Chapter 10 *Software Testing*

This chapter will discuss the following concepts:

10.1 Software Testing Fundamentals

During earlier software engineering activities, the engineer attempts to build software from an abstract concept to an actual product. Now comes testing. The engineer creates a series of test cases to test the software that has been built.

10.1.1 Testing Objectives

Testing objectives are:

1- Testing is a process of executing a program with the intent of finding an error.
2- A good test case is one that has a high probability of finding an as-yet undiscovered error.
3- A successful test is one that uncovers an as-yet-undiscovered error.

10.1.2 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Here is a set of testing principles that have been adapted for use in software testing:

1- All tests should be traceable to customer requirements.
2- Tests should be planned long before testing begins.
3- Testing should begin “in the small” and progress toward testing “in the large”.
4- Exhaustive testing is not possible.
5- To be most effective, testing should be conducted by an independent third party.

10.1.3 Testability

In ideal circumstances, a software engineer designs a computer program, a system, or a product with "testability" in mind. This enables the individuals charged with testing to design effective test cases more easily. Software testability is simply how easily a computer program can be tested.

10.2 White-Box Testing

White-box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that:

1- Guarantee that all independent paths within a module have been exercised at least once.
2- Exercise all logical decisions on their true and false sides.
3- Execute all loops at their boundaries and within their operational bounds.
4- Exercise internal data structures to ensure their validity.

Black-box testing, no matter how thorough, may miss the kinds of errors noted here while White-box testing is far more likely to uncover them.

10.3 Basis Path Testing

Basis path testing is a white-box testing technique that enables the test case designer to derive a logical complexity measure of a procedural

design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

10.3.1 Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 10.1. Where each circle represents one or more non-branching program design language (PDL) or source code statements.

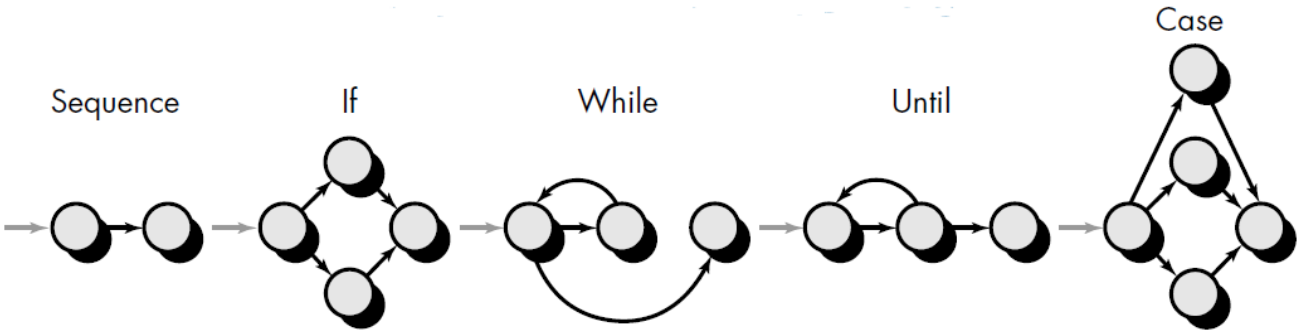
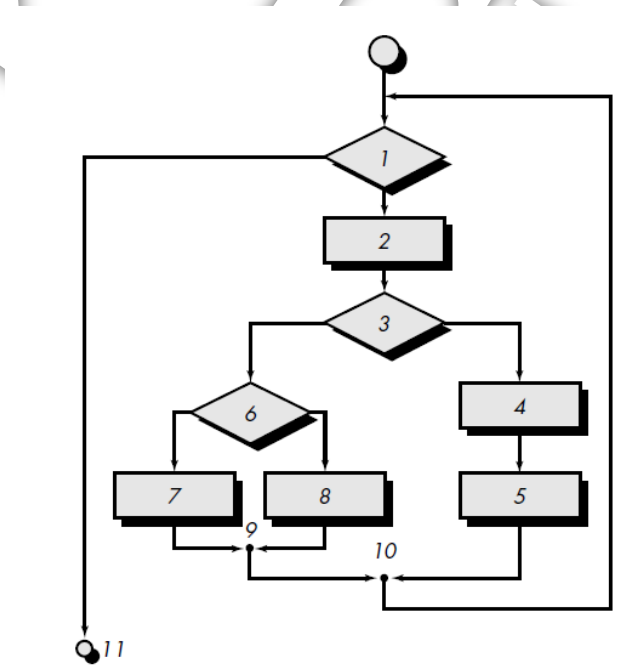


Figure 10.1 Flow graph notation

To illustrate the use of a flow graph, we consider the procedural design representation in Figure 10.2A. Here, a flowchart is used to depict program control structure.

Figure 10.2 B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart). Referring to Figure 10.2B,

- Each circle, called a flow graph node, represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct).
- Areas bounded by edges and nodes are called regions. When counting regions, we include the area outside the graph as a region.



(A)

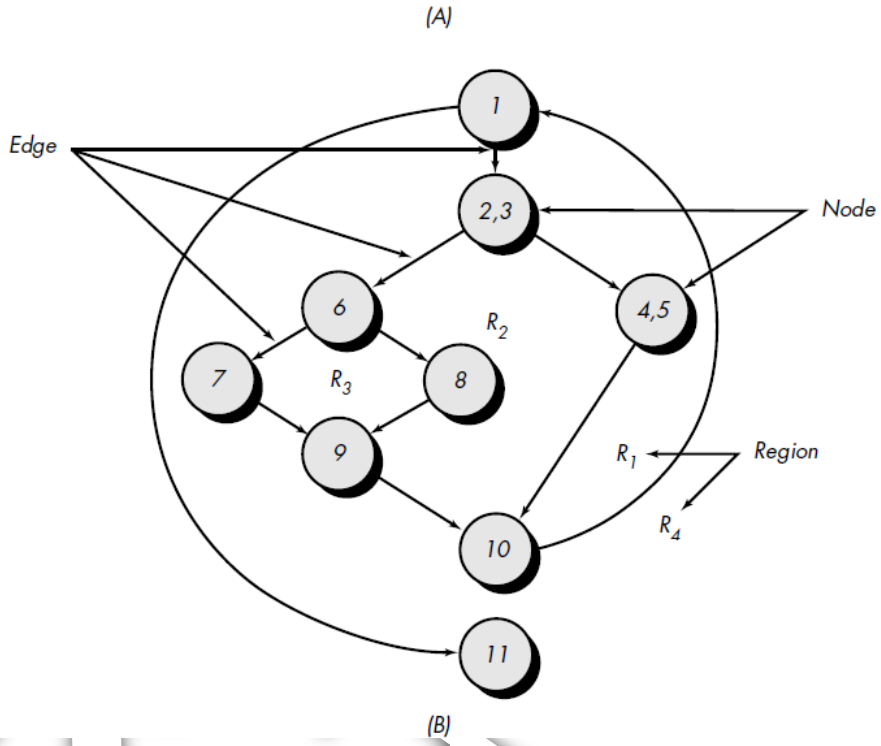


Figure 10.2 Flowchart, (A) and flow graph (B)

When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement. Referring to Figure 10.3, the PDL segment translates into the flow graph shown. Note that a separate node is created for each of the conditions a and b in the statement IF a OR b.

Each node that contains a condition is called a predicate node and is characterized by two or more edges emanating from it.

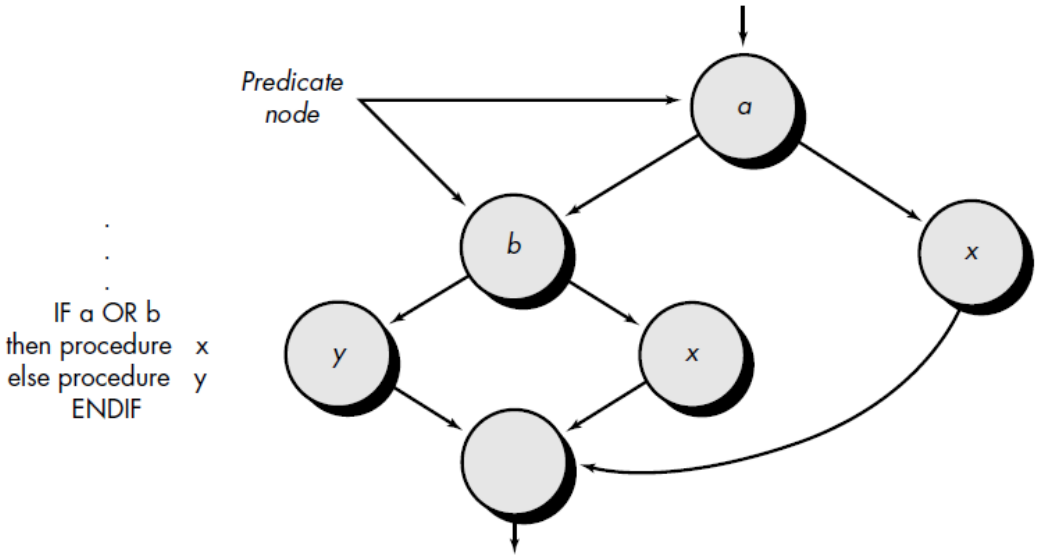


Figure 10.3 Compound Logic

10.3.2 Cyclomatic Complexity

Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 10.2B is: