INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH & DEVELOPMENT

ISSN: 2278 – 0211 (Online)

An Efficient Software Testing By Diminishing No Of Test Executions

Pradeep Udupa
Ass.professor M.E.S Engineering College Kerala, India

Abstract:

Software Testing is the process of executing a program or system with the intent of finding errors or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a fixed (and reasonably small) set of ways. By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and-tear -- generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects -- or bugs -- will be buried in and remain latent until activation.

In this study execution code& algorithm is developed To optimize The testing efficiency by fetching test inputs from the database which will reduce time, effort & no of executions. here an efficient code is developed to fetch test data from database and to fetch the data from data table to increase the execution speed decrease the effort and increase testing efficiency

And our main objective is

1) To reduce number of all test cases. Generally, the larger the input domain, the more exhaustive the testing would be. To avoid this problem, a minimum set of test cases needs to be created using an algorithm to select a subset that represents the entire input domain. In addition, when test cases are larger, the testing itself would take longer to run, particularly for regression testing where every change in the program demands repeat testing. Therefore, reducing number of the test cases does have advantage in efficiency.

2) To find the technique for automatic generation of test cases. To reduce the high cost of manual software testing while increasing reliability of the testing Processes, IT researchers and technicians have found methods to automate the reduction process. With the automatic process, the cost of software development could be significantly reduced.

3) To keep a minimum number of test runs. The best technique must be able to generate test cases from only one example test run.

1.Introduction

Software testing is a process of *verifying* and *validating* that a software application or program. Software testing

- Meets the business and technical requirements that guided its design and development, and
- Works as expected.

Software testing also identifies important defects, flaws, or errors in the application code that must be fixed. The modifier "important" in the previous sentence is, well, important because defects must be categorized by severity. Software testing also identifies important defects, flaws, or errors in the application code that must be fixed. The modifier "important" in the previous sentence is, well, important because defects must be categorized by severity.

During test planning we decide what an important defect is by reviewing the requirements and design documents with an eye towards answering the question "Important to whom?" Generally speaking, an important defect is one that from the customer's perspective affects the usability or functionality of the application. Using colors for a traffic lighting scheme in a desktop dashboard may be a no-brainer during requirements definition and easily.

2.Need & Scope Of The Study

- To increase Testing Efficiency
- Reduce No of Execution & Execution Time &Effort

2.1.Software-Testing Techniques

With finding errors as the primary objective of software testing, higher probability of detecting defects has become the defining quality of an effective test. Computer-based systems, Which are known to offer testers with diversity of testing methods and, hence, enhance probability of detection, are therefore recommended as the most efficient tools currently Available [4], [6].

- Path testing: aims to inspect the validity of selected Paths without the need for testing every possible path (as Required in Structural testing). The test is preferable when the Number of all available paths is so great that testing all of Them become impractical [1].

- Independent program paths: an independent program 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:

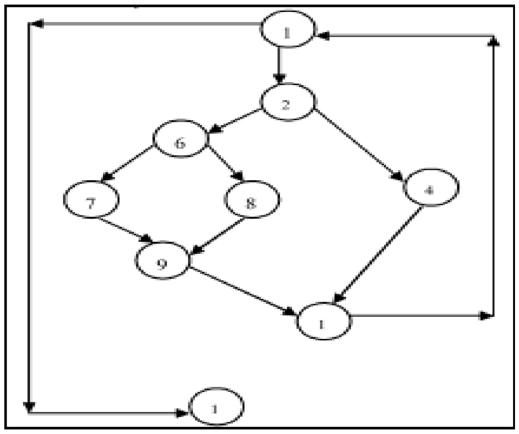

Figure 1

2.2.Cyclomatic Complexity

The cycloma tic complexity gives a quantitative measure of The logical complexity. This value gives the number of Independent paths in the basis set and an upper bound for the Number of tests to ensure that each statement is executed at Least once. An independent path is any path through program That is A new condition (i.e. new edge) [1]. Example

- Number of regions of flow graph
- Edges-nodes+2
- Predicate node+1.

2.3.Deriving Test Cases

- 1.Using the design or code, draw the corresponding flow graph
- 2.Determine the cyclomatic complexity of the flow graph
- 3.Determine a basis set if independent paths.
- 4. Prepare test cases that will force execution of each path in The basis test.

Independent paths: Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11 Path 3: 1-2-3-6-8-9-10-1-11 Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 Is not considered to be an Independent

Already specified paths and does not traverse any new edges. Paths 1, 2, 3, and 4 constitute a basis set for the flow graph in Figure 2.1. That is, if tests can be designed to force execution Of these paths (2, 4, 6, 7), every statement in the program is Guaranteed to be executed at least one time, and every Condition will have been executed on its true and false sides. It Should

Number of different basis sets can be derived for a given Procedural design.

B.Dynamic Domain Reduction (DDR): DDR is the technique that creates a set of values that Executes a specific path. It transforms source code to a Control Flow Graph (CFG). A CFG is a directed graph that Represents the control structure of the program. Each node in The graph is a basic block, a junction, or a decision node [8].

2.4.Test Case Generation Technique

DDR uses the GetSplit algorithm to find a split point to divide the domain.

The GetSplit algorithm is as follows:

Algorithm

Getsplit (LeftDom, RightDom, SrchIndx) Precondition

LeftDom and RightDom are initialized appropriately And SrchIndx is one more than the last time Getsplit was called with these domains for this expression.

Split value = (LeftDom.Bot AND RightDom.Bot) and Split value =(LeftDom.Top AND RightDom.Top) Input

LeftDom: Left expr’s domain with Bot and Top values RightDom: right expr’s domain with Bot and Top values Output

Split–a value the divides a domain of values into two sub domains.

```
BEGIN
        --  Compute the current search point
        --  srchPt = (1/2, 1/4, 3/4, 1/8, 3/8 …)
        --  Try to equally split the left and right expression's domains.
IF (LeftDom.Bot>= RightDom.Bot AND LeftDom.Top< = RightDom.Top)
    • Split=(LeftDom.Top -LeftDom.Bot)*srchPt + LeftDom.Bot ELSE IF
      (LeftDom.Bot<= RightDom.Bot AND LeftDom.Top >= RightDom.Top)
    • Split=(RightDom.Top -RightDom.Bot)*srchPt + RightDom.Bot
```

- ELSE IF (LeftDom.Bo>t= RightDom.Bot AND LeftDom.Top >= RightDom.Top)
- Split=(RightDom.Top - LeftDom.Bot)*srchPt + LeftDom.Bot
- ELSE -- LeftDom.Bot<= RightDom.Bot AND LeftDom.Top< = RightDom.Top
- Split=(LeftDom.Top - RightDom.Bot)*srchPt + RightDom.Bot

2.5.End If Return Split End Getsplit

In the dynamic domain reduction procedure, loops are handled dynamically instead of finding all possible paths.

The procedure exits the loop and continues traversing the path on the node after the loop. This eliminates the need for loop unrolling, which allows more realistic programs to be handled. [2][7]

3.Propesd Technique

3.1.Objectives

- In this study execution code& algorithm is developed to Optimize the testing efficiency by fetching test inputs from The database which will reduce time, effort & no of Executions. Here an efficient code is developed to Fetch test data from database and to fetch the data from Data table to increase the execution speed decrease the effort and increase testing efficiency.
- To develop execution code for fetching the data from data table which is used to retrieve test case data directly from Data table without expecting data to be inputted By implementing this procedure. It will automate the execution process And cost & effort involved in doing manual work Will be diminished.
- Developing execution code to calculate overall time required to find an efficiency of execution Example test run. in terms of speed In this paper, a new algorithm is used to meet the above-mentioned objectives, using the following steps.

3.2.Test Cases Generation Technique

There are four steps to generate test cases:

- Finding all possible constraints from start to finish nodes. A Constraint is a pair of algebraic expressions which dictate Conditions of variables between start and finish nodes (>, >=, <, <=, ==, !=)
- Identifying the variables with maximum and minimum Values in the path, if any. Using conditions dictated by the Constraints, two variables, one with maximum value and the Other with minimum value, can be identified. To reduce the Test cases, the maximum variable would be set at the highest Value within its range, while assigning the minimum variable At the lowest possible value of its range.
- Finding constant values in the path, if any. When constant Values can be found for any variable in the path, the values Would then be assigned to the given variables at each node.
- Using all of the above-mentioned values to create a table to Present all possible test cases.

3.3.Expected Results

- Using the methodology, the new algorithm would have the Following characteristics:
- Number of test cases. The number of test cases is smaller since each variable has a fixed value, either as maximum,Minimum or constant values.
- Automatic test cases generation. The test cases can be automatically generated with the reduction process.
- Less time to test run. A single generation of test cases Reduces the time of test run and compilation.

4.Evaluation

A comparative evaluation has been made between the Proposed Techniques, the Existing Technique (Get Split Algorithm technique). The following areas are used to compare With existing techniques:

- Number of test cases
- Reduction percentage of test cases

- Compilation time

The evaluation is described using two examples

4.1.Example

The function value takes three marks as input such as mark1, Mark2, mark3 and returns some total mark for student depending upon the performance.

- Source code

```
int  Result(mark1,mark2,mark3)
{ int total; Total=0;
 If(mark1<mark2)
  {
   Mark3=mark3+5;
If (mark1<mark3) sum=mark1+10; Else Total=mark1+5;
Else{mark3=mark3+10;
       sum=mark1+mark2+mark3;
       }
  return  (sum);
}
```

5.Proposed Technique For Increasing Efficiency

In this study A Test Execution technique is adopted for making test case efficient by feeding data from database directly Advantage of this is once we store test data in database we can use it for many Test run and save the overall time & Effort

And for this purpose a piece of code is developed as follows

```
Option explicit Dim con,rs
Set con=createobject(“adodb.connection”)
 Set rs=createobject(“adodb.recordset”) con.provider=(“Microsoft.jet.oledb.4.0”)
con.open “c:\document anssettings\mydocuments\test.mdb”
rs.open “select * from test”,con do until rs.eof=true invokeapplication “c;\program files\test.exe”
 Diolog(“Result”).Activate
Diolog(“Result”).WinEdit(“mark1:”).Set rs.fields(“m1”)
Diolog(“Result”).WinEdit(“mark2:”).Setrs.fields(“m2”)
Diolog("Result").WinEdit("mark3:").Setrs.fields("m3")
 Window("testapplication").close Rs.movenext
 loop
```

Another approach used is first feeding all Test case data into data table and then using it from the

Data table of excel by using following proposed code

EXECUTION Code Used For Calculating Test Case Time =>

```
Service.startTransaction
Dim m1,m2,m3,ST,ET,TT
ST=timer()
ET=timer()
m1=datatable("mark1",1)
m2=datatable("mark2",1)
m3=datatable("mark3",1)
invokeapplication"c;\programfiles\test.exe"
Diolog("Result").Activate Diolog("Result").WinEdit("mark1:").Set m1
 Diolog("Result").WinEdit("mark2:").Set m2
 Diolog("Result").WinEdit("mark3:").Set m3
Window("testapplication").close
T= Result(m1,m2,m3)
TT=ST-ET
msgbox("Totaltime:="&TT)
Service.EndTransaction
```

Contributing in increasing in efficiency & Execution code for calculating test case

Time is implemented to find total time required

5.2.Control Flow Graph

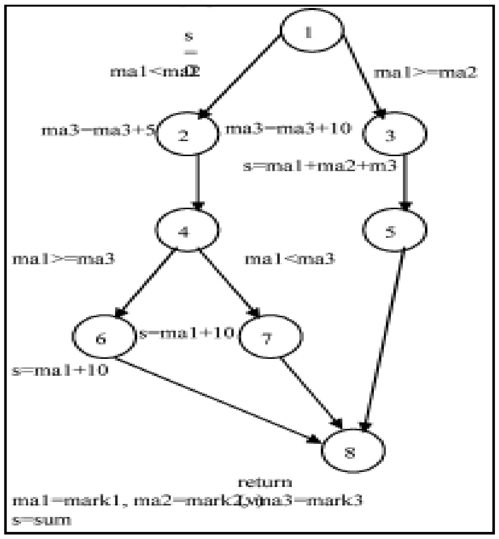

Figure 2

5.3.No Of Independent Path

- Path1: 1, 2,4,6,8
- Path2: 1, 2,4,7,8 Path3: 1,3,5,8

5.4.Evaluation Result For Proposed Method

Assume that the path 1-2-4- 6-8 is elected and the initial domains of the input variables are <0 to 30>, <10 to 50>, <0 to 40> A step follows:

- Finding all possible constraints from start to finish nodes. Ma1 < ma2, ma1> = ma3
- Find minimum values in the path, if any.

From the above conditions, it is possible to identify ma3 as the Variable with the minimum value and ma2 as the variable with Maximum value. In accordance to the finding, a value of zero, The lowest value within the range of variable ma3, can then be Assigned to ma3 while the value of ma2 can be set at 50, the Highest value of the variable.

- Finding constant values in the path, if any. Ma1 constant Value for variable ma3found on Node 2 of the path has been Used to replace the fix value of ma3 (10) at the node.
- Using all of the above-mentioned values to create a table to Present all possible test

Cases.ma1 value is 10..30, ma2 as the Variable with maximum value = 50, ma3 as the

Variable with The minimum value = 10.

Reduced test cases:

Variables	**All test cases**
ma1 ma2 ma3	10 50 10
	11 50 10
	12 50 10
	13 50 10
	14 50 10
	15 50 10
	16 50 10
	17 50 10
	18 50 10
	19 50 10
	20 50 10
	21 50 10
	22 50 10
	23 50 10
	24 50 10
	25 50 10
	26 50 10
	27 50 10
	28 50 10
	29 50 10
	30 50 10
Total	21

5.5.Evaluation Result For Existing Method

Assume that the path 1-2-4- 6-8 is elected and the initial Domains of the input variables are

<0 to 30>, <10 to 50>, <0 to 40>

A step follows:

- Finding all possible constraints from start to finish nodes. ma1<ma2, ma1>=ma3, ma3=10
- Calculate split value and splitting Intervals for all constraints.
  - For constraints ma1<ma2

Splitting values are 8, 10, 11, 13, 15. We choose the split Value=15from above mentioned values. Then divided the input domain into two intervals

No	**Ma1**	**Ma2**
1	0 to 15	10 to 30
2	16 to 30	31 to 50

Table 1

From the constraints ma1 is lesser than ma2. Then choose the interval from constraints checking. The selected interval is

No	Ma1	Ma2
1	0 to 15	-
2	16 to 30	31 to 50

Table 2

For the second constraint ma1>=ma3.the split values

Are 7, 10, 11, 15, 17.We choose the split value=10 from above Mentioned values. Then divided the input domain into two Intervals

No	**Ma1**	**Ma2**
1	0 to 10	0 to 10
2	11 to 30	11 to 40

Table 3

From the constraints ma1 is Greater than equal to ma3 then choose the interval from constraints checking. The selected interval is

No	Ma1	Ma2	Ma3
1	0 to 10	0 to 10	16
2	11 to 30	-	-

Table 4

- Third constraint is ma3=16

No	Ma3
1	16

Table 5

From Table 2, Table4, Table 5, finally calculate all selected intervals

No	**Ma1**	**Ma2**	**Ma3**
1	0 to 10	-	16
2	11to 30	31 to 50	-

Table 6

6.Evaluation Results

Method/ area	Proposed Algorithm	Existing Algorithm
All possible test Cases	52111	52111
Reduced test cases	21	651
Saving (%)	99.95	98.75
Time of compilation	5.25	162.25

Table 7

Total possible test case came from number values on each variable 31*41*41

Saving (%) = 100-((100*Reduced Test Case)/All PossibleTest Case).

7.Analysis Graph

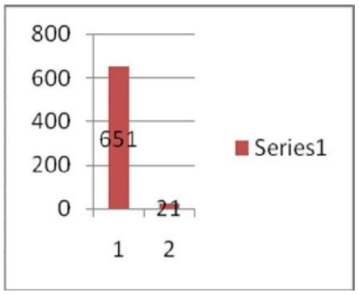

Figure 1: X-axis for algorithm, Y-axis for reduced test cases, 1- for proposed solution, 2- for existing solution

7.Conclusion

The new implemented technique has achieved higher reduction Percentage of the test cases by fetching data directly from the data table or D.B and running as many times as needed. Furthermore, because It retrieves test data directly from data table or D.B it takes less time Among the one existing technique. Based on the observation done,The proposed method can be considered a superior technique From all others available in current literatures. and it diminishes no of executions

The proposed Technique Lies In its requirement for Identification Of Fix values For All variables, either as Maximum, minimum or constant values

8.Reference

1. G. Rothermel, L. Li, and M. Burnett. Testing strategies for Form-based visual programs. In Proceedings of the 8thInternational Symposium on Software Reliability Engineering,Pages 96–107, Albuquerque, NM, November 1997.
2. G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In Proceedings of the International Conference on Software Maintenance, pages 179–188, August 1999.
3. G. Rothermel, Roland H. Untch, Chengyun Chu, and M. J.Harrold. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, 27(10):929–948, October 2001a.
4. Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher Dupuis, and Andrei Sheretov. A methodology for Testing Spreadsheets. ACM Transactions on Software Engineering and Methodology, 10(1):110–147, January 2001b.
5. Andrew Sears. Layout appropriateness: A metric for evaluating user interface widget layout. IEEE Transactions on Software Engineering, 19(7):707–719, 1993.
6. Andrew Sears. Layout appropriateness: Ametric for evaluating user interface widget layout. IEEE Transactions onSoftwareEngineering, 19(7):707–719, 1993.
7. Forrest Shull, Ioana Rus, and Victor Basili. Improving software inspections by using reading techniques. In Proceedings of the 23rd International Conference on Software Engineering, pp 726–727. IEEE Computer Society, 2001.
8. Ian Sommerville. Software Engineering. Addison-Wesley, 6th edition, August 2000.

9. Elaine Weyuker. Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering, (12): 1128–1138,December 1986.
10. James A. Whittaker. What is software testing? and why is it so hard? IEEE Software, 17(1):70–76, January/February 2000.