Test Plan Methodology

Reza Pazirandeh
Principle [sic!] Member of Technical Staff
Citicorp TTI, 3100 Ocean Park, Santa Monica CA 90405

1. Introduction

Although a software development staff may spend a substantial amount of time testing software, there is still a lack of planning and rigour that is required if the task of testing is to be taken seriously enough. Usually, the testing that is performed on a piece of software is made up of randomly selected test cases which are obvious and the test data (data base and transaction) used for these cases are not well thought out and devised. There is a tendency to think that volume testing will uncover most of the errors which is not true. And, of course, the ever famous statement "all known bugs have been resolved" is often used to indicate the completion of the unit testing.

Part of the reason for not treating the task of testing with more rigour is that testing software is not an easy task. And part of the difficulty is due to the fact that it is not easy to define a "how to" for testing.

In light of the previous paragraphs, this paper provides a non-arbitrary and practical approach to testing and test case defintion. Of course, the task of testing must begin at system definition phase and must continue thru the design phase. Therefore, this paper assumes that the document against which the test plan is developed is itself reliable (i.e., tested).

Let us also be aware that to assure software reliability in any software shop, in addition to a rigorous test methodology, there has to also exist an implicit or explicit statement of policy regarding the organiztion's goal in testing. The test goal defines the degree of software reliability that developers must strive for and it is a function of the criticality of the software being tested.

2. Basic Approach To Testing

This section will first define what rigorous and systematic testing involves and then it will provide some rather straightforward steps on how to achieve it. The reader should note that the techniques presented here can be applied at any level of testing. As such, a program (i.e., code) and program specification are to be viewed interchangeably in this context. The examples that are provided, although COBOL-like, should also be viewed as a piece of logic or specification.

2.1 Path Testing

The test of a piece of software in general consists of testing all of the conditions contained in that piece of software. Each of these conditions requires a specific set of data to test.

However, it is quite impractical and inefficient to test each condition separately since the number of test runs would be huge and each test run would be executing some of the already tested conditions redundantly. In other words, the test of many conditions in a program can be performed thru a single test (run). It turns out that such a set of conditions constitute a "path" in a program. So, the task of testing boils down to testing all the logic or processing paths in a piece of software. Upon careful consideration it becomes apparent that many paths in a program are redundant (non unique): That is, they do not test any part of the program which has not been tested before. In fact, the number of test runs that completely test the logic in a piece of software is equal to the number of the unique paths contained therein.

As mentioned above, the concept of processing paths is a valid one whether one is dealing with a piece of code, some specifications written in pseudo code, or even a narrative form of specification.

The next subsection describes why some paths are unique while others are not.

2.2 Dependent VS. Independent Condition

The uniqueness or non-uniqueness of processing paths in a program is determined by dependent and independent conditions:

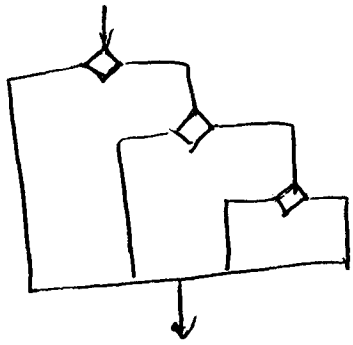
Two conditions are dependent if the result or the execution of one condition depends on the other.

Conversely, two conditions are independent if the result or the execution of one does not depend on the other.

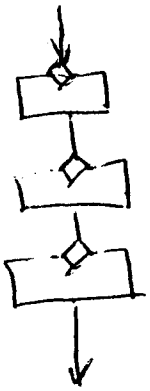
For example, consider the following two pieces of logic and their condition chart:

```
1)  INPUT VAR-1.

    IF VAR-1 = 1
        SET COLOR TO WHITE
    ELSE
    IF VAR-1 = 2
        SET COLOR TO BLUE
    ELSE
    IF VAR-1 = 3
```



```
        SET COLOR TO RED.
```



```
2)  INPUT VAR-1, VAR-2, VAR-3.

    IF VAR-1 = 1
       SET COLOR TO WHITE.

    IF VAR-2 = 1
       SET SEX TO MALE.

    IF VAR-3 = 1
       SET STATUS TO MARRIED.
```

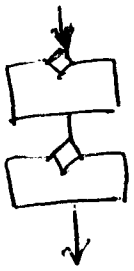
The first example illustrates dependent conditions; it requires four test runs to be fully tested:

```
first test : VAR-1 = 1
second test: VAR-1 = 2
third test : VAR-1 = 3
fourth test: VAR-1 = not 1,2, or 3
```

The second example illustrates independent conditions; it requires two test runs to be fully tested:

```
first test : VAR-1 = 1, VAR-2 = 1 VAR-3 = 1
second test: VAR-1 = NOT 1, VAR-2 = NOT 1, VAR-3 = NOT 1
```

The following is another form of dependent conditions. Here, four test runs are needed to fully test the logic.



```
3)  INPUT VAR-1, VAR-2.

    IF VAR-1 = 1
       SET VAR-2 TO 1.

   IF VAR-2 = 1
       SET COLOR TO BLACK.
```

```
First test  : VAR-1 = 1, VAR-2 = 1
Second test : VAR-1 = not 1, VAR-2 = not 1
Third test  : VAR-1 = 1, VAR-2 = not 1
Fourth test : VAR-1 = not 1, VAR-2 = 1
```

The conclusion is that dependent conditions can not be tested together (i.e., they can not share the same path) whereas the independent conditions can be tested together.

Understanding these two types of conditions is essential in the identification of paths in a program.

Let us now proceed with the identification of the unique paths in a program. The following sections spell out how this can be done.

2.3 Determining Unique Paths

This section will provide a cookbook-like series of steps that will identify the unique paths.

2.3.1 Identify And Number Conditions

Walk thru the program to be tested, underlining and numbering each condition. Conditions in a program include all IF's, all ELSE's (whether or not they are explicitly coded), and all loop variables. See the example in figure 2-4.

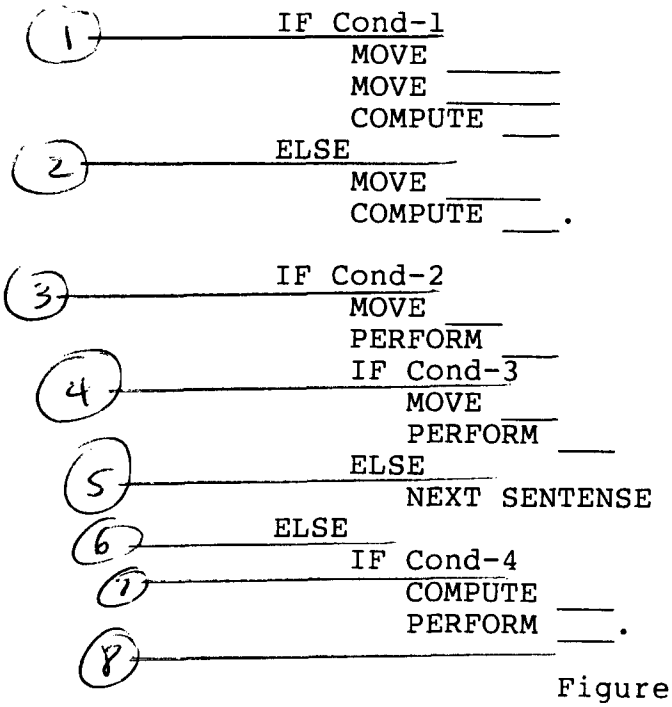


Figure 2-4

The numbering should be done according to the physical occurance of conditions and not according to the flow of the program. The reason for this is to make sure the numbers are physically in sequence and can be easily located later on.

2.3.2 Draw A Condition Graph

The easiest method of identifying the paths in a program is to draw a condition flow chart of the program.

A condition flow chart is basically like a traditional flow chart; figure 2-5 serves as an example. Diamond shape boxes are used to indicate decision points and rectangular boxes show the processing. In this type of chart, we are primarily interested in showing the decision points. So, even if the processing boxes are not drawn, the chart will be just as effective. The processing boxes can be used, where needed, to serve as markers to indicate what logic is being performed.

As you draw the chart, you can also put the condition numbers on the appropriate branches of the diagram. See figure 2-6.

The diagram may be hand-drawn with a pencil on a computer output. It should be good enough to identify the paths. Do not spend any more time than necessary drawing this diagram.

It is worthwhile to know that the task of drawing the condition chart by itself serves as a good review and walkthru of the material to be tested.

Once the diagram is completed, the task of identifying/marking the paths begins.

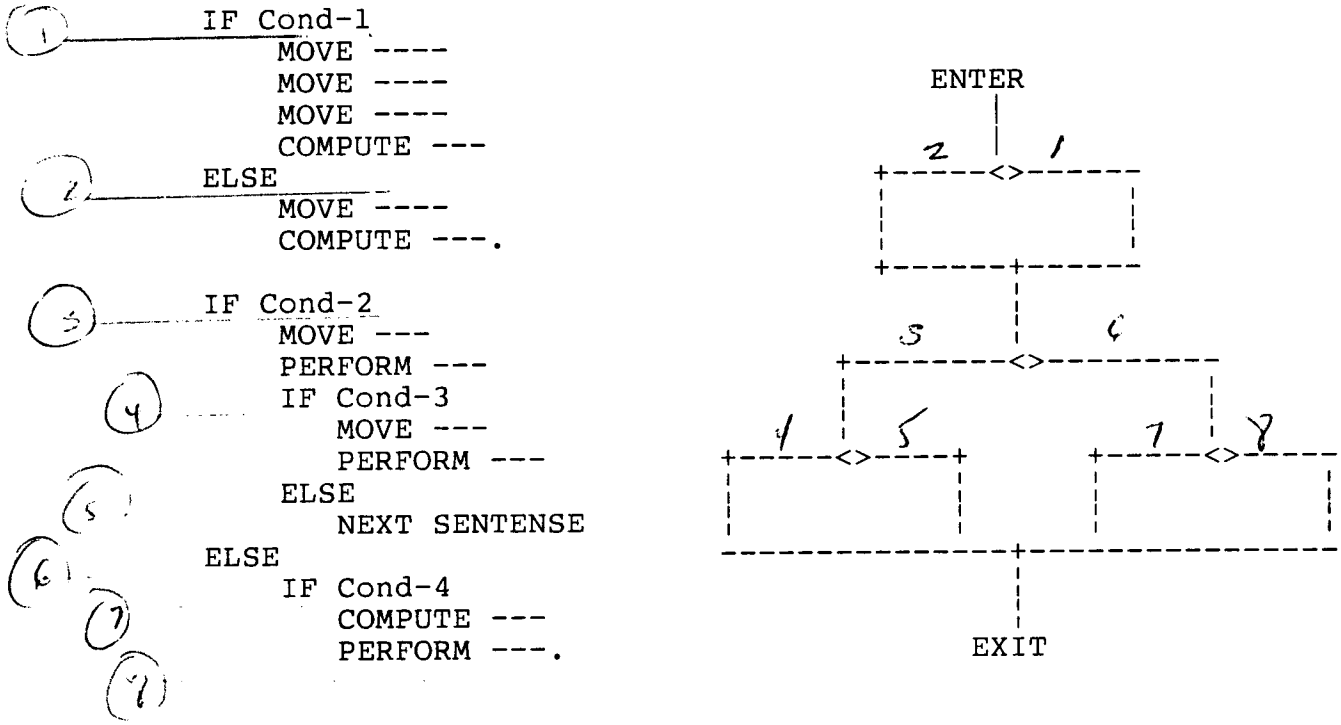
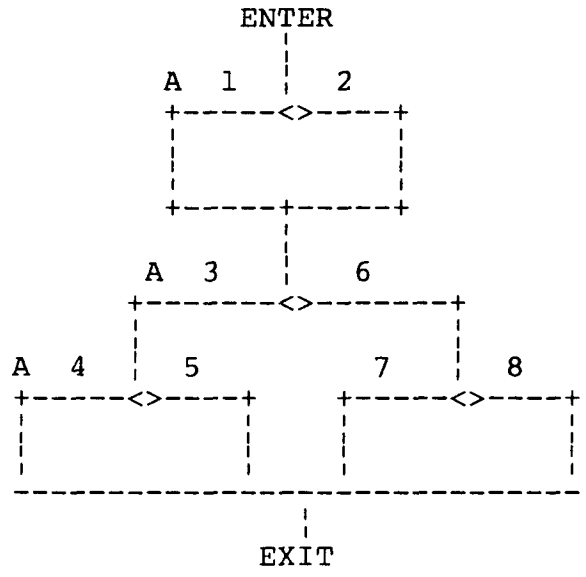


Figure 2-5

2.3.3 Identify (Mark) The Independent Paths

Although not essential, begin traversing the most commonly used paths first, marking each section (branch) of the path with a letter. This letter will help keep track of which path has been tranversed and it will also serve to identify a test case. As you see in figure 2-6 first path A and then path B are traversed and marked.

1. Path A is marked



2. Path B is marked

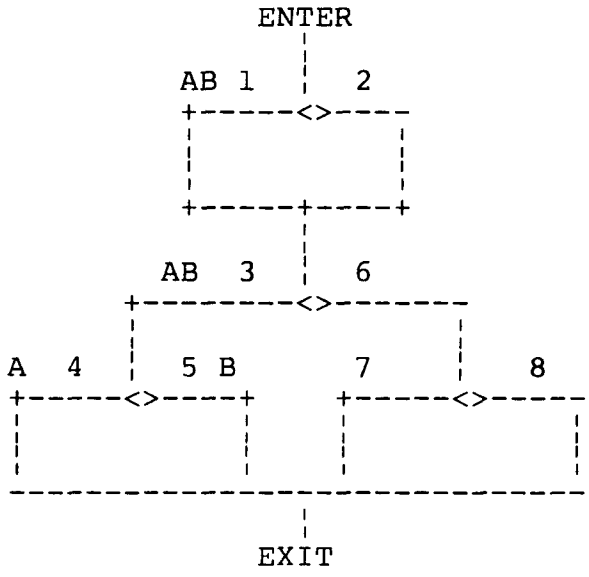


Figure 2-6

If you run out of letters, you may use A', B', and so on. However, if a module test plan is being developed, you should also question the size of a single module which requires more than 26 test cases to test.

In order To traverse only the unique paths, do not traverse a path whose branches have ALL been marked. Bear in mind, however, that dependent conditions can not share the same path.

As an example, in figure 2-7 path C will not traverse paths A, or B. In other words, once all branches of a condition have been tested, they need not be tested again.

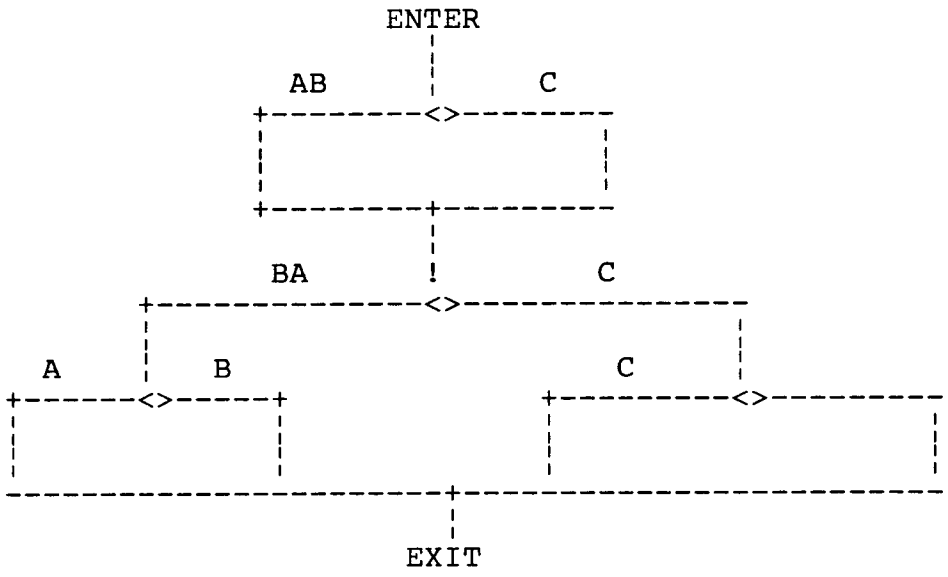


Figure 2-7

The recognition of dependent conditions (particularly, the type illustrated by example 3 in section 2.2) in a program is very important and is one area in this approach that requires the programmer/test plan writer to use some analysis to make certain that no path is left untested. See section 2.2 for more explanation on independent vs. dependent conditions.

2.4 Preparing The Required Test Data

Once logic paths are identified, determining the required data to test the paths is not too difficult.

First of all, the programmer (tester) must have the contents (dump) of the relevant area(s) or record(s) of the data base. If there are no data base records already defined or the data base is inadequate, then this task will also include defining records to be loaded into the data base.

Follow each path, tracing the conditions involved in the path. Remember that the conditions are numbered and easy to find. You now have to document the required input (transaction) data and the test data base data.

The required input and the expected results are recorded in the Test Plan form. The required data base data should be documented as appropriate for your organization.

3. Function/System Testing

Let us define the purpose of function or system testing as follows:

1. To validate the system behavior from the point of view of the us
2. To validate the interfaces between modules and programs.

The difference between these two purposes is a matter of degree of testing. That is, to satisfy the second purpose, the tester needs to identify and test the functions that test the interfaces whereas the first purpose will be satisfied only if all functions are identified and tested.

3.1 Determining Test Cases For Function Testing

In order to test the "functions" of a system, we must first define what a function is. This definition is important since our test philosophy and strategy will depend on how accurately and rigorously we define this term.

Let us define the functions performed by a system to be the set of all the variations of all the transactions performed by that system. The variations of a transaction result from the various business or processing rules or options operating on that transaction. Hence, a function is a particular variation of a transaction. The following examples will help clarify this concept.

Suppose a piece of software (i.e., a module or a program) translates a Gregorian date to a Julian date. Loosely speaking, the function performed by this piece of software is the stated date conversion. But, in terms of testing, the functions performed by this piece of software consist of all variations of this function which include incorrect gregorian dates, dates with leap year, and so on and so forth.

Upon closer scrutiny it becomes evident that each variation (of a transaction) is in fact a processing path that starts from an entery point and ends at an exit point in the system. These paths may be totally contained within one module or they may expand a module limit. So, to identify the set of all functions performed by a system, we must identify all processing paths in the system. We can for the moment ignore the processing paths that end in system type failures (data base fatal errors, bad data transmission, etc.). Although, the user would want to verify the system behaviour in the case of a system failure as well.

Identifying functions can be done once the requirements or the functional specifications are defined. For organizations using the structured approach it would be either at the end of the logical or physical model. Naturally, for the purposes of development (testing being part of it), a large system is normally broken down into subsystems. So, the function testing will initially take place at a subsystem level.

As stated before, the steps in defining the test cases or paths for function testing are exactly those used for unit testing.

4. Reasonable Testing

As stated in the introduction, the goal of testing dictates how well a piece of software has to be tested. That notwithstanding, considering the complexity and size of some (application) programs, the number of paths or test cases would be huge and maybe unmanageable. The alternative would be to make sure that a program is "reasonably" tested. The question is how to define a reasonably tested program. It is fair to state that to reasonably test a program, at least the following must be tested:

a) all business conditions (rules)
This would include all validations, editting, and so on.

b) all architectural rules
This would include duplicate detection, recovery, replay, automatic reversals, and so on.

c) data base return codes (statuses)
This is the case where, due to the uniformity of some return codes, not all instances of the same return code has to be tested.

It is also reasonable to state that the paths that are more likely to be executed in a program ought to be tested first. Hence, the paths that make up the normal processing in a program should be tested before the ones ending in an exception or error condition.

5. Required Test Tools

For a rigorous (unit) test, as a minimum, the following test tools are essential:

1. Drivers and stubs;

2. A tool to get the formatted dump of specific records on the data base so that the required test data base is created; this tool is also used to verify the changes to the data base after testing;

3. A tool to generate the input data to the driver;

4. A tool to simulate some data base return codes so that data base error conditions are tested;

6. Summary

The task of testing software is a difficult and an important one. it is all too often performed in a non-rigorous and arbitrary manne This paper offers a method by which the task is made somewhat less difficult and certainly less arbitrary and quite rigorous. Th methodology has been applied and proven to be quite practical. The steps, presented in this document, to define test cases may be modified as users see fit. The rigour of the basic approach, howev should not be altered.