Techniques for Functional and Structural Testing of Software

Alfredo Cruz, PhD
Associate Professor
Department of Electrical Engineering
Polytechnic University
email: across@coqui.net

Adrián E. Méndez
Graduate Student
Polytechnic University
email: amendez@pupr.edu

ABSTRACT

This work will present the structural and functional approaches as software testing process techniques. The software testing process is one that consumes at least half of the labor expended to produce a working program. Also, the cost of testing critical software can be three to five times as much as the cost of all the other activities combined. For those reasons the development of techniques to test a program in order to reduce time and cost becomes necessary. Some of the techniques of the structural and the fundamental approaches will be studied here as well as the comparison of two of their techniques: the boundary test and the path test, with their extensions: the boundary worst case and the branch and statement test.

SINOPSIS

En este trabajo se presentarán los enfoques estructural y funcional como técnicas para el proceso de prueba de programas de computadoras o "software testing". Este proceso generalmente consume al menos la mitad del esfuerzo realizado para producir un programa veraz y fidedigno. Además el costo de prueba de un programa de función crítica puede ser de tres a cinco veces el costo total de todas las demás actividades combinadas. Por esta razón el desarrollo de técnicas para el proceso de prueba son necesarias tanto como para reducir el tiempo de prueba como el costo de realizarlas. Se estudiarán algunas técnicas del enfoque funcional como lo son la prueba de límites y la prueba de equivalencias con algunas de sus variantes. También se estudiarán algunas técnicas del enfoque estructural como lo es la prueba de pasos con algunas de sus variantes la prueba por declaración y la prueba por cobertura de ramas.

I- INTRODUCTION

The main limitation of software testing is that, in order to prove that a program is 100% free of bugs, the testing should be very exhaustive and thorough. Exhaustive and thoroughly testing will not only take a lot of time in research and testing but also a lot of resources and money. Exhaustive testing requires the execution of every statement in the program and every possible path combination through the program. In practice, this is impossible. There are an infinite number of path combinations as each sequence of loop execution represents a separate path. Testing must be based on a sub-set of possible test cases and a good choice of the test data. In an exhaustive test of an integer addition algorithm, the test case would include approximately 2^{64} test execution, assuming that integers are stored in 32 bits. For a computer that performs 2^{24} operations per second, it will take 2^{40} seconds or approximately 35,000 years to complete an exhaustive test of the addition algorithm.

Approximately 65% of all bugs can be caught in unit testing which the path testing method dominates. Path testing catches approximately half of all bugs caught during unit testing or approximately 35% of all bugs [1]. When path testing is combined with other methods such as limit checks on loops, the percentage of bugs caught rises from 50% to 60% in unit testing. Path testing is more effective for unstructured than for structured software [2].

A- BASIC NOTATION AND DEFINITIONS

Because testing technology has evolved over decades, some of the terms can be confusing. Some of the these terms are:

Definition 1: Error - An error is a non-expected mistake. It usually means that the software does not do what the requirements document describes, or that the system is not working properly. When the programmer(s) make this "mistake" while coding, we refer to the "mistake(s)" as bug(s).

Definition 2: Fault - The defect that is found as the result of an error is called a fault. Basically, faults fall in two categories:

Fault of commission - Occurs when we enter something into a representation that is incorrect.

Fault of omission - Occurs when we fail to enter correct information.

Definition 3: Failure - A failure is an occurrence of an error somewhere in the software system, it occurs when a fault executes.

Definition 4: Incident - An incident is the symptoms associated with a failure that alerts the user of an occurrence of a failure.

B- The Goal of Software Testing

The goal of software testing is not to make 100% error free programs, which may be almost impossible and also have several consequences, but to establish the presence of defects in a program in order to identify and correct or debug the defect. Debugging usually follows testing, but they differ in respect to goals, methods, and most important, psychology [2]. Testing is finding the existence of errors and debugging is finding the cause of the errors and correcting them.

II- SOFTWARE TESTING FUNDAMENTAL APPROACHES

To identify the test case the programmers have two fundamental approaches: functional testing and structural testing. Each of these approaches has several distinct test case identification methods, commonly called testing methods.

III- FUNCTIONAL TESTING

In the functional testing approach, the program or system is treated as a *Black Box*. Only inputs and outputs are taken in consideration by the tester when testing and evaluating the program or system. In this approach, the tester considers the program to be tested like a function that maps values from its input domain to values in its output range.

This type of testing attempts to find errors provoked by incorrect or missing functions, interface errors, errors in data structure or external database access, performance errors, and initialization and termination errors. The most important advantages offered by functional testing are that it is independent of how the software is implemented and that the test case development can occur in parallel with the implementation.

The negative side of this approach, since the functional method is based on a specific behavior, is that it cannot identify behaviors that are not specified.

One of the mainline approaches of functional testing will be examined next.

A- The Angle Problem Solver

An example of a problem to determine the angle of a function of the two variables S and R is presented to analyze the Black Box testing method. The function is as follows:

$$\theta = \frac{S}{R},$$

where θ is the angle in radians, S is the arc length of a circle and R is the radius of the circle.

Next, we can determine the final quadrant of the angle described by the function shown above. The possible output of the program should be one of the four quadrants described by the coordinate axis of x and y. The final position of a point will be determined by the function θ. The definition of a quadrant would be the angle between the range:

Quadrant 1 = $0 < Q1 < 90$,
Quadrant 2 = $90 < Q2 < 180$,
Quadrant 3 = $180 < Q3 < 270$,
Quadrant 4 = $270 < Q4 < 360$

The values for S and R would be restricted to values from 1 to 20. The code of the program implemented in C language is as follows:

```
/* Program that determines the quadrant of a given
angle determined by a function of two variables */

pi = 3.141593
variables r, s, angle, refangle;

if (s<=20 && r<=20 && s>0 && r>0) {
      angle=((s/r)*(180/pi));
      while (angle>360)
            refangle = angle - 360;
            /*give to angle a value in
            range from 0 to 360 */

if (angle>0 && angle<90)
      printf ("\n First quadrant");
else
      if (angle>90 && angle<180)
            printf ("\n Second quadrant");
      else
```

```
        if (angle>180 && angle<270)
              printf ("\n Third quadrant");
        else
              if (angle>270 && angle<360)
                    printf ("\n Fourth quadrant");
              else
                    printf ("\n Quadrant axis");

}   /* End of the program */
```

B- Boundary Value Testing

Boundary value analysis focuses on the boundaries of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. The U.S. Army (CECOM) made a study of its software and found that a surprising portion of faults turned out to be boundary value faults. Loop conditions, for example, may test for < when they should be tested for <= , and counters are "off by one".

The basic idea behind boundary value analysis is as follows: the use of input variables at their minimum, just below their minimum, a nominal value just below their maximum and their maximum. These values are referred to as min, min+, nom, max-, and max+. This convention will be used for this study.

The next part of boundary value analysis is based on a critical assumption known as the "single fault" assumption in reliability theory. This assumption says that failures are rarely the result of the simultaneous occurrence of two or more faults.

To consider the number of variables in a function of n variables, it is necessary to set one variable at the nominal value. The remaining variables are set to the min, min+, nom, max- and max values. This process should be repeated for each variable of the function. Thus, the boundary value analysis for a function of n variables yields 4n + 1 test cases. For our example of angle problem of a function of two variables (S and R), the boundary test would produce 4(2) + 1 = 9 test cases. The test cases for the angle problem are shown in Table 1.

The boundary value analysis test cases for the function F of two variables are:

{$<x_{1nom}, x_{2min}>$, $<x_{1nom}, x_{2min}+>$, $<x_{1nom}, x_{2nom}>$, $<x_{1nom}, x_{2max}->$, $<x_{1nom}, x_{2max}>$, $<x_{1min}, x_{2nom}>$, $<x_{1min}+, x_{2nom}>$, $<x_{1nom}, x_{2nom}>$, $<x_{1max}-, x_{2nom}>$, $<x_{1max}, x_{2nom}>$}

C- Worst Case Testing

Worst case testing is another extension of boundary analysis. It follows the same generalization and has the same limitation. The main difference between each analysis is in the number of test cases produced for testing the software.

The number of test cases produced would be in a function F of n number of variable equal to 5^n. In general, the result is that a larger number of test cases would be produced. The test cases are the Cartesian product of the input variable defined for the min, min+, nom, max- and max values.

For the angle problem of two variable, the test cases produced would be $5^2 = 25$ test cases. These test cases are shown in Table 2.

Worst case testing is clearly more thorough in the sense that boundary value analysis test case is a proper subset of worst case testing. It is easy to see the assertion when comparing the test cases

Table 1: *Test Case for the Boundary Test*

	S	R	Expected Output
Sn , Rmin	7	1	First Quadrant
Sn , Rmin+	7	2	Third Quadrant
Sn , Rn	7	7	First Quadrant
Sn , Rmax-	7	19	First Quadrant
Sn , Rmax	7	20	First Quadrant
Smin , Rn	1	7	First Quadrant
Smin+, Rn	2	7	First Quadrant
Smax- , Rn	19	7	Second Quadrant
Smax , Rn	20	7	Second Quadrant

Table 2: *Worst Cases*

	S	R	Expected Output
Smin , Rmin	1	1	First Quadrant
Smin , Rmin+	1	2	First Quadrant
Smin, Rmax-	1	19	First Quadrant
Smin, Rmax	1	20	First Quadrant
Smin+, Rmin	2	1	Second Quadrant
Smin+, Rmin+	2	2	First Quadrant
Smin+, Rmax-	2	19	First Quadrant
Smin+, Rmax	2	20	First Quadrant
Smax-, Rmin	19	1	First Quadrant
Smax-, Rmin+	19	2	Third Quadrant
Smax-, Rmax-	19	19	First Quadrant
Smax-, Rmax	19	20	First Quadrant
Smax, Rmin	20	1	First Quadrant
Smax, Rmin+	20	2	Third Quadrant
Smax, Rmax-	20	19	First Quadrant
Smax, Rmax	20	20	First Quadrant

produced by each boundary approach: Boundary Value test cases = 9 and Worst Case test cases = 25.

Note that the Worst Case test will give the same test cases produced by the Boundary Value test cases and those produced by the Cartesian product of the input variable.

Probably the best application for worst case testing is where physical variables have numerous interactions and when the failure of the function represents a high cost.

As with any other technique, the boundary value analysis offers some limitations. These limitations are not only related with the variable type, but also when the variables are not independent and do not represent bounded physical quantities. The boundary analysis test case is derived from the extreme of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function nor the semantic meaning of the variable.

D- Equivalence Class Testing

The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, the potential redundancy among test cases is greatly reduced.

An equivalence relation is a relation that satisfies the *reflexive*, *transitive* and *symmetric* properties. An example of an equivalence relation is the numerical equality. An equivalence class relation occurs when a set of objects satisfies an equivalence relation. The significance of this concept is that we can relate an object from one class to any other object of that class. The idea behind this concept is to make partition of the input space so that every object in each subset of the partition is equivalent. Then, if any object of the partition is tested, we can assume that each member of the same partition would respond in the same form as the object tested. This methodology is known as equivalence partitioning.

The equivalence class then helps to select a small subset of possible inputs and make a well-selected test case. The test case produced obeys the characteristics described by the following statements [4]:

1- It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of "reasonable testing".
2- It covers a large set of other possible test cases. That is, it tell us something about the presence or absence of errors over and above the specific set of input values [4].

This process of test case design proceeds in two steps: identifying the equivalence classes and defining the test cases. To identify the equivalence class each input condition must be taken and partitioned in two or more groups. As an example of variable inputs we can identify two types of equivalence classes; valid equivalence classes or the valid inputs to the program, and the invalid equivalence classes, which are all other possible erroneous input values.

In the process of identifying the test cases, a unique number has to be assigned to each equivalence class as shown in Example 1 and Example 2. All valid equivalence classes must be covered by test cases; it is also necessary to make as many test cases as possible for the uncovered valid equivalence classes.

In the same way, all invalid equivalence class have been covered by test cases, but for the uncovered invalid equivalence class only one test case is written. This helps to reduce the number of test cases and avoids some program error failures.

As an example of equivalence class testing, the angle problem and some of the different approaches to derive the test cases will be considered.

Example 1: *Traditional Equivalence Class (Eq. C.)*

External condition	Valid Eq.C.	Invalid Eq.C.
S and R should be 0 < S, R < 21	0 < S < 21 (1)	S < 1 (3)
		S > 20 (4)
	0 < R < 21 (2)	R < 1 (5)
		R > 20 (6)

Example 2: *Uncovered Valid Output*

possible outputs	First Quadrant (7)
	Second Quadrant (8)
	Third Quadrant (9)
	Fourth Quadrant (10)
	Axis Quadrant (11)

Note that the number of test cases is the sum of each equivalence class (which equals 11)

Table 3 shows the test cases derived [4] as described at the beginning of this section. Notice that some equivalencies are repeated. This does not necessarily means or represents the same equivalence class.

The Output Range Equivalence classes are equivalencies defined from the output range. This gives some sense that the test is exercising important parts of the program. The test cases for the angle problem is shown in table 4:

Table 3: *Traditional Equivalence Cases*

Test cases	S	R	Expected output
Tec 1	7	7	First Quadrant
Tec 2	7	7	First Quadrant
Tec 3	0	7	Error Msg.
Tec 4	21	7	Error Msg.
Tec 5	7	0	Error Msg.
Tec 6	7	21	Error Msg.
Tec 7	7	7	First Quadrant
Tec 8	19	7	Second Quadrant
Tec 9	7	2	Third Quadrant
Tec 10	15	3	Fourth Quadrant
Tec 11	1.570798	1	Axis Quadrant

Table 4: *Output Range Equivalence Cases*

Test Cases	S	R	Expected Output
OUT 1	7	7	First Quadrant
OUT 2	19	7	Second Quadrant
OUT 3	7	2	Third Quadrant
OUT 4	15	3	Fourth Quadrant
OUT 5	1.570798	1	Axis Quadrant

IV- STRUCTURAL TESTING

The other fundamental approach is the Structural Testing, sometimes called *White Box*. In structural testing, the test object is viewed as an open box where the internal structure and logic are completely familiar to the programmer. The programmer then can look at the implementation details in a programming style, controlling methods and coding details. This allows the tester to identify test cases based on how a function is actually implemented.

A- PATH TESTING

Path testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through a program [1]. If the set of paths is properly chosen, then we have reached some measure of test thoroughness.

Three different testing strategies out of potentially infinite family of strategies are:

Statement testing – Every statement in the program is executed at least once under some testing, as denoted by C_1 [1]. For decision statements like IF-THEN-ELSE, we could execute just one decision alternative and satisfy the statement coverage criterion. One weakness of statement testing is that there is no guarantee that all outcomes of branches are properly tested. This is the weakest criterion in the path testing family.

The table 5 shows the statement test cases derived for the angle problem.

Branch testing – For every decision point in the program, each branch alternative is exercised at least once under some test, as denoted by C_2. For decision statements like IF-THEN and IF-THEN-ELSE statements, both true and false branches have to be covered. The table 5 shows the cases derived. Note that the number of test cases is reduced to two.

The starting point for path testing is a program flow graph. A flow graph consists of nodes representing decisions and edges showing flow control. The path through the programs are the sequence of instructions or statements that start at an entry, junction or decision and end at another, possibly the same junction, decision or exit.

Figure 1 shows a flow graph for the angle problem. Note that the flowchart represents each statement in the program, while the flow graph only represents the decision statements of the program. Each decision statement is represented by nodes (circles) and the possible paths are represented by the arrow lines.

The number of instructions or statements executed along the path measures the length of the path. The assumption of this approach is that something has gone wrong with the software that makes it take a different path than the intended.

B- STRUCTURAL TESTING FOR THE ANGLE PROBLEM

For the angle problem, let's see some of the test cases derived for the statements and branch test using Figure 1.

1- Statement test (C_1):

Table 5: Statement Cases

S	R	Path	Expected output
0	7	1, 2, 3, 16	Error Msg.
1.5707	1	1, 2, 4, 5, 6, 7, 9, 11, 13, 15, 16	Axis Quadrant

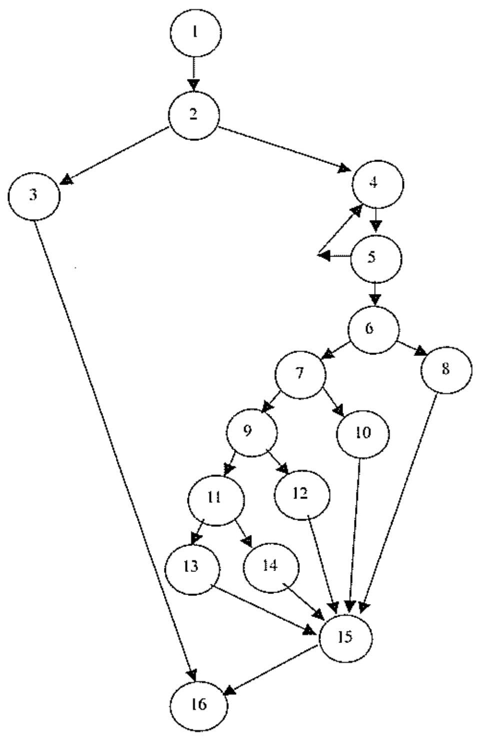

***Figure 1:** Flow graph for the angle's problem*

Note that in table 5, for values of S = 1.5707 and R = 1 all decisions are covered for at least one alternative.

2- Branch Coverage (C_2)

***Table 6:** Branch Coverage Cases*

S	R	Path	Expected Output
7	0	1, 2, 3, 16	Error Message
1.5707	1	1, 2, 4, 5, 6, 7, 9, 11, 13, 15, 16	Axis Quadrant
7	1	1, 2, 4, 5, 6, 8, 15, 16	First Quadrant
19	7	1, 2, 4, 5, 6, 7, 10, 15, 16	Second Quadrant
7	2	1, 2, 4, 5, 6, 7, 9, 12, 15, 16	Third Quadrant
15	3	1,2, 4, 5, 6, 7, 9, 11, 14, 15, 16	Fourth Quadrant

These test cases shown on table 6 give a major sense of branch coverage as it verifies each outcome, true or false, for each decision statement. The larger number of test cases derived, compared with the statement derived test cases, gives a better sense of proving or testing.

This structural approach has several limitations. The number of unique logic path through a program is astronomically large. Exhaustive path testing means a complete test, that is, every path in a program could be tested yet the program might still be loaded with errors. This could have a dead end or path that the test case cannot reach.

Nevertheless, it is important to remember that the predicted result for a path test is the path itself, not the output of the program or function.

V- CONCLUSIONS

Because of the combinational explosion, neither exhaustive testing to *specifications* (Black Box) nor testing to *code* (White Box) is feasible. A compromise is needed, using techniques that will highlight as many faults as possible, accepting that there is no way to guarantee that all faults have been detected. A reasonable way to proceed is to use black-box test cases first (testing to specifications) and then develop additional test cases using white-box techniques (testing to code).

There is no controversy between *structural* versus *functional* tests: both are useful, both have limitations, but each target different kinds of bugs. *Functional* tests can, in principle, detect all bugs but would take infinite time to do so. *Structural* tests are inherently finite but cannot detect all errors, even if completely executed. There is no best method, neither an approach is sufficient by itself. Only a judicious combination will provide the confidence of functional testing and the measurement of structural testing.

VI- REFERENCES

1- Beizer, B. (1990). "Software Testing Techniques". 2nd Edition. New York: Van Nostrand Rheinold.
2- G. M. Weinberg 1971 , "The Psychology of Computer Programming"
3- Boehm and et al. (1975). "Some Experience With Automated Aids to the Design of Large-scale Reliable Software". In Proc. *IEEE Trans. Software Engineering*, **SE-10**, 3, pp. 290-303.
4- Myers, G. J. (1979). "The Art of Software Testing". New York, Wiley Interscience.
5- Plfeeger, S.L. (1987) "Software Engineering". 2nd Edition. New York, Macmillian Publishing Company.