A SURVEY OF SOFTWARE FUNCTIONAL TESTING METHODS

A.A.OMAR F.A.MOHAMMED
Atomic Energy Organization, Cairo, Egypt

ABSTRACT

Functional testing is used to find disagreement between the specifications and the actual implementation of the software systems. The method of representing the specification can help to detect inconsistency and incompleteness in it. The various specification representation schemes are outlined in the paper. The basic technique of functional testing of software systems is the black box technique. This technique generates the test data using the information contained in the program's specification, independent of the implemented program's code. Black box testing cannot discover errors contained in the functions which are not mentioned explicitly in the specification. Therefore, a program dependent testing is necessary to discover this type of errors. The paper surveys the different methods of generating test data for both techniques; the black box and the program dependent techniques.

I- INTRODUCTION

The purpose of software testing is to determine whether a program contains any errors [9]. In general, software errors can be classified as performance errors and logic errors. Performance errors are the failure to produce results within specified or desired time and space limitations. Logic errors are caused by the production of incorrect results independent of the time and space required. There are four types of logic errors:
1- Construction errors; the failure to satisfy a specification through error in an implementation.
2- Specification errors; the failure to write a specification that correctly represents a design.
3- Design errors; the failure to satisfy an understood requirements.
4- Requirements errors; failure to satisfy the real requirements.
Tests must be able to detect all these types of errors, and this means that test data selection criteria must reflect information derived from each stage of software development.

Program testing methods can be divided into two broad classes, depending on the way in which test data is generated; functional testing and structural testing [15],[35]. Functional testing involves the generation of tests that are based on the requirement, specifications, and design functions for a program. Structural testing makes use of the program flowgraph in designing an adequate test. Test data generation for functional testing can be carried out during requirement, specification, and design; while test data sets that satisfy structural testing can only be generated after complete coding. The two approaches are complementary rather than exclusive. This paper deals with the functional testing methods.

The functional testing technique consists of two parts:
a) Identification of functions and selection of functional test data from requirements, specifications, and design documents.
b) Each of the functions in the development documents is tested by executing the section of code that implements it. Test data are selected by first analyzing the I/O variable spaces for the function and then choosing input values that cause the function to be tested over important values of function variables.

Before starting the functional testing process, it is preferable to test the specification in order to ensure that it is consistent, complete, and correct. If so, then the specification will represent the actual requirements well. The method of representing the specification affects the correctness and helps to reduce the number of errors in it. Some representation schemes of the software specifications are discussed in the next section, along with a discussion of some methods for testing the specifications. The later sections will be concerned with the discussion of each part of the functional testing techniques in more detail.

II- SPECIFICATION REPRESENTATION AND TESTING

The specification of a software system describes in detail the expected external behavior of the system. Errors in specifications are often not detected until after system implementation. These errors may arise due to ambiguity, inconsistency, and incompleteness in the specifications and the elimination of them is difficult specially when the specification is written in a natural language. The use of formal models to represent the specification reduces the number of errors that may remain in the specification because the formal techniques can alert the specification writer to many potential errors. A survey of some of the available techniques designed to represent the software specifications are described below [4]. Then a discussion about how to test the specification follows.

2-1 Finite State Machine

A finite state machine (FSM) is a hypothetical machine that can be in only one of a given number of states at any specific time. In response to an input, the machine generates an output and changes state. Specifications can be represented as a FSM using either the state transition diagrams (STD) or the state transition matrices (STM). In the STD,a circle denotes a state, a label on a direct arc connecting two states denotes the input that triggers the transition and the output with which the system responds. In an STM, a table is drawn with all the possible states labeling the rows and all the possible stimuli labeling the columns. The next state and the required system response appear in each intersection.

Finite state machines have been used effectively for

requirements specifications for telephony applications [20,32].

2-2 Decision Tables and Decision Trees.

When the FSM representation makes no sense for a specific system, we can use the decision tables and decision trees techniques [3,23]. To construct a decision table (DT), first draw a row for each condition (or stimulus) that will be used in the process of making a decision. Next draw a column for every possible combination of outcomes of these conditions. Finally, add rows at the bottom of the table for each action that you may want the system to perform (or generate), and fill in the boxes to reflect which actions you want performed for each combination of conditions. DTs can easily be automated with any spreadsheat package.

A decision tree captures the same information as a DT but is graphical rather than tabular. Basically it is a flowchart without loops and without more than one row pointing to the same node.

2-3 Program Design Language

Program Design Language (PDL) is simply free form English with special meanings for certain key words. One of the biggest problems with using PDL during the requirements phase is the incredible ease with which one can fall into design. To stay out of design, one should select all the words that represent things that can be seen, smelled, felt, heard, or tasted by somebody who is an external observer of the system. PDL presentation was demonstrated for telephony by Casey [2] and Davis [5] and for a flight control application by Heninger [13].

2-4 Statecharts

Statecharts are extensions to FSMs and are proposed by Harel [11]. They make it easier to model complex real-time system behavior without ambiguity. The first extension to FSMs allows a transition that is a function not only of an external stimulus but also of the truth of a particular condition. The next extension is the superstate, which can be used to aggregate sets of states with common transitions. For example, if there are two states S1 and S2 both transition to state S3 upon the same stimulus I, then S1 and S2 could be grouped together, figure (1).

Figure (1) The Statechart

2-5 Petri-nets

Petri-nets were first introduced in 1962 by Petri [29] and later described by Petrson [28]. They are relatively simple to understand and are best used to describe pieces of intended system behavior where ambiguity cannot be tolerated and precise process synchrony is important. Applying Petri-nets for the requirements specification can be illustrated in the following example. Figure (2) shows a small fragment of the specification of the external behavior of a telephone switching system. At the arrival of the first clock pulse, the dial tone place will be fired. The second clock pulse will cause the ringing place to be fired, then the third clock pulse will cause the connect place to be fired. For other different examles, refer to [1],[33].

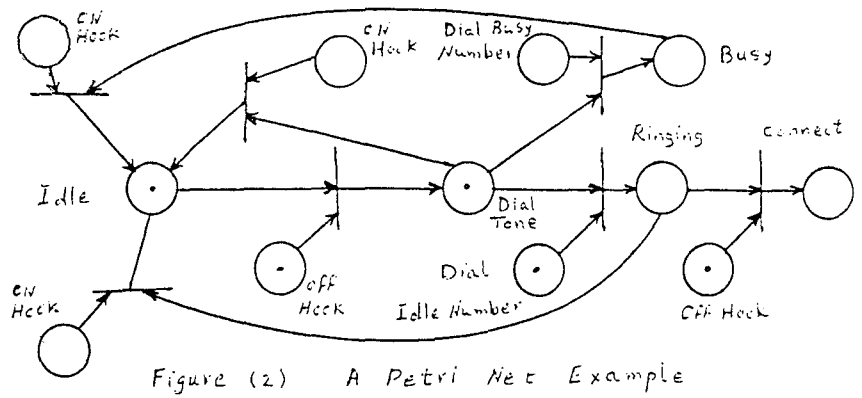

Figure (2) A Petri Net Example

2-6 Specification Testing

As stated before, the specification must be shown to be consistent with the original requirements. The approach to do so differs according to the way in which the specification is represented. Kemmerer [21], used the state machine representation, which specifies the effect of performing each operation based on certain entry condition being satisfied. The formal specification of the system described by Kemmerer is written in a language called Ina Jo. This language is a nonprocedural assertion language that is an extension of first-order predicate calculus. The nonprocedural languages are not executable, and this represents a problem. There are two approaches to solve this problem. The first is to convert the nonprocedural specifications into a procedural program (PASCAL for example). This procedural program then serves as a rapid prototype to use for testing the specifications to see if they allow the desired functionality. However, since the resulting program represents only one of many possible consistent implementations, a successful test shows only that the functional requirements are satisfiable by this implementation but not necessarily valid. To be valid, it must be proved that all implementations of the formal specifications are consistent with the functional requirements. Functional requirements are tested on the executable code by trying different test cases that are instances of the functional requirement.

The second approach for dealing with the nonprocedural language is to perform a symbolic execution of the sequence of operations and check the resultant symbolic values to see if they define the desired set of resultant states. The difference between the prototype and the symbolic tools is that the prototype tool compiles the nonprocedural program into procedural

one. Thus the rapid prototype tool is the more difficult to implement, but it provides the user with a prototype which he can exercise to see if it meets his functional requirements.

Jalote [19], describes a system that automatically tests for completeness of axiomatic specifications of abstract data types. The axiomatic specofication technique of abstract data types is illustrated in [8],[10],[24]. The basic approach is to generate an implementation from the specifications that faithfully translates the specifications into an executable program, such that if the specifications are not complete, the implementation is not complete, and the behavior of all of the sequences of valid operations on the abstract data type is not defined. The system also contains a test case generator that creates a set of test cases. The implementation is executed with the generated test cases to determine the completeness of specifications. A side benefit of the system is that the results can also be used to check the correctness of specifications.

In [12], other testing techniques which are applied to the abstract data types are described. They differ from the previous methods [7],[10],[19] in that the previous methods use an algebraic specification of a data type using axioms which interrelate the operation on the data types, while in [12] a model-based specifications of the individual operations is used which gives the effect of each operation acting upon an abstract state. The algebraic approach is more appropriate for more primitive data types, while the model based approach is more manageable for specifying larger modules.

III- BLACK BOX TESTING

Black box [30] refers to testing which involves only observation of the output for certain input values; that is there is no attempt to analyze the code which produces the output. The internal structure of the program is ignored during test data set selection [14]. Tests are constructed from the functional properties of the program that are specified in the program's requirements. To find all errors in the program using the black box approach, exhaustive input testing should be applied. Exhaustive input testing is the use of every possible input condition as a test case. This is absolutely impossible since the number of inputs to any program, in general, is unlimitted. For example, if it is required to test a compiler, it should be tested on every possible source program expected to be compiled. We should also apply any invalid program to that compiler to ensure that the compiler will detect any error in that invalid input program.

Since exhaustive testing is out of question, the objective of testing should be to maximize the number of errors found by a finite number of test cases. We should design effective test cases that have the highest probability of detecting most of the errors. The remainder of this section will discuss most of the well-known black box testing methodologies.

3-1 Random Input Testing

Random input testing is probably the poorest of all of the test case design methods. It is the process of testing a program by selecting, at random, some subset of all possible input values. In terms of the probability of detecting most of the errors, a randomly selected test cases has little chance of being an optimal subset.

3-2 Equivalence Partitioning [25]

In the equivalence partitioning method, the 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 a test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in a class detects an error, all other test cases in the class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the class would find an error. Two steps are required in implementing this method:

1- The equivalence classes are identified by taking each input condition (usually a sentence or phrase in the specification) 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 contain more than one invalid class. This is to ensure that no two invalid classes mask each other.

3-3 Boundary-Value Analysis [25]

Boundary-value analysis divides the input domain into equivalence classes as the previous method. The differences between the two methods are:

1- Boundary-value analysis requires that one or more elements be selected from each equivalence class such that each edge of the class is the subject of a test.

2- Test cases are also derived by considering not only the input domain but also the output equivalence classes.

3-4 Cause-Effect Graphing [6],[25]

One weakness of boundary-value analysis and equivalence partitioning is that they do not explore combinations of input circumstances. The testing of input combinations is not a simple task because the number of combinations is usually astronomical. Cause-effect graphing [6] is a systematic method of generating test cases representing combinations of conditions. It has a beneficial side effect in pointing out incompleteness and ambiguities in the specification.

A cause-effect graph is a formal language into which a natural language specification is translated. The graph is actually a digital-logic circuit using a simpler notation than

the logic notation (refer to [25] for details of the notations). The following process is used to generate test cases:
1- The causes and effects in the specification are identified. A cause is a distinct input condition or an equivalence class of input conditions. An effect is an output condition. Causes and effects are identified by reading the specification word by word and underlining words or phrases that describe causes and effects. Each cause and effect is assigned a unique number.
2- The semantic content of the specification is analyzed and transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph.
3- The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
4- By tracing the state conditions in the graph, the graph is converted into a limited-entry decision table [22]. Each column in the table represents a test case.
5- Columns in the decision table are converted into test cases.

Although cause-effect graphing does produce a set of useful test cases, it normally does not produce all of the useful test cases that might be identified. Also, the cause-effect graph does not adequately explore boundary conditions. For this reason it is best to consider a separate boundary-value analysis. The most difficult aspect of the technique is the conversion of the graph into the decision table. This process is algorithmic, so it could be automated by writing a program. Another problem with the cause-effect graphing method is that it is difficult to apply in practice, particularly because it can become very complex when a function has a large number of causes [27].

3-5 The Condition Table Method [9]

The goal of this method is to define how to select test data for a particular program such that if the program processes all test data correctly, we can be reasonably confident the program will process all data correctly. The method begins by constructing a condition table in which each column represents a combination of conditions that can occur during execution of the program. There are several sources of information about the conditions and combinations of conditions relevant to a program's operation:
1- The general requirement a program is to satisfy.
2- The program's specification.
3- General characteristics of the implementation method.
It can be seen here how properties of a condition table is a natural way to check a specification for completeness and lack of ambiguity.
A test predicate is found by identifying conditions and combinations of conditions relevant to the correct operation of a program. Conditions arise first and primarily from the specification for a program. As implementations are considered, further conditions and predicates may be added. Conditions are written down in the table, and then the columns of the table are generated and checked. The checking process may suggest further conditions to be added.

The condition table method forces attention toward specifications and what the program should do rather than an actual program structure and what a program seems to do. It is not divorced from a program's internal structure since all program predicates must be represented in the condition table if the table is to define a reliable set of test predicates.

3-6 The Category-Partition Method

The category-partition method [27] is a specification-based method that uses partitioning to generate functional tests for complex software systems. The method begins with the decomposition of the functional specification into functional units that can be tested independently. This process increases the chances of finding problems in the specifications. The tester performs all the next steps to each functional unit in order to test it. The tester identifies the parameters that affect the function's execution behavior. Parameters are the explicit inputs to a functional unit, supplied by either the user or another program. The next step is to determine the different characteristics of each parameter, each characteristic will be called CATEGORY. The three steps performed till now: identify the functional units, identify the parameters of each unit, and determine the categories of each parameter are performed by reading the specification carefully and analysing the meaning of every word. The next steps will use the information derived to generate the test cases.

Each category is partitioned into distinct classes, each class includes the different kinds of values that can be treated the same by the function. These classes are called CHOICES. Each choice within a category is a set of similar values that can be assumed by the type of information in the category. The choices are the partition classes from which representative elements will be taken to build test cases. The tester then determines the constraints among the choices of the different categories. For example, a choice from one category cannot occur at the same time in a test case with certain choices from other categories. Once the categories, choices, and constraints have been established, they are written into a formal test specification for each functional unit. The written specification is then processed by a generator to produce a set of test frames. A test frame consists of a set of choices, with each category contributing either zero or one choice. The tester then converts the test frames produced by the tool into actual test cases by specifying a single element from each of the choices in the frame.

The advantages of the category-partition method are that the tester can easily modify the test specification when necessary, and can control the complexity and number of the tests by annotating the tests specification with constraints. The disadvantage of this method, like all the partitioning methods, is that it lacks a systematic approach and depends on the experience of the tester and thus it cannot be automated.

IV- PROGRAM DEPENDENT METHODS

Black box testing fails to find errors for which it is not enough to test a function with a particular choice of special value for some input variable, but for which it is necessary to execute parts of the code for that choice of input value [15]. The white box rules can be needed to force some parts of code to be executed using branch or path testing [18],[25],[26].

4-1 Functional Testing

Howden [14], proposed a refined method of the black box technique in such a way that it could be used to test the important functional properties of the design of a program, as well as the functional properties that are part of the program's requirements. This approach is called the functional testing, and like all other testing approaches, depends on the availability of an oracle [16,17]. Oracles are external sources of information about functions. There are three basic kinds of oracles:

1- Input-Output oracle, which is capable of determining ,for a given input case, the output correctness for that input.
2- Trace oracle, which is capable of determining whether a sequence of function calls occuring along some path is correct.
3- Interface oracle, which may be used to determine whether it is correct for one type of data to be transformed into another.

The design and implementation of a program involves the integration and concatenation of a set of functions. In the functional testing method, it is required that a complete set of tests be constructed for each of the functions which are part of a program's design. Different types of design functions are :

1- Computational functions which compute values that are either part of the program's output or are used by other functions to compute output values.
2- Control functions which choose between alternative computations.

Functions are considered to be either elementary or formed by composition from other functions. There are two general kinds of compositions : forms and structures. Structural synthesis will be discussed in the next subsection.

Forms are divided into three kinds: algebraic, conditional, and iterative. In algebraic synthesis, a function is synthesized by joining functions together using an algebraic expression or relation. In conditional synthesis, a conditional form can be used to synthesize a new function F3 from two functions F1 and F2 and a Boolean function B as in : F3(x) = if B(x) then F1(x) else F2(x). In iterative synthesis, an iterative form is used to compose a function from a loop function and a Boolean as in :

```
F2(x) = repeat while B(x)
        x <-- F1(x)
        endrepeat
```

There is a natural association between I/O oracles, fault detection, and forms. Functional testing depends on the availability of an I/O oracle which can be used to determine, for a formed function, whether the output produced by that function is correct or not for a selected fault revealing input test data. Functional testing of design functions requires that the domains of all I/O variables for each design function be fully documented in the comments preceding each code section. In order to test the design functions in a program, it is necessary to monitor the intermediate values of program variables that are used as I/O variables by the functions. Test harness [17] can be used for that purpose. It also may be used to insert the required values into the function intermediate variables.

We will now discuss the characteristics of the functions, the methods of identifying and testing them, and the application of functional testing method.

4-1-1 Functions.

A function is defined to be a set of ordered pairs(xi,yi) where xi=input, yi=output. A program is considered to be a function of one or more inputs and one or more output variable. Each variable is defined over a set of possible values called the domain of the variable.

Domains can contain numbers, data structures or other programs:

1- Numeric variables

The only property of the element in the domain of numeric variable is its value. Functional testing requires that the set of allowable values for each input and output variable for a program be formally defined. The nature of the domains of input and output variables affects the test data in the following manner:

a) If the input variable domain is a small set of discrete values, tests should be carried out on all these values.

b) If the domain of the input variable consists of one or more intervals of numbers, tests should be carried out on the end points of the intervals and at least one value which is interior to each interval.

c) Input data should be selected which lie outside the domains of input variables to check that the program guards against such values.

d) Data must also be selected to generate the output over the domains of the program's output variables. This is carried out using the same guidelines described in a,b,and c above for the input variables.

e) Tests should include certain values which have special mathematical properties. These values may or may not lie at the endpoints of variable domain intervals or correspond to one of the discrete values in a small finite domain.

2- Arrays and Vectors

Arrays have dimensions in addition to the values of their elements. Functional testing requires the construction of tests in which each of the possible special sets of values for an array's dimensions is combined with each of the special sets of values for its elements. The set of allowable values for the dimensions of the arrays in the domain of an array variable should be completely specified. This is in addition to the specifications of the sets of allowable values for the individual elements of arrays.

The values of the elements of an array form a set that can

be treated either as a single entry or a collection of values of individual variables, all depends on program I/O specifications.

3- Array Substructures

In many applications, particular columns or rows or other substructures of an array have a functional identity of their own. It is necessary in functional testing to consider the value of an array taken as a whole, the values of individual array elements, and also the values of array substructures such as columns and rows.

4- Combinations of Test Values

If a program has an n input variables, each of which can take on k special test values, then there are k|n possible combination of test values. This should be tremendous in case of n is large. In order to avoid this combinational explosion, we have to identify small "functionally related" subsets of variables and to require that tests be constructed which include all combinations of special values of the variables in each subset. An example of functionally related variables is the variables which appear in the same assignment statement or branch predicates.

Functional testing requires the identification of, and then the testing of all the functions implemented in a program [16,17].

4-1-2 Function identification.

One of the drawbacks to functional testing is the necessity of having to identify the functions to be tested. The functions which are implemented in a program can be identified in several ways. If the life cycle approach to software development is used, then requirements and design documents can be used to identify both the functions computed by the system as a whole as well as intermediate level functions. In addition to these functions, systems may also contain implicit functions, and the program code must be examined to identify also these functions.

There are two ways to identify functions from the design documents, procedural cohesion and sequential cohesion [34]. Procedural cohesion depends on the control flow of the program. All the statements in a program path, for example, are bound by procedural cohesion. Procedural cohesion is a weak way of identifying functions and the sequential cohesion, which depends on data flow rather than control flow, is more powerful. Sequential cohesion between objects a,b, and c, for example, is achieved when object a generates data which are used by object b, and object b generates data which are used by object c.

Implicit functions can be detected using the coverage metrics. Coverage metrics are ways of measuring how much of a program has been executed during a sequence of tests. Functional testing requires that all combinations of program statements that correspond to a program function be tested, and not only once, but over fault revealing functional test data. Since every branch or statement is part of at least one function, then functional testing implies branch or statement testing [18],[25]. However, some program paths may not correspond to a particular program function, therefore functional testing does not imply path testing schemes [25],[26].

4-1-3 Testing functions.

Once the functions to be tested have been identified, it is necessary to test them over fault revealing test data. Test selection rules will be discussed for three kinds of forms : algebraic expressions, conditionals, and iteratives. Algebraic expressions are further subdivided into arithmetic expressions and arithmetic relations.

1- Arithmetic Expressions: Fault in arithmetic expressions can be detected by testing it on data which cause it to evaluate to a nonzero value. Any fault in the arithmetic expression can be detected using data which are selected randomly.

2- Arithmetic Relations: Suppose that rel is an arithmetic relation of the form exp op 0 , where exp is an arithmetic expression and op is <,<=,>,>=,= or #. Let @ stand for the smallest absolute value that exp can take on. Construct tests which cause exp to take on the values -@,0, and +@. On at least one of these tests rel will return a wrong value than it is supposed to return.

3- Conditional Forms: Suppose that F is in the form

if B then F1 else F2

Any fault in the boolean B will be reflected on F. If B is an arithmetic relation, then the rules for testing the form will be closely related to those for testing arithmetic relations.

4- Iterative Forms: Incorrect while-loop iterative forms result in the loop being executed the wrong number of times, and are therefore the result of incorrect loop continuation booleans. If the loop booleans are arithmetic relations, then the test rules for relations can be used for testing iterative forms in the same way that they were used for testing conditional forms.

4-1-4 Application of functional testing.

Data flow analysis can be used to try to automatically recognize functions and ensure that they have been tested. Functional testing will still require the examination of intermediate values for functions that correspond to parts of programs, and it is difficult to see how this process could be completely automated. If only the output from the program as a whole is analyzed, then the functional testing process can be almost completely automated, but at the cost of not finding functional faults whose effect is masked by code which is executed after a function ends.

4-2 Functional Analysis Technique

As stated before, there are two types of compositions for the functions: forms and structures. In structural synthesis, the information available about the correct program may consist of knowledge of correct function sequences rather than the input-output information usually associated with forms. The structural synthesis uses graphs to model interacting function sequences. There is a corresponding association between trace oracles, interface oracles, error detection, and structural synthesis. Functional analysis depends on the availability of oracles which

can be used to determine if a sequence of functions is correct.

Functional analysis involves the examination of sequences of functional transformations that occur in programs, in order to determine if the functional application structure of the program is correct. Simple faults, like a missing function application, can be detected in this way.

In practice, many functions do not correspond to explicitely identified pieces of code. Programs are thought of in terms of the types of data they produce and use rather than in terms of function applications. Function identification in this case consists of the identification of different kinds of data and of transformation between program states.

A type of data consists of axioms which characterize the properties of the operations which can be performed on the type. Requirements and general design documents are sources for information about data types. The recognition of types of data from detailed designs and programs may be done from explicit type declarations.

Functional analysis is used to determine if the sequences of functions or data types and states that can occur in an execution of a program are correct. Functional analysis can be done manually and can be considered to be a form of code reading.

4-3 The Revealing Subdomains Method [31]

The aim of this method is to expose the presence of certain specified errors or demonstrating that these errors do not occur. This goal is achieved by partitioning the problem's input domain in an intelligent way using both program dependent and program independent information which include the problem's specifications, the algorithm used, and the input/output data structures. In other words, this method combines the black box and white box methods to derive a partition of a function's input domain into smaller, and more manageable revealing subdomains. A revealing subdomain contains elements that are either all processed correctly or all processed incorrectly. Once such a subdomain has been identified, executing the program on a single one of its elements is sufficient to test the entire subdomain for the existence of a specific error or not.

The tester first creates a problem partition from the specification, by looking for classes of inputs that should be treated the same way by the program. The next step is to create a path partition, whose equivalence classes contain inputs that actually are treated the same way by the program. The partition used for functional testing is then created by intersecting the two partitions, creating a set of equivalence classes whose elements both should be and are treated the same way by the program. A test set is built by choosing one element from each of the testing partition's classes.

The main difficulty in the implementation of the revealing domain method is that there are no formal or systematic guidelines for creating the problem partition. The category-partition method [27] could be used to help create the problem partition since it provides a systematic approach to creating test set on the basis of the specification.

V- SUMMARY

Software testing is one of the most important and complex problems. There are two main approaches to software testing; functional testing and structural testing. This paper concentrates on the functional testing approach.

The goal of functional testing is to find discrepancies between the actual behavior of the implemented system and the desired behavior as described in the system's functional specification. The method used to represent the specification helps to discover incompleteness and inconsistency in it. The use of formal models to represent the specification reduces the number of errors that may remain in the specification. Some methods for representing the specifications were discussed above, along with the methods of testing the specifications. Functional tests can be derived from the software's specifications, from design information, or from the code itself. Code-based tests relate to the structure, control flow, and data flow of the software. Design-based tests relate to the programming abstractions, data structures, and algorithms used to construct the software. Specification-based tests relate directly to what the software is supposed to do, and therefore are probably the most appealing type of functional tests.

Specification-based testing is achieved using the black box strategy. In black box testing [25], the tester is completely unconcerned about the internal behavior and structure of the program. He is only interested in finding circumstances in which the program does not behave according to its specifications. Test data are derived only from the specifications without taking advantage of the internal structure of the program.

Black box testing could be done exhaustively or selectively. Exhaustive testing of a program's input domain is a process not necessarily guaranteed to terminate [9]. There are some programs which have infinite input domains, so exhaustive testing can never be completed. A standard approach for generating specification-based functional tests is to partition the input domain of a program into a finite number of equivalence classes such that a test of a representative of each class will test the entire class and hence the equivalent of exhaustive testing of the input domain can be performed. Several black box methods which use this approach are described in this paper. The appeal of black box testing lies in its echonomy over methods which examine the code, and in the fact that less technical expertise is required. Furthermore, consumers may want to perform acceptance sampling on software in situations where source code is withheld to protect a copyright.

However, there are many errors which could not be detected using the black box technique. This is due to the existence of functions in the code which are not mentioned explicitely in the specifications. These functions may contain errors, and thus test data which are dependent on the internal program code must be generated in addition to the tests generated according to the black box technique. The program dependent methods for tesy data generation were outlined in this paper. We should conclude that

neither the black box techniques nor the code dependent techniques are sufficient to detect all the errors in a software system. The two techniques are complementing rather than competing, and neither of them should be ignored.

REFERENCES

[1] T.Agerwala, "Putting Petri-nets to work," IEEE Comput. 12,12, Dec. 1979, pp. 85-94.
[2] B.Casey, and B.Taylor, "Writing requirements in English: A natural alternative," In Proceedings of the IEEE Software Engineering Standards Workshop (San Francisco, California, August), IEEE Press, Wash., D.C.,1981, pp 95-101.
[3] V.Chvalovsky, "Decision tables," Software Pract. Exper. 13, 423-429, 1983.
[4] A.M.DAVIS, "A comparison of techniques for the specification of external system behavior," Comm. of the ACM, vol 31, no. 9, Sept. 1988.
[5] A.Davis, and T.Rauscher, "Formal techniques and automatic processing to ensure correctness in requirements specifications," In Proceedings of the IEEE Specifications of Reliable Software Conference, April 1979.
[6] W.R.Elmendorf, "Functional analysis using cause-effect graphs," Proceedings of SHARE XLIII, SHARE,New York, 1974, 567-577.
[7] J.D.Gannon, P.Mc Mullin, and R.Hamlet, "Data abstraction implementation specification and testing," ACM Trans. Programming Lang. Syst., vol 3, no. 3, pp.211-223, July 1981.
[8] N.Gehani, "Specification: Formal and informal- A case study," Software Practice Experience, vol 12, 1982.
[9] J. B. Goodenough and S.L.Gerhart, "Toward a theory of test data selection," IEEE Trans. on Software Engineering, vol. SE-1, no. 2, June 1975.
[10] J.V.Guttag, "Notes on type abstraction (version 2)," IEEE Trans. Software Eng., vol SE-6, Jan. 1980.
[11] D.Harel,"Statecharts: A visual formalism for complex systems," Sci. Comput. Prog. 8, 1987.
[12] I.J.Hayes, "Specification directed module testing," IEEE Trans. on Software Eng., vol. SE-12, no. 1, Jan 1986.
[13] K.Heninger, "Specifying software requirements for complex systems: New techniques and their application," IEEE Trans. Software Eng., vol6, no. 1, Jan. 1980, pp 2-13.
[14] W.E.Howden, "Functional Program Testing," IEEE Transactions on Software Engineering, vol. 6, no. 2, March 1980.
[15] -------- , "Validation of Scientific Programs," ACM Computing Surveys, June 1982.
[16] --------, "A Functional Approach to Program Testing and Analysis," IEEE Trans. on Software Engineering, Vol. SE-12, No. 10, October 1986.
[17] --------, Functional Program Testing and Analysis. New York: McGraw-Hill, 1987.
[18] J.C.Huang, "An approach to program testing," ACM on Computing Surveys, Sept. 1975.
[19] P.Jalote, "Testing the completeness of specifications," IEEE on Software Engineering, vol. 15, no. 5, May 1989.
[20] H.Kawashima et al., "Functional specification of call processing by state transition diagram," IEEE Trans. Commun., vol 19, no. 5, Oct. 1971.
[21] R.A.Kemmerer, "Testing Formal Specifications to Detect Design Errors," IEEE Trans. on Software Engineering, vol. SE-11, no. I, Jan 1985.
[22] Michael Montalbano, "Decision Tables," Science Research Associates, Inc., Chicago, 1974.
[23] B.Moret, "Decision trees and diagrams," ACM Comput. Surv. 14, 4 (Dec. 1982), 593-623.
[24] D.A.Musser, "Abstract data type specification in the AFFIRM system," IEEE Trans. Software Eng., vol SE-6, Jan. 1980.
[25] G.J.Myers,The Art of Software Testing. New York: Wiley,1979.
[26] S.C.Ntafos and S.L.Hakimi, "On path cover problems in digraphs and applications," IEEE Trans. on Software Engineering, vol. SE-5, no. 5, Sept. 1979.
[27] T.J.Ostrand and M.J.Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests," Comm. of the ACM, Vol 31, No. 6, June 1988.
[28] J.Peterson, "Petri-nets," ACM Comput. Surv. 9,3 , sept. 1977, 223-252.
[29] C.Petri, "Kommunikation mit automation," Schriften des Reinsch Westfalischen Inst. fur Intrumentelle Mathematik an der Universitat Bonn, Bonn. West Germany, 1962.
[30] R.P.Roe and J.H.Rowland, "Some theory concerning certification of mathematical subroutines by black box testing," IEEE Trans. on Software Engineering, vol. SE-13, no. 6, June 1987.
[31] E.J.Weyuker and T.J.Ostrand, "Theories of program testing and the application of revealing subdomains," IEEE Trans. on Software Engineering, vol. SE-6, no.3, May 1980.
[32] V.Whitis, and W.Chiang, "A state machine development method for call processing software," Proceedings of the IEEE Electro 81 , April 1981.
[33] M.Yoeli, and Z.Barzilai, "Behavioural descriptions of communication switching systems using extended Petri-nets," Digital Processes 3,4 , 1977, 307-320.
[34] E.Yourdon and L.Constantine, Structured Design. Englewood Cliffs, NJ: Prentice-Hall,1979.
[35] A.A.Omar, F.A.Mohammed, "Structural Testing of Programs," ACM press, Software Engineering Notes, Vol 14, No 2, Apr 1989.