Chapter 10. Software Testing

10.1 Introduction

- Software testing is the execution of the software with actual test data.
- Sometimes it is called ***dynamic software testing*** to distinguish it from ***static analysis***, which is sometimes called static testing.
- Static analysis involves analyzing the source code to identify problems.
- Although other techniques are very useful in validating software, actual execution of the software with real test data is essential.

The Goals of Software Testing

- To demonstrate to the developer and the customer that the software meets its requirements.
  - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

- The first goal leads to validation testing
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

Verification vs Validation

- Verification:
  - "Are we building the product right".
  - The software should conform to its specification.

- Validation:
  - "Are we building the right product".
  - The software should do what the user really requires.

Inspections and Testing

- Software inspections
  - Concerned with analysis of the static system representation to discover problems *(*static verification)
  - May be supplement by tool-based document and code analysis.

- Software testing
  - Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.

Inspections and Testing

- Software inspections
  - Concerned with analysis of the static system representation to discover problems *(*static verification)
  - May be supplement by tool-based document and code analysis.

- Software testing
  - Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.

Stages of Testing

- Development testing, where the system is tested during development to discover bugs and defects.
- Release testing, where a separate testing team test a complete version of the system before it is released to users.
- User testing, where users or potential users of a system test the system in their own environment.

Development testing

- Development testing includes all testing activities that are carried out by the team developing the system.
  - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Unit testing

- Unit testing is the process of testing individual components in isolation.
- It is a defect testing process.
- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality.

Component testing

- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

System testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the emergent behaviour of a system.

User testing

- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

- Types of user testing
  - Alpha testing
    - Users of the software work with the development team to test the software at the developer's site.
  - Beta testing
    - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
  - Acceptance testing
    - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

10.2 Software Testing Fundamentals

- ***Exhaustive testing*** is the execution of every possible test case. Rarely can we do exhaustive testing. Even simple systems have too many possible test cases. For example, a program with two integer inputs on a machine with a 32-bit word would have 2^{64} possible test cases (see Review Question 10.1). Thus, testing is always executing a very small percentage of the possible test cases.
- Two basic concerns in software testing are (1) what test cases to use (***test case selection***) and (2) how many test cases are necessary (***stopping criterion***).
- Test case selection can be based on either the specifications (***functional***), the structure of the code (***structural***), the flow of data (***data flow***), or ***random*** selection of test cases.
- The stopping criterion can be based on a ***coverage criterion***, such as executing *n* test cases in each subdomain, or the stopping criterion can be based on a behavior criteria, such as testing until an error rate is less than a threshold *x*.

10.2 Software Testing Fundamentals

- A program can be thought of as a mapping from a ***domain*** space to an answer space or ***range***. Given an input, which is a point in the domain space, the program produces an output, which is a point in the range. Similarly, the specification of the program is a map from a domain space to an answer space.
- A ***test case*** should always include the expected output. If the expected output is different from the actual output, then the tester and/or user can decide which is correct.

10.3 Test Coverage Criterion

- A ***test coverage criterion*** is a rule about how to select tests and when to stop testing. One basic issue in testing research is how to compare the effectiveness of different test coverage criteria. The standard approach is to use the subsumes relationship.

10.3.1 SUBSUMES

- A test criterion A ***subsumes*** test coverage criterion **B** if any test set that satisfies criterion **A** also satisfies criterion **B**.
- This means that the test coverage criterion **A** somehow includes the criterion **B**.
- For example, if one test coverage criterion required every statement to be executed and another criterion required every statement to be executed and some additional tests, then the second criterion would subsume the first criterion.

10.3.2 FUNCTIONAL TESTING

- In functional testing, the specification of the software is used to identify subdomains that should be tested.
- One of the first steps is to generate a test case for every distinct type of output of the program. For example, every error message should be generated.
- Next, all special cases should have a test case. Tricky situations should be tested. Common mistakes and misconceptions should be tested.
- The result should be a set of test cases that will thoroughly test the program when it is implemented.
- This set of test cases may also help clarify to the developer some of the expected behavior of the proposed software

10.3 Test Coverage Criterion

EXAMPLE 10.1

For this classic triangle problem, we can divide the domain space into three subdomains, one for each different type of triangle that we will consider: scalene (no sides equal), isosceles (two sides equal), and equilateral (all sides equal). We can also identify two error situations: a subdomain with bad inputs and a subdomain where the sides of those lengths would not form a triangle. Additionally, since the order of the sides is not specified, all combinations should be tried. Finally, each test case needs to specify the value of the output.

This list of subdomains could be increased to distinguish other subdomains that might be considered significant. For example, in scalene subdomains, there are actually six different orderings, but the placement of the largest might be the most significant based on possible mistakes in programming.

Subdomain	***Example Test Case***
Scalene:	
Increasing size	(3,4,5—scalene)
Decreasing size	(5,4,3—scalene)
Largest as second	(4,5,3—scalene)
Isosceles:	
a=b & other side larger	(5,5,8—isosceles)
a=c & other side larger	(5,8,5—isosceles)
b=c & other side larger	(8,5,5—isosceles)
a=b & other side smaller	(8,8,5—isosceles)
a=c & other side smaller	(8,5,8—isosceles)
b=c & other side smaller	(5,8,8—isosceles)
Equilateral:	
All sides equal	(5,5,5—equilateral)
Not a triangle:	
Largest first	(6,4,2—not a triangle)
Largest second	(4,6,2—not a triangle)
Largest third	(1,2,3—not a triangle)
Bad inputs:	
One bad input	(-1,2,4—bad inputs)
Two bad inputs	(3,-2,-5—bad inputs)
Three bad inputs	(0,0,0 – bad inputs)

10.3.3 TEST MATRICES

- A way to formalize this identification of subdomains is to build a matrix using the conditions that we can identify from the specification and then to systematically identify all combinations of these conditions as being true or false.

10.3 Test Coverage Criterion

EXAMPLE 10.2

The conditions in the triangle problem might be (1) $a = b$ or $a = c$ or $b = c$, (2) $a = b$ and $b = c$, (3) $a < b + c$ and $b < a + c$ and $c < a + b$, and (4) $a > 0$ and $b > 0$ and $c > 0$. These four conditions can be put on the rows of a matrix. The columns of the matrix will each condition is true and an F when the condition is false. All valid combinations of T and F will be used. If there are three conditions, there may be $2^3 = 8$ subdomains (columns). Additional rows will be used for values of a, b, and c and for the expected output for each subdomain.

Conditions	1	2	3	4	5	6	7	8
$a = b$ or $a = c$ or $b = c$	T	T	T	T	T	F	F	F
$a = b$ and $b = c$	T	T	F	F	F	F	F	F
$a =< b + c$ or $b =< a + c$ or $c =< a + b$	T	F	T	T	F	T	T	F
$a => 0$ or $b => 0$ or $c => 0$	T	F	T	F	F	T	F	F
Sample test case	0,0,0	3,3,3	0,4,0	3,8,3	5,8,5	0,5,6	3,4,8	3,4,5
Expected output	Bad inputs	Equilateral	Bad inputs	Not triangle	Isosceles	Bad inputs	Not triangle	Scalene

10.3.4 STRUCTURAL TESTING

- Structural testing is based on the structure of the source code. The simplest structural testing criterion is ***every statement coverage***, often called C0 coverage.

10.3 Test Coverage Criterion

10.3.4.1 C0—Every Statement Coverage

- This criterion is that every statement of the source code should be executed by some test case. The normal approach to achieving C0 coverage is to select test cases until a coverage tool indicates that all statements in the code have been executed.

EXAMPLE 10.3

The following pseudocode implements the triangle problem. The matrix shows which lines are executed by which test cases. Note that the first three statements (A, B, and C) can be considered parts of the same node.

By the fourth test case, every statement has been executed. This set of test cases is not the smallest set that would cover every statement. However, finding the smallest test set would often not find a good test set.

Node	Source Line	3,4,5	3,5,3	0,1,0	4,4,4
A	`read a,b,c`	*	*	*	*
B	`type=''scalene''`	*	*	*	*
C	`if(a==b||b==c||a==c)`	*	*	*	*
D	`type=''isosceles''`		*	*	*
E	`if(a==b&&b==c)`	*	*	*	*
F	`type=''equilateral''`				*
G	`if(a>=b+c||b>=a+c||c>=a+b)`	*	*	*	*
H	`type=''not a triangle''`			*	
I	`if(a<=0||b<=0||c<=0)`	*	*	*	*
J	`type=''bad inputs''`			*	
K	`print type`	*	*	*	*

10.3.4.2 C1—Every-Branch Testing

- A more thorough test criterion is ***every-branch testing***, which is often called C1 test coverage. In this criterion, the goal is to go both ways out of every decision.

EXAMPLE 10.4

If we model the program of Example 10.3 as a control flow graph (see Chapter 2), this coverage criterion requires covering every arc in the control flow diagram. See Fig. 10-1.

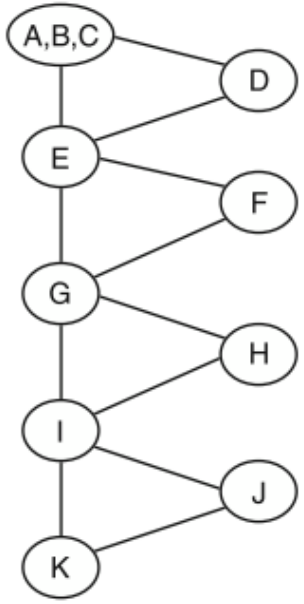

Fig. 10-1. Control flow graph for Example 10.3.

Arcs	3,4,5	3,5,3	0,1,0	4,4,4
ABC–D		*	*	*
ABC–E	*			
D–E		*	*	*
E–F				*
E–G	*	*	*	
F–G				*
G–H			*	
G–I	*	*		*
H–I			*	
I–J			*	
I–K	*	*		*
J–K			*	

10.3.4.3 Every-Path Testing

- A more thorough test criterion is the every-path testing criterion. A path is a unique sequence of program nodes that are executed by a test case. In the testing matrix (Example 10.2) above, there were eight subdomains. Each of these just happens to be a path. In that example, there are sixteen different combinations of T and F. However, eight of those combinations are infeasible paths. That is, there is no test case that could have that combination of T and F for the decisions in the program. It can be exceedingly hard to determine if a path is infeasible or if it is just hard to find a test case that executes that path.
- Most programs with loops will have an infinite number of paths. In general, every-path testing is not reasonable.

10.3 Test Coverage Criterion

EXAMPLE 10.5

The following table shows the eight feasible paths in the triangle pseudocode from Example 10.3.

Path	T/F	Test Case	Output
ABCEGIK	FFFF	3,4,5	Scalene
ABCEGHIK	FFTF	3,4,8	Not a triangle
ABCEGHIJK	FFTT	0,5,6	Bad inputs
ABCDEGIK	TFFF	5,8,5	Isosceles
ABCDEGHIK	TFTF	3,8,3	Not a triangle
ABCDEGHIJK	TFTT	0,4,0	Bad inputs
ABCDEFGIK	TTFF	3,3,3	Equilateral
ABCDEFGHIJK	TTTT	0,0,0	Bad inputs

10.3.4.4 Multiple-Condition Coverage

- A multiple-condition testing criterion requires that each primitive relation condition is evaluated both true and false. Additionally, all combinations of T/F for the primitive relations in a condition must be tried. Note that lazy evaluation of expressions will eliminate some combinations. For example, in an ‘‘and’’ of two primitive relations, the second will not be evaluated if the first one is false.

10.3 Test Coverage Criterion

EXAMPLE 10.6

In the pseudocode in Example 10.3, there are multiple conditions in each decision statement. Primitives that are not executed because of lazy evaluation are shown with an "X".

```
if(a==b||b==c||a==c)
```

Combination	Possible Test Case	Branch
TXX	3,3,4	ABC-D
FTX	4,3,3	ABC-D
FFT	3,4,3	ABC-D
FFF	3,4,5	ABC-E

```
if(a==b&&b==c)
```

Combination	Possible Test Case	Branch
TT	3,3,3	E-F
TF	3,3,4	E-G
FX	4,3,3	E-G

```
if(a>=b+c||b>=a+c||c>=a+b)
```

Combination	Possible Test Case	Branch
TXX	8,4,3	G-H
FTX	4,8,3	G-H
FFT	4,3,8	G-H
FFF	3,3,3	G-I

```
if(a<=0||b<=0||c<=0)
```

Combination	Possible Test Case	Branch
TXX	0,4,5	I-J
FTX	4,−2,−2	I-J
FFT	5,4,−3	I-J
FFF	3,3,3	I-K

10.3.4.5 Subdomain Testing

- ***Subdomain testing*** is the idea of partitioning the input domain into mutually exclusive subdomains and requiring an equal number of test cases from each subdomain. This was basically the idea behind the test matrix. Subdomain testing is more general in that it does not restrict how the subdomains are selected. Generally, if there is a good reason for picking the subdomains, then they may be useful for testing. Additionally, the subdomains from other approaches might be subdivided into smaller subdomains. Theoretical work has shown that subdividing subdomains is only effective if it tends to isolate potential errors into individual subdomains.
- Every-statement coverage and every-branch coverage are not subdomain tests. There are not mutually exclusive subdomains related to the execution of different statements or branches. Every-path coverage is a subdomain coverage, since the subdomain of test cases that execute a particular path through a program is mutually exclusive with the subdomain for any other path.

10.3 Test Coverage Criterion

EXAMPLE 10.7

For the triangle problem, we might start with a subdomain for each output. These might be further subdivided into new subdomains based on whether the largest or the bad element is in the first position, second position, or third position (when appropriate).

Subdomain	Possible Test Case
Equilateral	3,3,3
Isos – first	8,5,5
Isos – sec	5,8,5
Isos – third	5,5,8
Scalene – first	5,4,3
Scalene – sec	4,5,3
Scalene – third	3,4,5

Subdomain	Possible Test Case
Not triangle – first	8,3,3
Not triangle – sec	3,8,4
Not triangle – third	4,3,8
Bad input – first	0,3,4
Bad input – sec	3,0,4
Bad input – third	3,4,0

10.3.4.6 C1 Subsumes C0

EXAMPLE 10.8—C1 Subsumes C0

For the triangle problem, in Example 10.3 we selected good test cases until we achieved the C0 coverage. The test cases were (3,4,5—scalene), (3,5,3— isosceles), (0,1,0—bad inputs), and (4,4,4—equilateral). These tests also covered four out the five possible outputs. However, we can achieve C1 coverage with two test cases: (3,4,5—scalene) and (0,0,0—bad inputs). This test is probably not as good as the first test set. However, it achieves C1 coverage and it also achieves C0 coverage.

10.4 Data Flow Testing

- ***Data flow testing*** is testing based on the flow of data through a program. Data flows from where it is defined to where it is used. A ***definition of data***, or ***def***, is when a value is assigned to a variable. Two different kinds of use have been identified. The ***computation use,*** or ***c-use***, is when the variable appears on the right-hand side of an assignment statement. A c-use is said to occur on the assignment statement. The ***predicate use***, or ***p-use***, is when the variable is used in the condition of a decision statement. A p-use is assigned to both branches out of the decision statement. A ***definition free path***, or ***def-free***, is a path from a definition of a variable to a use of that variable that does not include another definition of the variable.

EXAMPLE 10.9—Control Flow Graph of Triangle Problem (Example 10.3)

The control flow graph in Fig. 10-2 is annotated with the definitions and uses of the variables a, b, and c.

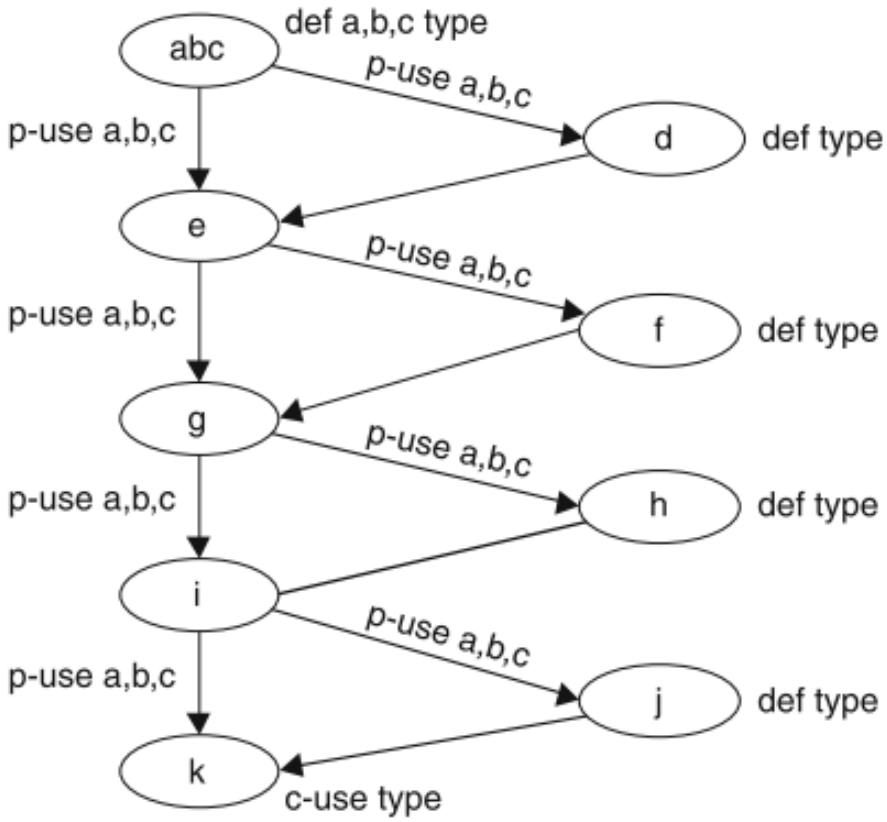

Fig 10-2. Control flow graph of triangle problem.

10.4 Data Flow Testing

- There are many data flow testing criteria. The basic criteria include ***dcu***, which requires a def-free path from every definition to a c-use; ***dpu***, which requires a def-free path from every definition to a p-use; and ***du***, which requires a def-free path from every definition to every possible use. The most extensive criteria is ***all-du-paths***, which requires all def-free paths from every definition to every possible use.

10.4 Data Flow Testing

EXAMPLE 10.10—Data Flow Testing of Triangle Problem

dcu—The only c-use is for variable type in node k (the output statement).

From def type in node abc to node k	Path abc,e,g,i,k
From def type in node d to node k	Path d,e,g,i,k
From def type in node f to node k	Path f,g,i,k
From def type in node h to node k	Path h,i,k
From def type in node j to node k	Path j,k

dpu—The only p-use is for variables a,b,c and the only def of a,b,c is node abc.

From node abc to arc abc-d
From node abc to arc abc-e
From node abc to arc e-f
From node abc to arc e-g
From node abc to arc g-h
From node abc to arc g-i
From node abc to arc i-j
From node abc to arc i-k

du—All defs to all uses.

All test cases of dcu and dpu combined.

all-du-paths—All def-free paths from all defs to all uses.

Same as du tests.

10.5 Random Testing

- ***Random testing*** is accomplished by randomly selecting the test cases. This approach has the advantage of being fast and it also eliminates biases of the testers. Additionally, statistical inference is easier when the tests are selected randomly. Often the tests are selected randomly from an operational profile.

EXAMPLE 10.11

For the triangle problem, we could use a random number generator and group each successive set of three numbers as a test set. We would have the additional work of determining the expected output. One problem with this is that the chance of ever generating an equilateral test case would be verysmall. If it actually happened, we would probablystart questioning our pseudorandom number generator.

10.5 Random Testing

10.5.1 OPERATIONAL PROFILE

- Testing in the development environment is often very different than execution in the operational environment. One way to make these two more similar is to have a specification of the types and the probabilities that those types will be encountered in the normal operations. This specification is called an ***operational profile***. By drawing the test cases from the operational profile, the tester will have more confidence that the behavior of the program during testing is more predictive of how it will behave during operation.

10.5 Random Testing

EXAMPLE 10.12

A possible operational profile for the triangle problem is as follows:

#	Description	Probability
1	equilateral	.20
2	isosceles – obtuse	.10
3	isosceles – right	.20
4	scalene – right triangle	.10
5	scalene – all acute	.25
6	scalene – obtuse angle	.15

To apply random testing, the tester might generate a number to select the category by probabilities and then sufficient additional numbers to create the test case. If the category selected was the equilateral case, the tester would use the same number for all three inputs. An isosceles–right would require a random number for the length of the two sides, and then the use of trigonometry to calculate the other side.

10.5.2 STATISTICAL INFERENCE FROM TESTING

- If random testing has been done by randomly selecting test cases from an operational profile, then the behavior of the software during testing should be the same as its behavior in the operational environment.

EXAMPLE 10.13

If we selected 1000 test cases randomly using an operational profile and found three errors, we could predict that this software would have an error rate of less than three failures per 1000 executions in the operational environment. See Section 3.8 for more information on using error rates.

10.6 Boundary Testing

- Often errors happen at boundaries between domains. In source code, decision statements determine the boundaries. If a decision statement is written as `x<1` instead of `x<0`, the boundary has shifted. If a decision is written `x=<1`, then the boundary, `x=1`, is in the true subdomain. In the terminology of boundary testing, we say that the on tests are in the true domain and the off tests are values of x greater than 1 and are in the false domain.
- If a decision is written x<1 instead of x=<1, then the boundary, x=1, is now in the false subdomain instead of in the true subdomain.
- Boundary testing is aimed at ensuring that the actual boundary between two subdomains is as close as possible to the specified boundary. Thus, test cases are selected on the boundary and off the boundary as close as reasonable to the boundary. The standard boundary test is to do two on tests as far apart as possible and one off test close to the middle of the boundary.
- Figure 10-3 shows a simple boundary. The arrow indicates that the on tests of the boundary are in the subdomain below the boundary. The two on tests are at the ends of the boundary and the off test is just above the boundary halfway along the boundary.

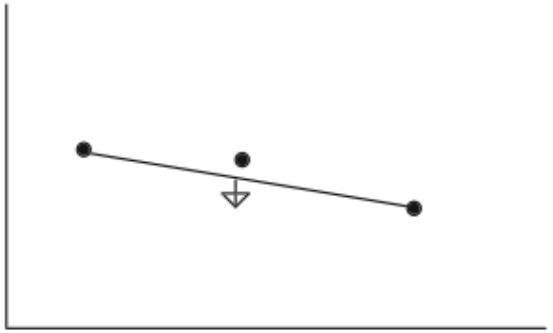

Fig. 10-3. Boundary conditions.

10.6 Boundary Testing

EXAMPLE 10.14

In the triangle example, the primitive conditions, a => b + c or b=> a + c or c=> a +b, determine a boundary. Since these are in three variables, the boundary is actually a plane in 3D space. The on tests would be two (or more) widely separated tests that have equality—for example, (8,1,7) and (8,7,1). These are both true. The off test would be in the other domain (false) and would be near the middle—for example, (7.9, 4,4).