International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:1, 2008

# Improving the Effectiveness of Software Testing through Test Case Reduction

R. P. Mahapatra, and Jitendra Singh

***Abstract*—**This paper proposes a new technique for improving the efficiency of software testing, which is based on a conventional attempt to reduce test cases that have to be tested for any given software. The approach utilizes the advantage of Regression Testing where fewer test cases would lessen time consumption of the testing as a whole. The technique also offers a means to perform test case generation automatically. Compared to one of the techniques in the literature where the tester has no option but to perform the test case generation manually, the proposed technique provides a better option. As for the test cases reduction, the technique uses simple algebraic conditions to assign fixed values to variables (Maximum, minimum and constant variables). By doing this, the variables values would be limited within a definite range, resulting in fewer numbers of possible test cases to process. The technique can also be used in program loops and arrays.

***Keywords*—**Software Testing, Test Case Generation, Test Case Reduction

## I. Introduction

A strategy for software testing integrates software test case design methods into a well-planned series of steps that result in the successful construction of software .The strategy provides a road map that describes the step to be conducted as part of testing, when these steps are planned then undertaken, and how much effort, time, and resource will be required. Therefore, any testing strategy must incorporate test planning, test case design, test case execution and resultant data collection and evaluation [1].

Software testing is a process of inspecting the performance of software. The objective of software testing is to detect faults in the program and therefore, provide more assurance for customers on the quality of the software. As a part of any software development process, software testing represents an opportunity to deliver quality software and to substantially reduce development cost as much as 50% [1]. Many testers believe that software testing involves only detection of defective code. However, the testing itself extends beyond identification of the defects, and actually covers reporting and offering recommendations for appropriate actions.

Manuscript received December 2007.
R. P. Mahapatra is with SRM-IMT, Modinagar Campus of SRM University Chennai as an Assistant professor and HOD (CSE & IT) (e-mail: mahapatra.rp@gmail.com).
Jitendra Singh is as with SRM-IMT, Modinagar Campus of SRM University Chennai as Sr. Lecturer (CS Dept) (e-mail: Jitendra.jit@gmail.com).

The lack of understanding in this principle usually leads to incomplete testing work [5]. Software testing can severely suffer from planning that is not based on or does not adequately reflect actual environments where the software is operated. This problem usually occurs when testers are without any backup plan and or awareness of the environments themselves. Contrary to the common nature of programming, software testing places more emphasis on the design than the code. Therefore, testers who employ methodologies to detect defective code are often failing to find the real problems, which are usually embedded in the design of software. Another important aspect of software testing is that the number of the test cases that have a direct effect on the cost of testing, particularly that of Regression testing [1]. When tests must be run repeatedly for every change in the program, it is advantageous to have as small a set of test cases as possible.

## II. Problem Description

### A. Issues of Interest

With a tremendous number of possible test cases available, testers have no means to generate appropriate test cases. The ideal test cases should enhance possibility of exposing undetected errors. Despite the importance of techniques in identifying these test cases, developing the techniques remains one the most difficult aspects of software testing. It is generally accepted that the availability of more effective Tests would significantly reduce the cost associated with software development [3].

### B. Interested Problems

This paper tries to improve test performance as follows:

• **Reducing the number of test cases** – The reduction technique reduces the cost of executing and validating tests. Therefore it is of great practical advantage to reduce the number of test cases.

• **Automatic test case generation** – One of the most important components in a testing environment is an automatic test data generator.

• **Minimum number of test runs** – Use less time is spent on test runs.

## III. Literature Reviews

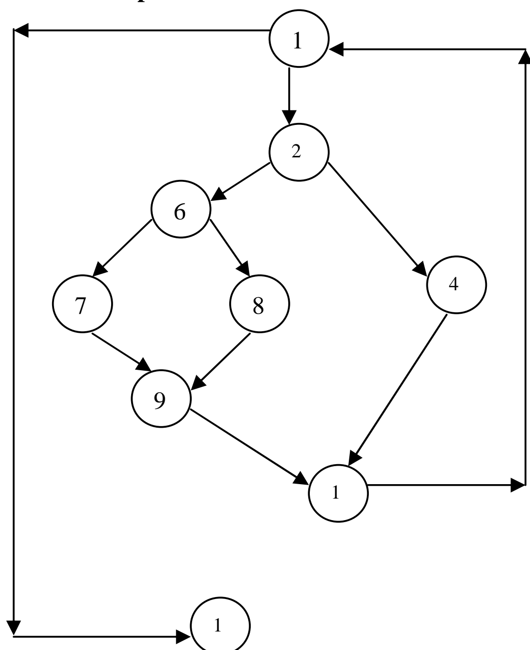
### A. Software-Testing Techniques

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].

**1) 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].

**2) 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**:




**CYCLOMATIC COMPLEXITY**:

The cyclomatic 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 introduces at least one new set of processing statements or a new condition (i.e. new edge) [1].

**Example**

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

**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 path because it is simply a combination of 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 be noted that the basis set is not unique. In fact, a 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].

### C. 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.*
***Postcondition***
*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.Bot= 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
**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]

## IV. Proposed Technique

*A. Objectives*

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.

In this paper, a new algorithm is used to meet the above-mentioned objectives, using the following steps.

### *A. Test Cases Generation Technique*

There are four steps to generate test cases:

1) 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 (>, >=, <, <=, ==, !=)

2) 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.

3) 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.

4) Using all of the above-mentioned values to create a table to present all possible test cases.

### *B. Expected Results*

Using the methodology, the new algorithm would have the following characteristics:

1) 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.

2) Automatic test cases generation. The test cases can be automatically generated with the reduction process.

3) Less time to test run. A single generation of test cases reduces the time of test run and compilation.

## V. 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:

1) Number of test cases
2) Reduction percentage of test cases
3) Compilation time

The evaluation is described using two examples

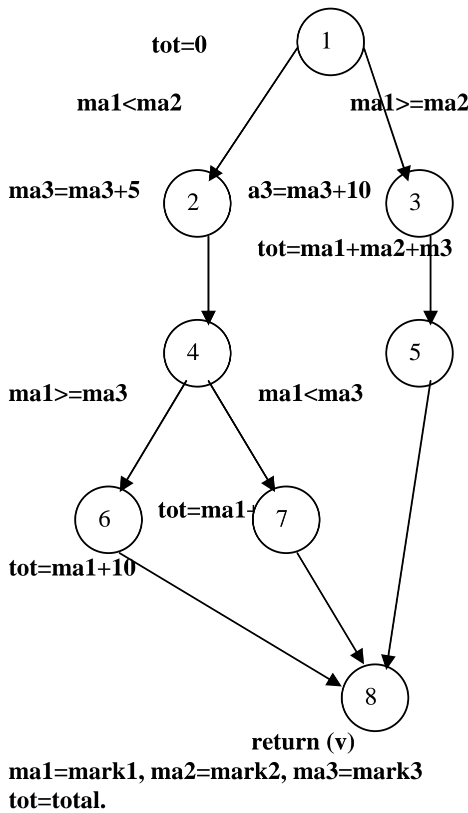
### *A. Example*

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

**1. Source code**

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

**2. Control flow graph**




**ma1=mark1, ma2=mark2, ma3=mark3**
**tot=total.**

3. No of independent path:

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

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:
1) Finding all possible constraints from start to finish nodes.
Ma1 < ma2, ma1> = ma3
2) 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.
3) Finding constant values in the path, if any. Ma1 constant value for variable *ma3*found on node 2 of the path has been used to replace the fix value of *ma3* (10) at the node.
4) 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. 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:

1. Finding all possible constraints from start to finish nodes.
ma1<ma2, ma1>=ma3, ma3=10
2. Calculate split value and splitting

Intervals for all constraints.

(i) 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

TABLE I

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

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

TABLE II

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

(ii) 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

TABLE III

| No | Ma1 | Ma3 |
|---|---|---|
| 1 | 0 to 10 | 0 to 10 |
| 2 | 11 to 30 | 11 to 40 |

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

TABLE IV

| No | Ma1 | Ma3 |
|---|---|---|
| 1 | 0 to 10 | 0 to 10 |
| 2 | 11 to 30 | - |

(iii) Third constraint is ma3=16.

TABLE V

| No | Ma3 |
|---|---|
| 1 | 16 |

From Table II, Table IV, Table V, finally calculate all selected intervals

TABLE VI

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

From the Table VI, total test cases are 651.

## VI. EVALUATION RESULTS

TABLE VII

| 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 | 162.75 |

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

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

## VII. ANALYSIS GRAPH

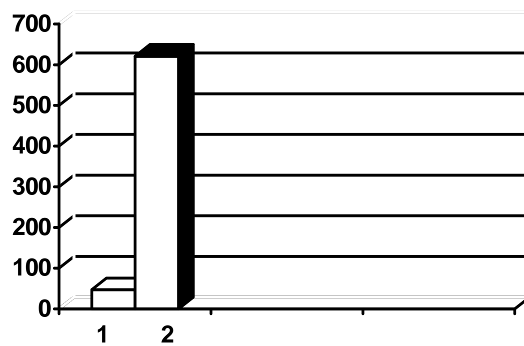



Fig. 1 X-axis for algorithm, Y-axis for reduced test cases, 1- for proposed solution, 2- for existing solution

## VIII. CONCLUSION

The new proposed technique has achieved greater reduction percentage of the test cases while keeping test cases generation to a single run. Furthermore, for compilation, it has been found that the new technique is the least time-consuming among the one existing technique. Based on the analysis done, the proposed method can be considered a superior technique from all others available in current literatures. Limitation of the proposed technique lies in its requirement for identification of fix values for all variables, either as maximum, minimum or constant values. The technique is not applicable where there are more than two variables in the program code. The future work on the technique would, therefore, address these problems and find practical measures to overcome them.

ACKNOWLEDGMENT

There are far too many people to try to thank them all; many people have contributed to the development of this paper. We owe our deep regards and honour to express our gratitude to Dr. S. P. Sabbarwal, Director, SRM-Institute of Management & Technology, Modinagar and all the faculty members for providing me invaluable support, guidance, help and inspiration all through this paper.

REFERENCES

[1] B. Beizer. “Software Testing Techniques.” Van Nostrand Reinhold, 2nd edition, 1990.
[2] B. Korel, “Automated Software Test Data Generation,” Conference on Software Engineering, Vol 10, No. 8, pages 870-879, August 1990.
[3] L. A. Clarke, “A System to Generate Test Data and Symbolically Execute Programs,” IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pages 215-222, September 1976.
[4] L. J. Morell. “A Theory of Error-Based Testing,” PhD thesis, University of Maryland, College Park MD, 1984, Technical Report TR-1395
[5] M. J. Gallagher and V. L. Narsimhan, “ADTEST: A Test Data Generation Suite for Ada Software Systems,” IEEE Transactions on Software Engineering, Vol. 23, No. 8, pages 473-484, August 1997.
[6] Neelam Gupta, A. P. Mathur and M. L. Soffa, “Automated Test Data Generation using An Iterative Relaxation Method,” ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering (FSE-6), pages 231-244, Orlando, Florida, November 1998.
[7] Offutt A. Jefferson, J. Pan and J. M. Voas. “Procedures for Reducing the Size of Coverage-based Test.

**R. P. Mahapatra** post graduated M.E (CSE) from University of Madras and pursuing PhD from Berhampur University Orissa. His employment experience includes the Anna University, Madras University, Mekelle University, Ethiopia and presently working as an Assistant professor and HOD (CSE & IT) SRM-IMT, Modinagar Campus of SRM University Chennai. His special fields of interest included Software Engineering, Network Security and Artificial Intelligence.

**Jitendra Singh** post graduated M.Tech. (CSE) from IETE, New Delhi and M.C.A from IGNOU University & MSc (Maths) from CCS University Meerut. His employment experience includes the U.P. Technical University Lucknow, CCS University Meerut, Dr. B. R. Ambedkar University Agra and presently working as a Senior Lecturer in SRM-IMT, Modinagar Campus of SRM University Chennai. His special fields of interest included Software Engineering, Network Security and User Authentication Mechanism