. S., Knight J. C., and Levenson N.G., The Consistent Comparison Problem in N-Version Software, *ACM Software Engineering Notes,* vol. 12, no. 1, January 1987, pp. 29-34

[Cheung 1990] Cheung W. H., Black J.P., and Manning E., A Framework for Distributed Debugging, *IEEE Software,* Jan. 1990, 106-115

[Deutsch 1979] Deutsch M., Verification and Validation, in *Software Engineering,* (Jensen R. and Tonies C., eds.), Prentice-Hall, 1979, pp. 329-408

[Dijkstra 1976] Dijkstra E. W., A *Discipline of Programming;* Prentice Hall, 1976

[Dunn 1984] Dunn R., *Software Defect Removal*, McGraw-Hill, 1984

[Elspas 1972] Elspas B. et al., An Assessment of Techniques for Proving Program Correctness, *Computing Surveys* 4, 1972

[Fagan 1976] Fagan M. E., Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems Journal*, Vol. 15(3), 1976

[Foster 1984] Foster K. A., Sensitive Test Data for Boolean Expressions, *ACM Software Engineering Notes,* vol. 9, no. 2, April 1984, pp. 120-125

[Frankl 1988] Frankl R. G., and Weyuker E. J., An Applicable Family of Automata for Testing Criteria, *IEEE Trans. Software Engineering,* vol. 14, no. 10, October 1988, pp. 1483-1498

[Halstead 1972] Halstead M. H., Natural Laws Controlling Algorithm Structure, *Sigplan Notices* Vol. 7, No. 2, 1972

[Hetzel 1984] Hetzel W., *The Complete Guide to Software Testing,* QED Information Sciences, 1984.

[Hoare 1969] Hoare C. A. R., An Axiomatic Basis for Computer Programming, *Comm. of the ACM*, Vol. 12, 1969

[Howden 1982] Howden W. E., Weak Mutation Testing and the Completeness of Test Cases, *IEEE Trans. Software Engineering,* vol. SE-8, no. 4, July 1982, pp. 371-379

[Hughes 1976] Hughes J. K., Michtom J. I., *A Structured Approach to Programming,* Prentice Hall, 1976

[Jones 1981] Jones T. C., *Programming Productivity Issues for the 80s,* IEEE, Computer Society Press, 1981.

[Knight 1989] Knight J., and Ammann P., Testing Software Using Multiple Versions, Report No. 89029N, *Software Productivity Consortium*, Reston, Virgillia, June 1989

[Knuth 1973] Knuth D. E., *The Art of Computer Programming,* Vol. 1-3, Addison-Wesley, 1968, 1969, 1973

[Manna 1969] Manna Z., The Correctness of Programs, *Journal of Computer and System Science*, Vol. 3, 1969

[McCabe 1976] McCabe T., A Software Complexity Measure, *IEEE Trans. Softwae Engineering,* vol. 2, no. 6, December 1976, pp. 308-320

[McCarthy 1962] McCarthy **J.,** Towards a Mathematical Science of Computation, Proc. *WT Congress*, Amsterdam, North-Holland Publications, 1962

[Miller 1979] Miller E., Automated Tools for Software Engineering, *IEEE Computer Society Press*, 1979, p. 169

[Myers 1979] Myers G., *The Art of Software Testing,* Wiley, 1979

[Naur 1966] Naur P., Proof ofalgodthms by General Snapshots, *BIT* 6, 1966

[Ntafos 1988] Ntafos S. C., A Comparison of Some Structural Testing Strategies, *IEEE Trans. Software Engineering,* vol. 14, no. 6, June 1988, pp. 868-874

[Pomberger 1991] Pomberger G. et al., Prototyping-Oriented Software Development - Concepts and Tools, *Structured Programming,* Vol. 12, No. 1, 1991

[Pomberger 1996] Pomberger G. and Blaschek G., *Object Orientation and Prototyping in Software Engineering*, Translated by Bach R., Prentice Hall, 1996

[Pressman 1987] Pressman R. S., *Software Engineering. A Practitioner's Approach,* McGraw-Hill, 1987

[Ramamoorthy 1974] Ramamoorthy C. V., et al., *Reliability and Integrity of Large Computer Programs;* Lecture Notes in Computer Science Fachtagung ProzeiBrechner 1974, Springer, 1974

[Schmitz 1982] Schmitz P., et al., *Software-Qualitdtssicherung - Testen im Software-Lebenszyklus,* Vieweg, 1982

[Tai 1987] Tai K. C., and H. K. Su, Test Generation for Boolean Expressions, Proc. *COMPSAC '87,* October 1987, pp. 278-283

[Tai 1989] Tai K. C., What to Do Beyond Branch Testing, *ACM Software Engineering Notes,* vol. 14, no. 2, April 1989, p. 58-61

[Van Vleck 1989] Van Vleck T., Three Questions About Each Bug You Find, *ACM Software Engineering Notes,* vol. 14, no. 5, July 1989, 62-63

[White 1980] White L. J., and Cohen E. I., A Domain Strategy for Program Testing, *IEEE Trans. Software Engineering,* vol. SE-6, no. 5, May 1980, pp. 247-257.