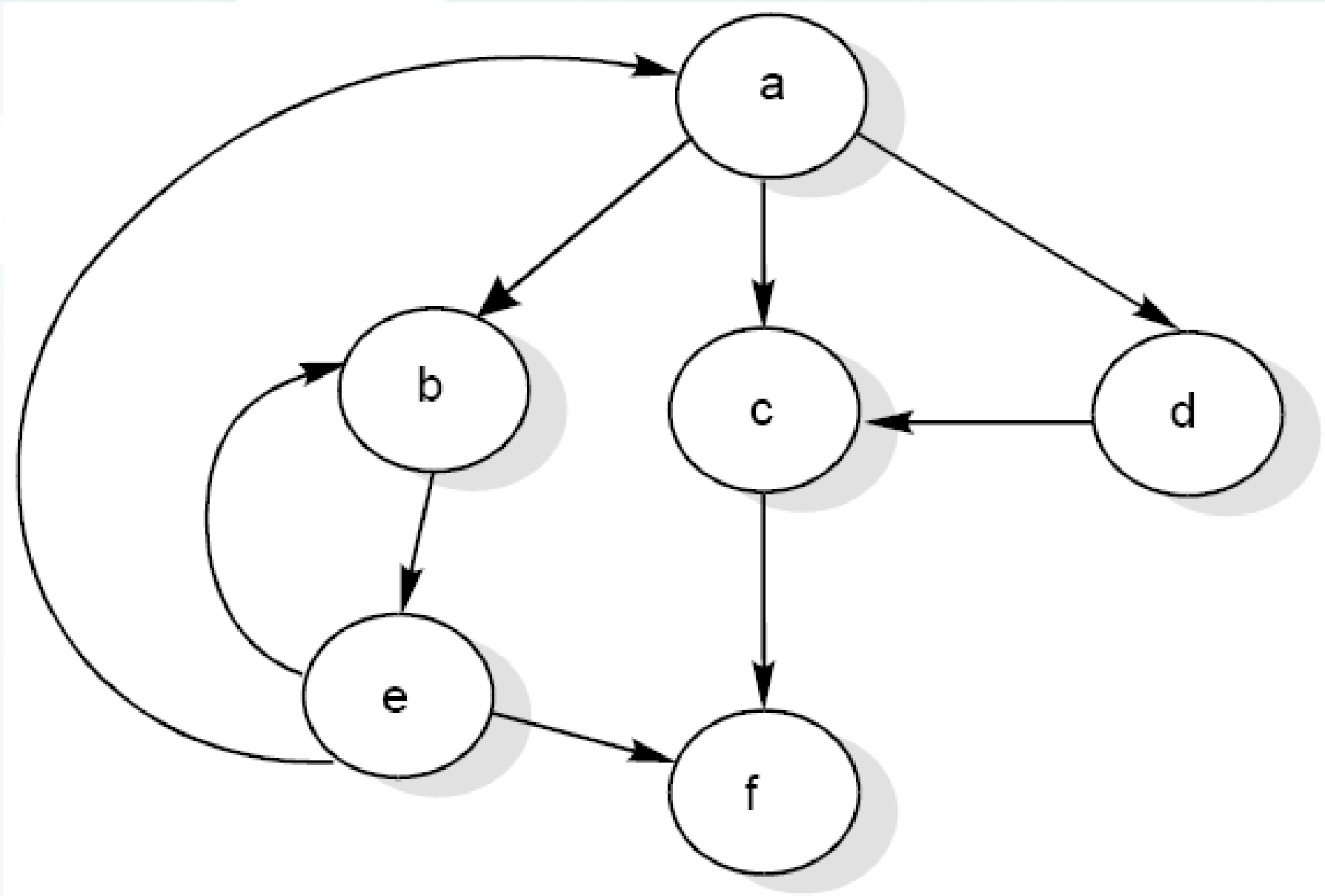
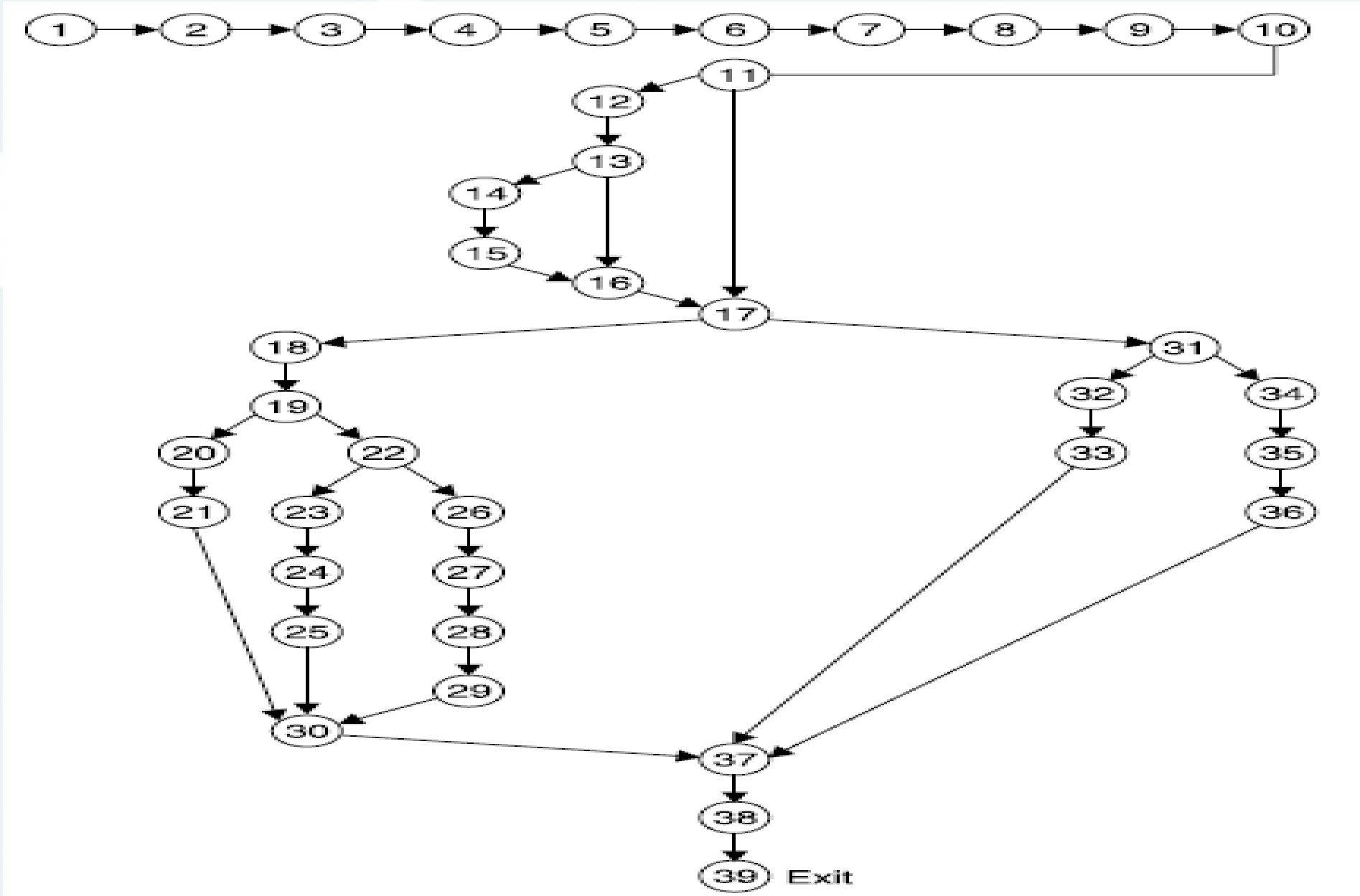
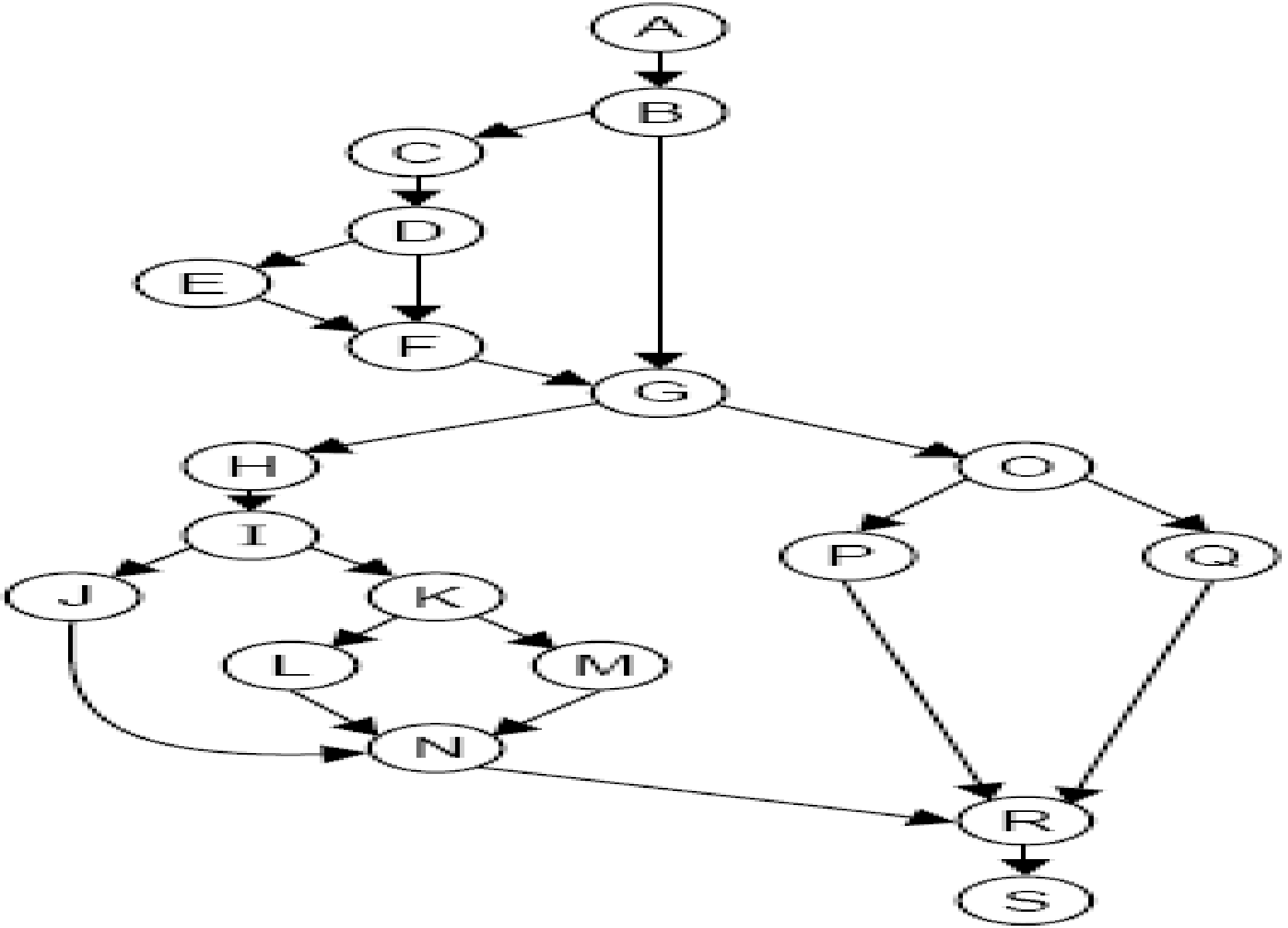

a
b
c
d
e
f



Program flow graph



Dd path graph

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14,15	E	Sequential node
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20,21	J	Sequential node
22	K	Decision node
23,24,25	L	Sequential node

Flow graph nodes	DD Path graph corresponding node	Remarks
26,27,28,29	M	Sequential nodes
30	N	Three edges are combined
31	O	Decision node
32,33	P	Sequential node
34,35,36	Q	Sequential node
37	R	Three edges are combined here
38,39	S	Sequential nodes with exit node

Independent paths are:

(i) ABGOQRS

(ii) ABGOPRS

(iii) ABCDFGOQRS

(iv) ABCDEFGOPRS

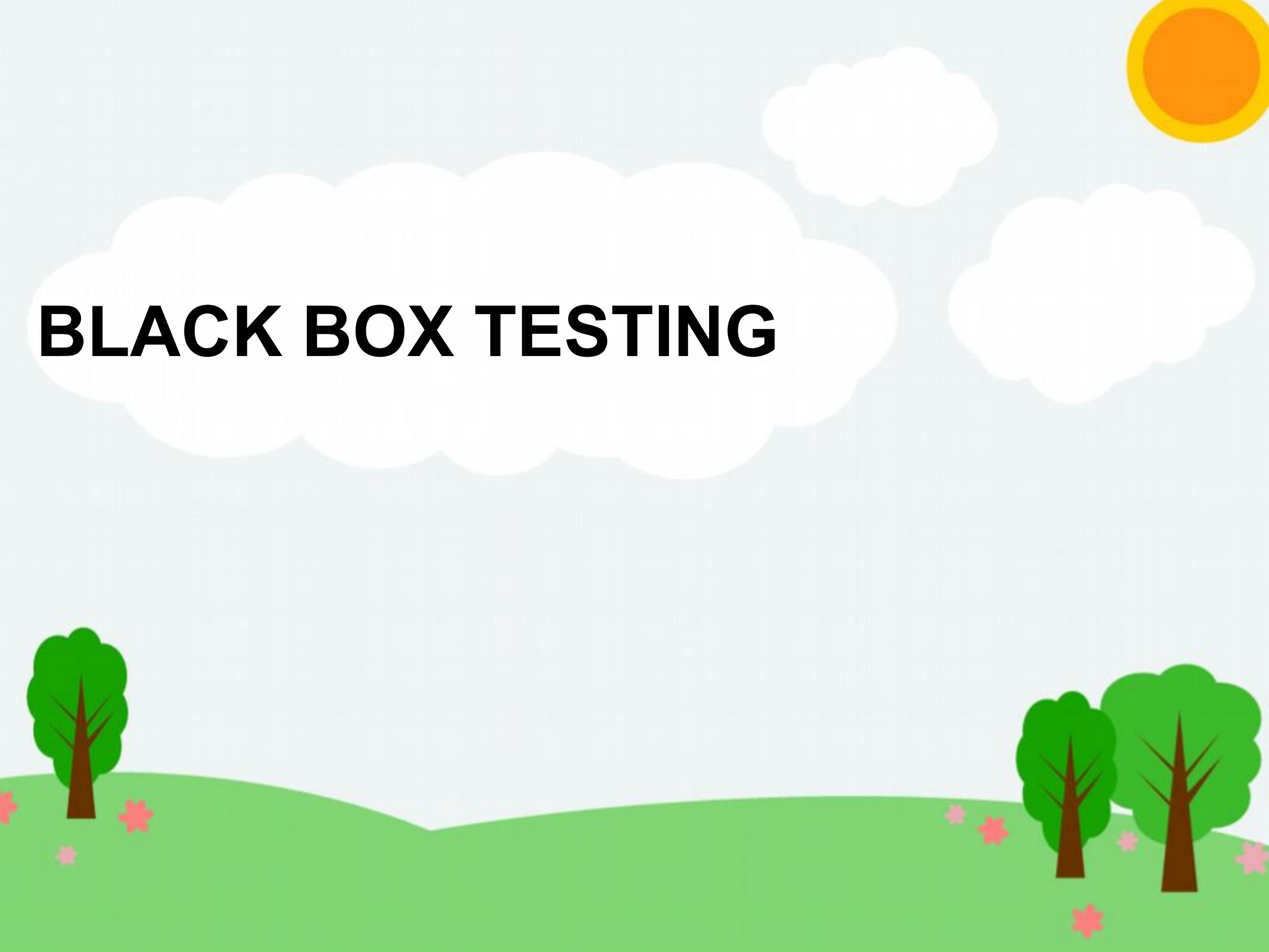
(v) ABGHIJNRS

(vi) ABGHIKLNRS

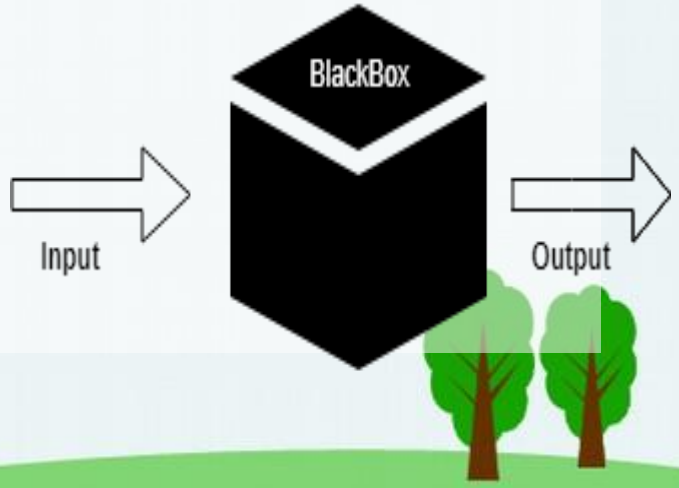
(vi) ABGHIKMNRS

- Number of nodes (n) = 19
- Number of edges (e) = 24
- (i) V(G) = e – n + 2 = 24 – 19 + 2 = 7
- (ii) V(G) = p + 1 = 6 + 1 = 7
- (iii) V(G) = Number of regions = 7
- Hence cyclomatic complexity is 7 meaning thereby, seven independent paths in the DD Path graph.

BLACK BOX TESTING



- Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.
- In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



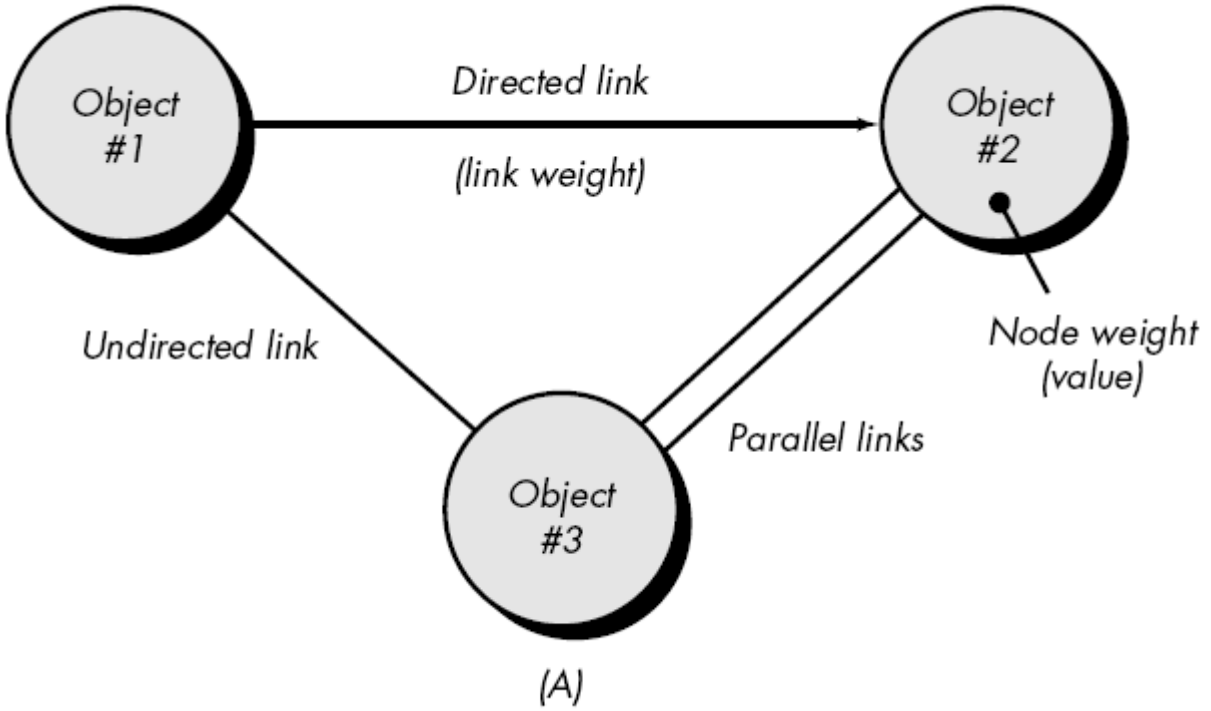
- Black-box testing attempts to find errors in the following categories:
- (1) incorrect or missing functions,
- (2) interface errors,
- (3) errors in data structures or external data base access,
- (4) behavior or performance errors, and
- (5) initialization and termination errors.

- Unlike white-box testing, which is performed early in the testing process, blackbox testing tends to be applied during later stages of testing. Tests are designed to answer the following questions:
- How is functional validity tested?
- How is system behavior and performance 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?

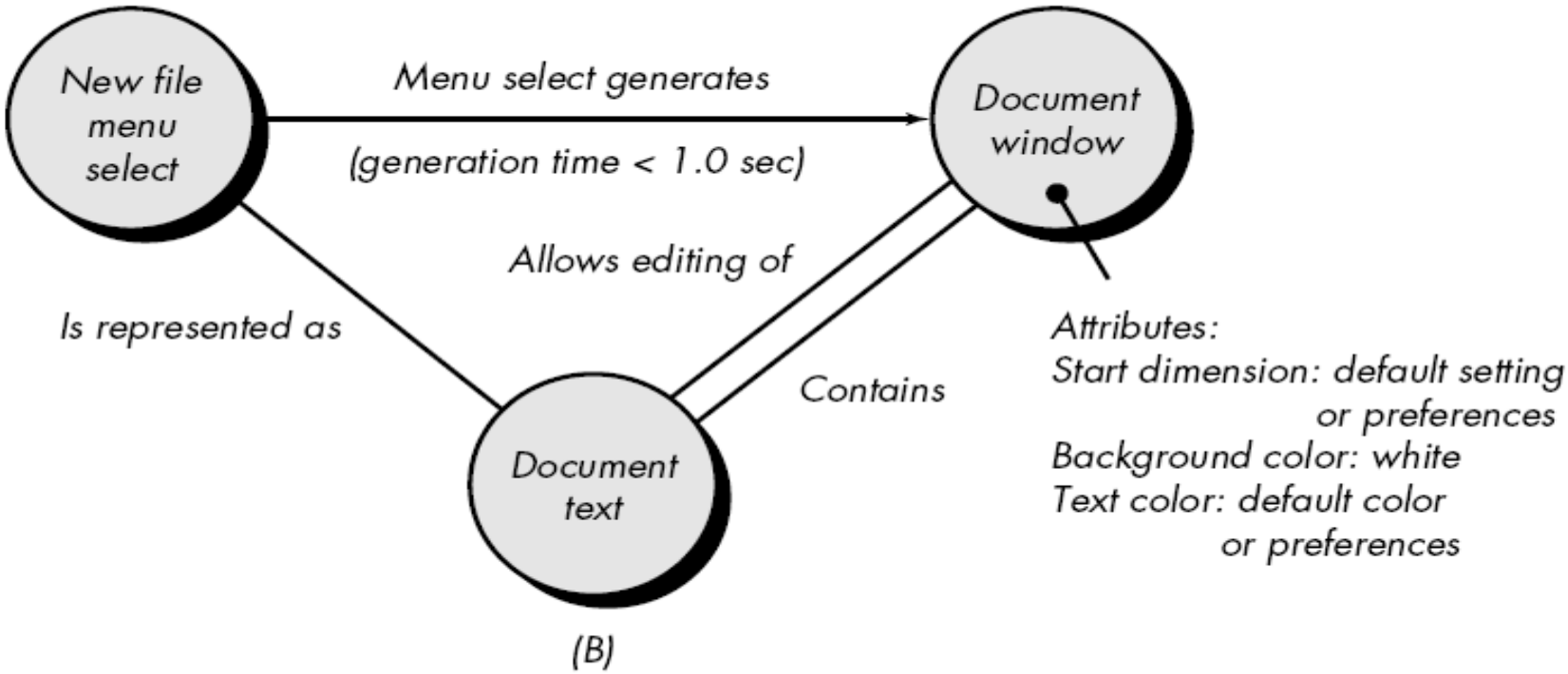
Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects that are modeled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify “all objects have the expected relationship to one another.” Stated in another way, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- To accomplish these steps, the software engineer begins by creating a graph—a collection of nodes that represent objects; links that represent the relationships between objects; node weights that describe the properties of a node (e.g., a specific data value or state behavior); and link weights that describe some characteristic of a link.

- The symbolic representation of a graph is shown in figure. Nodes are represented as circles connected by links that take a number of different forms. A directed link (represented by an arrow) indicates that a relationship moves in only one direction.



- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions. Parallel links are used when a number of different relationships are established between graph nodes.
- As a simple example, consider a portion of a graph for a word-processing application where
- Object #1 = new file menu select
- Object #2 = document window
- Object #3 = document text



- Referring to the above figure, a menu select on new file generates a document window. The node weight of document window provides a list of the window attributes that are to be expected when the window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the new file menu select and document text, and parallel links indicate relationships between document window and document text. In reality, a far more detailed graph would have to be generated as a precursor to test case design. The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

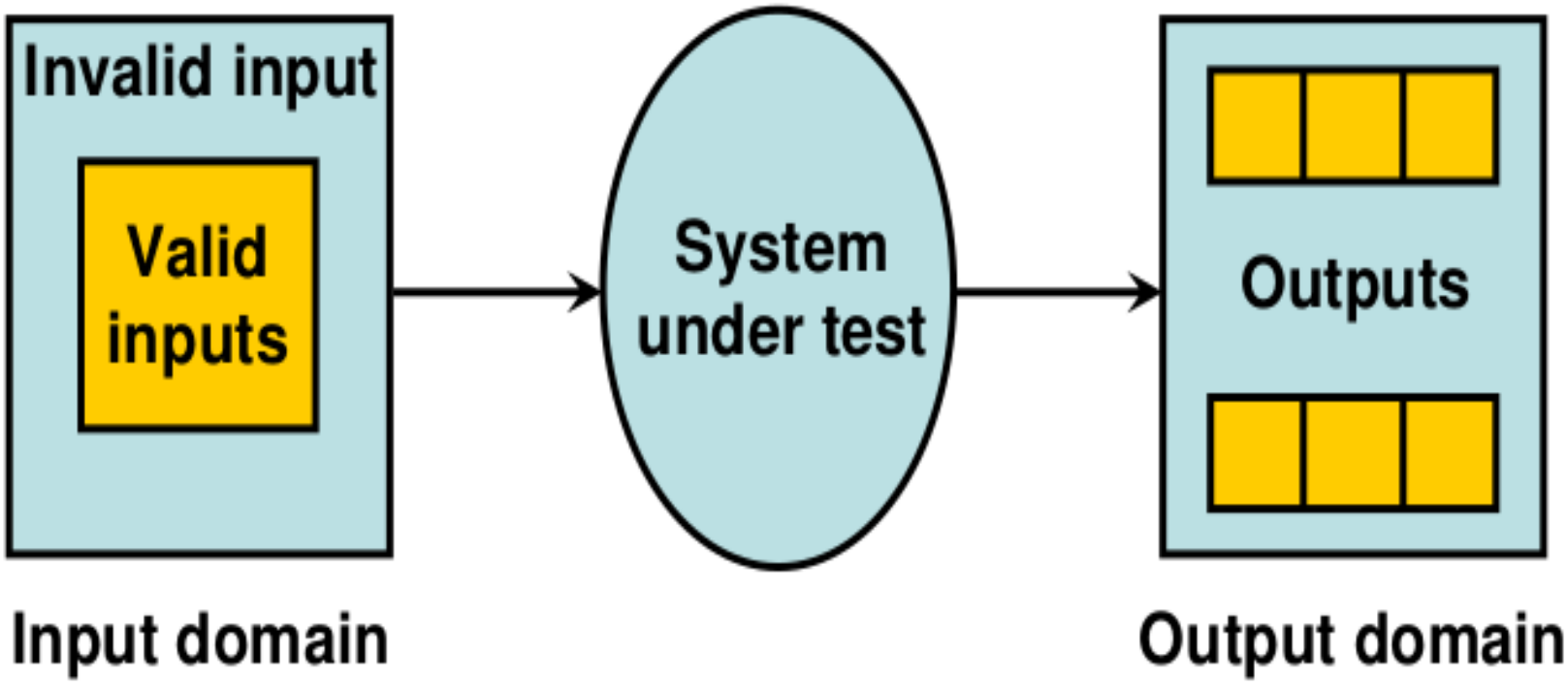
- number of behavioral testing methods that can make use of graphs:
- **Transaction flow modeling.** The nodes represent steps in some transaction (e.g., the steps required to make an airline reservation using an on-line service), and the links represent the logical connection between steps e.g., **flightinformationinput** is followed by **validationavailabilityprocessing()**. The data flow diagram can be used to assist in creating graphs of this type.
- **Finite state modeling.** The nodes represent different user observable states of the software (e.g., each of the "screens" that appear as an order entry clerk takes a phone order), and the links represent the transitions that occur to move from state to state (e.g., **order-information** is verified during inventory- **availabilitylook-up()** and is followed by **customerbilling-information input()**. The state transition diagram can be used to assist in creating graphs of this type.
- **Data flow modeling**. The nodes are data objects and the links are the transformations that occur to translate one data object into another. For example, the node FICA.tax.withheld (FTW) is computed from gross.wages (GW) using the relationship, FTW = 0.62 GW.
- **Timing modeling.** The nodes are program objects and the links are the sequential connections between those objects. Link weights are used to specify the required execution times as the program executes.

- Once nodes have been identified, links and link weights should be established. In general, links should be named, although links that represent control flow between program objects need not be named.
- In many cases, the graph model may have loops (i.e., a path through the graph in which one or more nodes is encountered more than one time). Loop testing can also be applied at the behavioral (black-box) level. The graph will assist in identifying those loops that need to be tested.
- Each relationship is studied separately so that test cases can be derived. The transitivity of sequential relationships is studied to determine how the impact of relationships propagates across objects defined in a graph. Transitivity can be illustrated by considering three objects, X, Y, and Z. Consider the following relationships:
- X is required to compute Y
- Y is required to compute Z
- Therefore, a transitive relationship has been established between X and Z:
- X is required to compute Z
- Based on this transitive relationship, tests to find errors in the calculation of Z must consider a variety of values for both X and Y.

- The symmetry of a relationship (graph link) is also an important guide to the design of test cases. If a link is indeed bidirectional (symmetric), it is important to test this feature. The UNDO feature in many personal computer applications implements limited symmetry. That is, UNDO allows an action to be negated after it has been completed. This should be thoroughly tested and all exceptions (i.e., places where UNDO cannot be used) should be noted. Finally, every node in the graph should have a relationship that leads back to itself; in essence, a “no action” or “null action” loop. These reflexive relationships should also be tested.
- As test case design begins, the first objective is to achieve node coverage. By this we mean that tests should be designed to demonstrate that no nodes have been inadvertently omitted and that node weights (object attributes) are correct.
- Next, link coverage is addressed. Each relationship is tested based on its properties. For example, a symmetric relationship is tested to demonstrate that it is, in fact, bidirectional. A transitive relationship is tested to demonstrate that transitivity is present. A reflexive relationship is tested to ensure that a null loop is present. When link weights have been specified, tests are devised to demonstrate that these weights are valid. Finally, loop testing is invoked .

Equivalence Class Testing

- In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.
- Two steps are required to implementing this method:
- 1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes[item<1] and [item>999].
- 2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.



Most of the time, equivalence class testing defines classes of the input domain.
However, equivalence classes should also be defined for output domain.
Hence, we should design equivalence classes based on input and output domain.

- Example

Consider the program for the determination of nature of roots of a quadratic equation as explained earlier Identify the equivalence class test cases for output and input domains.

- Output domain equivalence class test cases can be identified as follows:
- O 1 ={<a,b,c>:Not a quadratic equation if a = 0}
- O 1 ={<a,b,c>:Real roots if (b^2 -4ac)>0}
- O 1 ={<a,b,c>:Imaginary roots if (b^2 -4ac)<0}
- O 1 ={<a,b,c>:Equal roots if (b^2 -4ac)=0}`

Test case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

- We may have another set of test cases based on input domain.
- I 1 = {a: a = 0}
- I 2 = {a: a < 0}
- I 3 = {a: 1 ≤ a ≤ 100}
- I 4 = {a: a > 100}
- I 5 = {b: 0 ≤ b ≤ 100}
- I 6 = {b: b < 0}
- I 7 = {b: b > 100}
- I 8 = {c: 0 ≤ c ≤ 100}
- I 9 = {c: c < 0}
- I 10 ={c: c > 100}

Test Case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Boundary Value Analysis

It is generally observed that greater no of errors occur at the boundaries of the input domain rather than at the center. This technique leads to the development of the test cases that exercise the boundary values. Boundary value analysis is a test case design technique that complements equivalence partitioning.

Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines

a. If the input condition specifies a range bounded by values a & b, then test cases must be designed with inputs as a & b, just above and below a & b.

b. Similarly if the input condition specifies a no of values, then test cases must be developed that exercise the minimum, maximum values & also the values just above and below the maximum and the minimum.

c. Apply the above guidelines also to the output i.e. design test cases such that you get the minimum and the maximum values of the outputs.

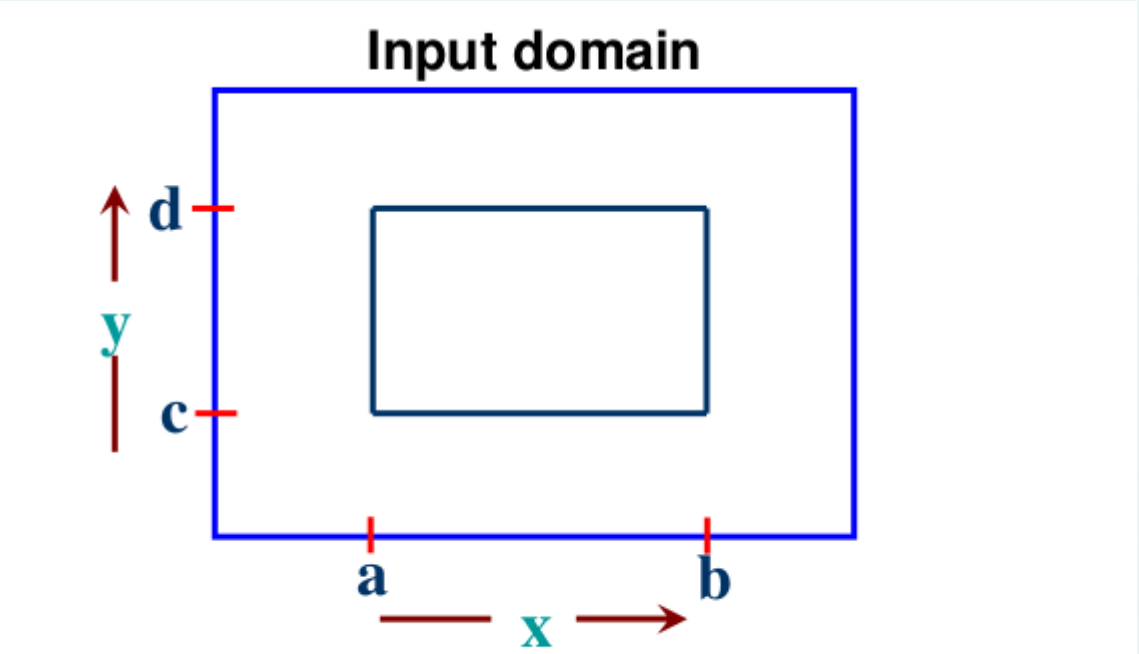
d. If the internal data structures have prescribed boundaries, then test the data structures at their boundaries (array limit of 100).

Consider a program with two input variables x and y. These input variables

have specified boundaries as:

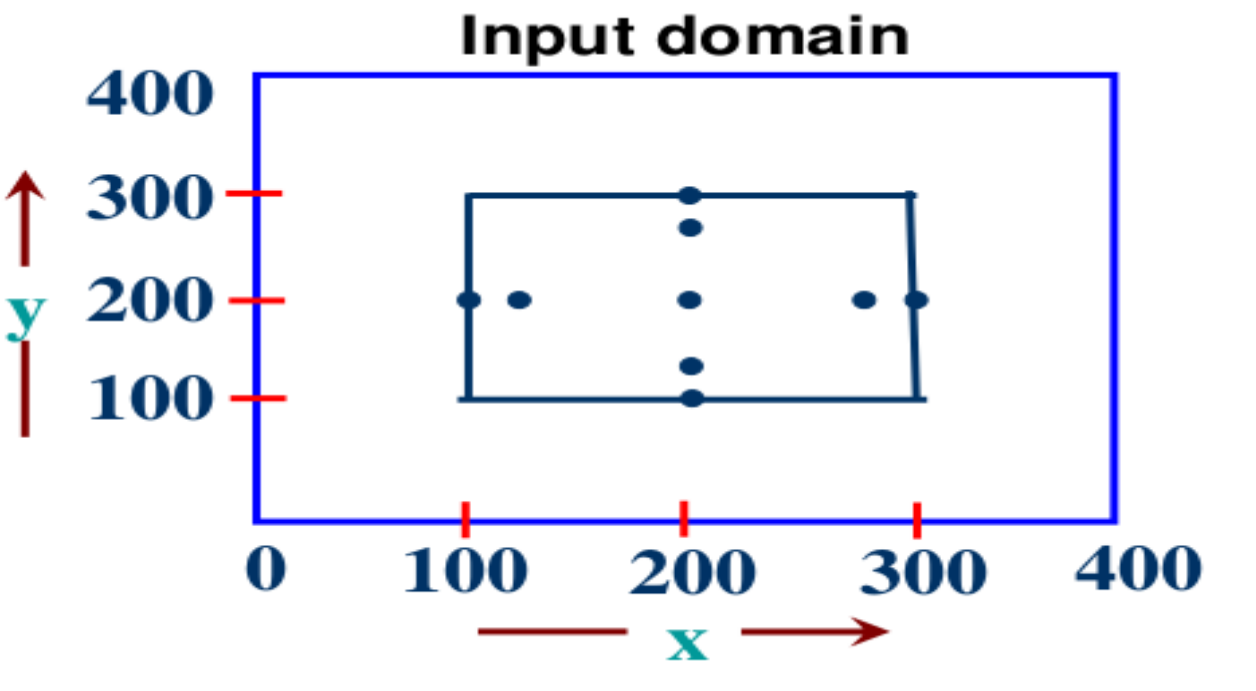
$a \leq x \leq b$

$c \leq y \leq d$



Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100),(200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig.. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield 4n + 1 test cases.



Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Quadratic equation will be of type:

ax 2 +bx+c=0

Roots are real if (b 2 -4ac)>0

Roots are imaginary if (b 2 -4ac)<0

Roots are equal if (b 2 -4ac)=0

Equation is not quadratic if a=0

The boundary value test cases are :

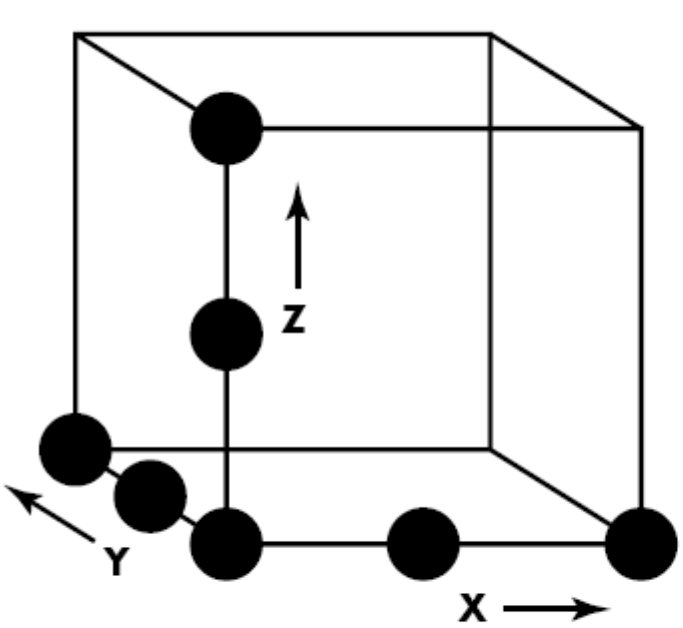
Test Case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Orthogonal array testing

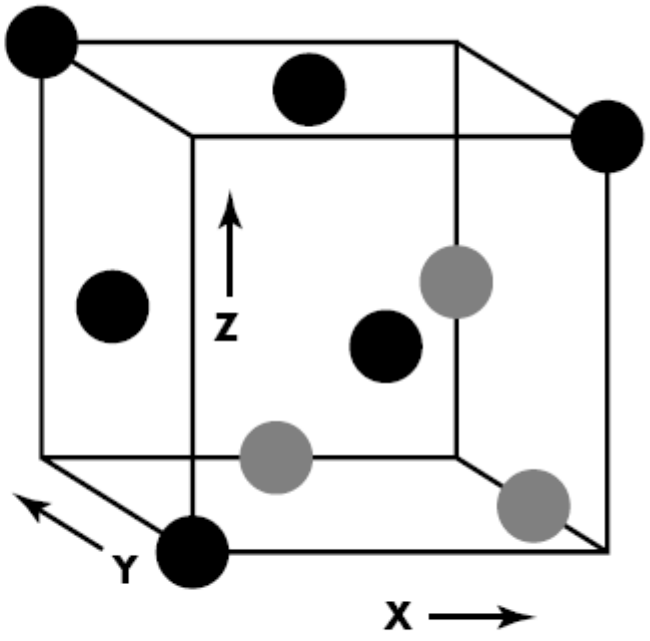
There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible.

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding errors associated with region faults—an error category associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional “one input item at a time” approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are 3^3 = 27 possible test cases. one of the suggestion geometric view of the possible test cases associated with X, Y, and Z illustrated in figure. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure



One input item at a time



L9 orthogonal array

When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a “balancing property .” That is, test cases (represented by black dots in the figure) are “dispersed uniformly throughout the test domain,” as illustrated in the right-hand cube in Figure. Test coverage across the input domain is more complete.

To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now

P1 = 2, send it one hour later

P1 = 3, send it after midnight

P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

If a “one input item at a time” testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3).Assesses these test cases in the following manner:

Such test cases are useful only when one is certain that these test parameters do not interact. They can detect logic faults where a single parameter value makes the software malfunction. These faults are called single mode faults. This method cannot detect logic faults that cause malfunction when two or more parameters simultaneously take certain values; that is, it cannot detect any interactions. Thus its ability to detect faults is limited. Given the relatively small number of input parameters and discrete values, exhaustive testing is possible. The number of tests required is 3^4 = 81, large, but manageable. All faults associated with data item permutation would be found, but the effort required is relatively high.

The orthogonal array testing approach enables us to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in figure.

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with “send it now” (P1 = 1)] as the source of the error. Such an isolation

assesses the result of tests using the L9 orthogonal array in the following manner:

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 = 1 cause an error condition, it is a single mode failure. In this example tests 1, 2 and 3 will show errors. By analyzing the information about which tests show errors, one can identify which parameter values cause the fault. In this example, by noting that tests 1, 2, and 3 cause an error, one can isolate [logical processing associated with “send it now” (P1 = 1)] as the source of the error. Such an isolation of fault is important to fix the fault.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault. Indeed, a double mode fault is an indication of pairwise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays [of the type shown] can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

Comparison testing

Here are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical.In such applications redundant hardware and software are often used to minimize the possibility of error. When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification.

In such situations,each version can be tested with the same test data to ensure that all provide identical output. Then all versions are executed in parallel with real-time comparison of results to ensure consistency.

Using lessons learned from redundant systems, researchers have suggested that independent versions of software be developed for critical applications, even when only a single version will be used in the delivered computer-based system. These independent versions form the basis of a black-box testing technique called comparison testing or back-to-back testing.

When multiple implementations of the same specification have been produced, test cases designed using other black-box techniques (e.g., equivalence partitioning) are provided as input to each version of the software. If the output from each version is the same, it is assumed that all implementations are correct. If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference. In most cases, the comparison of outputs can be performed by an automated tool.

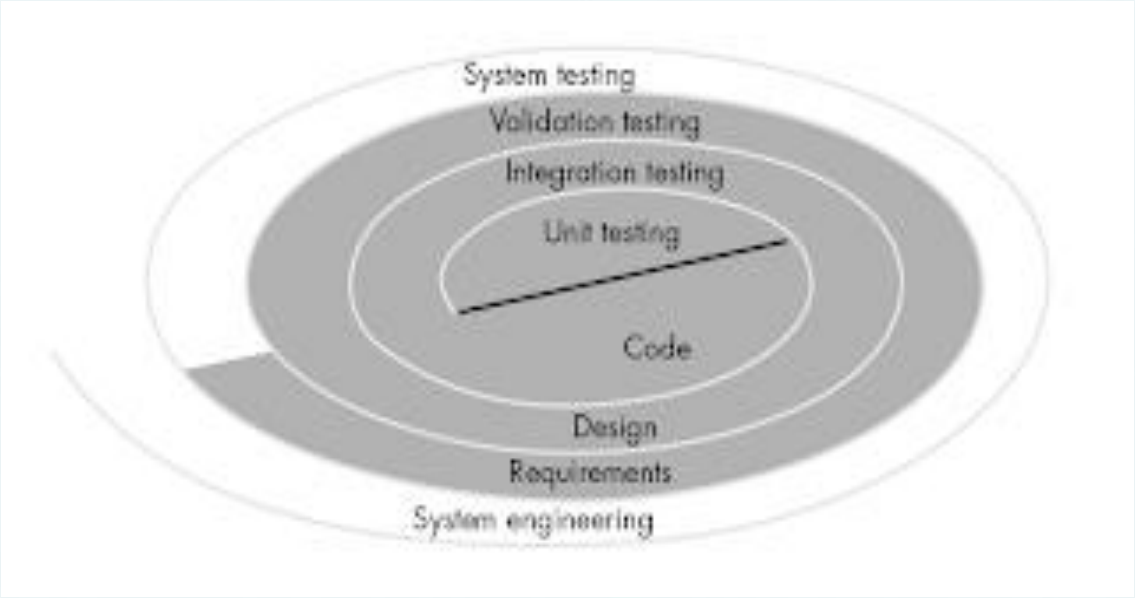
Comparison testing is not foolproof. If the specification from which all versions have been developed is in error, all versions will likely reflect the error. In addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

Software testing strategies

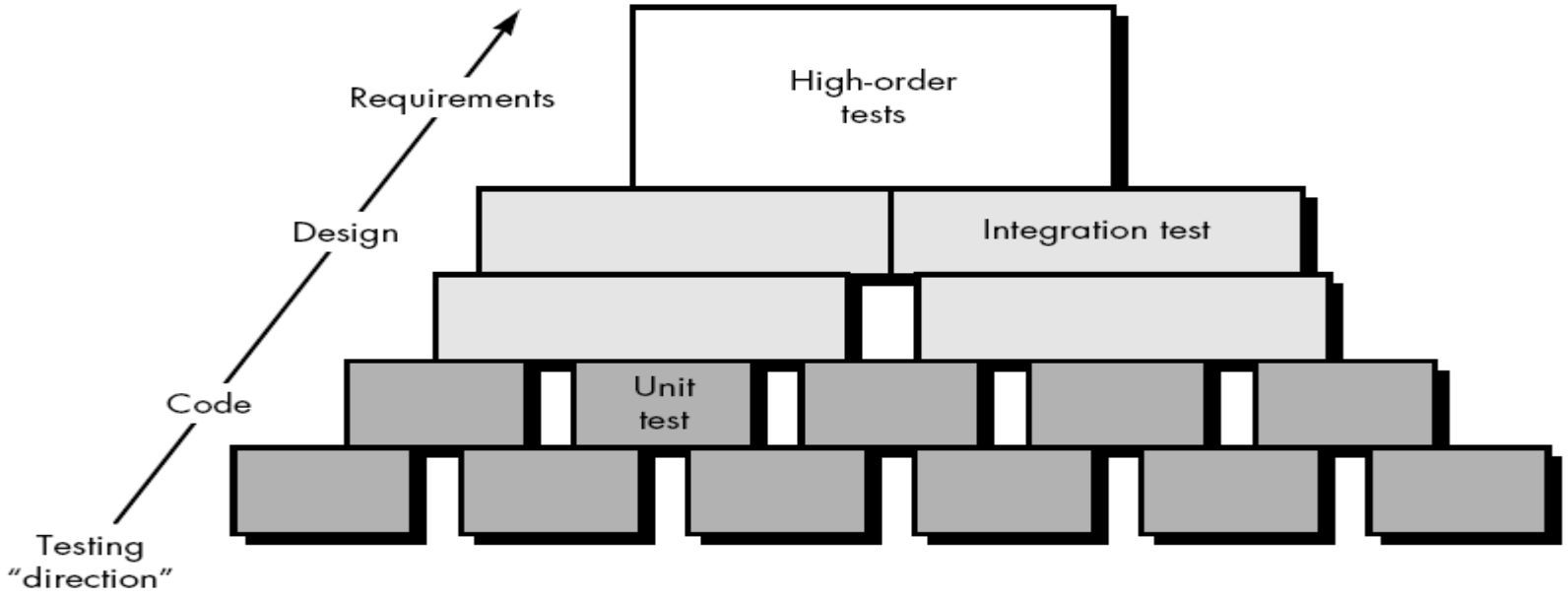
A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software. The strategy provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required. Therefore, any testing strategy must incorporate test planning, test case design, test execution, and resultant data collection and evaluation.

A software testing strategy should be flexible enough to promote a customized testing approach. At the same time, it must be rigid enough to promote reasonable planning and management tracking as the project progresses

the software engineering process may be viewed as the spiral illustrated in the figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, we come to design and finally to coding. To develop computer software, we spiral inward along streamlines that decrease the level of abstraction on each turn.



A strategy for software testing may also be viewed in the context of the spiral . Unit testing begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed. Finally, we arrive at system testing, where the software and other system elements are tested as a whole. To test computer software, we spiral out along streamlines that broaden the scope of testing with each turn.



Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of four steps that are implemented sequentially. The steps are shown in figure.

Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name unit testing. Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete software package. Integration testing addresses the issues associated with the dual problems of verification and program construction.

Black-box test case design techniques are the most prevalent during integration, although a limited amount of white-box testing may be used to ensure coverage of major control paths.

After the software has been integrated (constructed), a set of high-order tests are conducted. Validation criteria (established during requirements analysis) must be tested. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements. Black-box testing techniques are used exclusively during validation.

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.

Verification and validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

> Verification: "Are we building the product right?"
>
> Validation: "Are we building the right product?"

The definition of V&V encompasses many of the activities that we have referred to as software quality assurance (SQA).

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing . Although testing plays an extremely important role in V&V, many other activities are also necessary.