A Tool for Generation of Test Cases in Black Box Testing

Mamta Sharma¹, Durgesh Srivastava²

Computer Science Assistant Professor
Maharishi Dayanand University Rohtak Maharishi Dayanand University Rohtak

Abstract: Software Testing is one of the major area in software development. Each phase of software development life cycle includes the software testing as its major part. As the software development begin the role of software testing also begin. In this Research we have developed a tool to generate different test cases automatically. To show the validity of the tool, we have considered the line equation problem and generated different test cases, and finally we conclude that Robustness Technique is better than Boundary Value Analysis.

Keyword: Black Box Testing, Boundary Value Analysis, Robustness Testing, Black Box Tool, White Box Testing,

1. Introduction

Testing is the major quality control measure used during software development. Its basic function is to detect errors in the software. During requirement analysis and design, the output is a document that is usually textual and non-executable. After the coding phase, computer programs are available that can be executed for testing purposes. Thus the goal of testing is to uncover requirement, design and coding errors in the programs. Consequently, different levels of testing are used. The starting point of testing is unit testing [12]. In this, a module is tested separately and is often performed by the coder himself simultaneously along with the coding of the module. The purpose is to exercise the different parts of the module code to detect coding errors. After this, the modules are gradually integrated into subsystem, which are then integrated to eventually form the entire system. During integration of modules, integration testing is performed to detect design errors by focusing on testing the interconnection between modules.

2. Testing

It finds the errors in software design [3]. During software testing we find bugs and errors in the software and remove them. We can define the testing as:

(i) The process of exercising software to verify that it satisfies specified requirements.
(ii) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs), and to evaluate the features of the software item.
(iii) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.

The paper is organized as follows: Section 3 explains different types of testing i.e., black box and white box testing. Experimental work is carried out in section 4. Snapshots of the output are given in section 5. We have tabulated the results of the proposed tool in section 6, and finally we conclude the paper in section 7.

3. Types of Testing

There are 2 types of testing: [1, 2, 3]

A) Black box testing
B) White box testing

A) Black Box

This type of testing involves testing the software for functionality, it is used to find out the errors in data structure, faulty functions, interface errors, etc. Black box testing ignores internal mechanism of a system [1]. It identifies bugs only according to software malfunctions as they are revealed in its erroneous outputs. It is used to find incorrect functions that led to undesired output when executed, incorrect conditions due to which the functions produce incorrect outputs, when they are executed [2]. Following techniques are used to test a program using black box testing techniques [1, 3, 4].

1. Boundary Value Analysis (BVA)
2. Robustness
3. Worst case
4. Equivalence Partitioning
5. Decision Table

Black Box Testing allows us to carry out the majority of testing classes, most of which can be implemented solely by black box tests. Black box testing requires fewer resources. Brief description about white box testing is given in the next sub-section. In this paper we will discuss only two techniques i.e., Boundary Value Analysis and Robustness Technique.

B) White Box

White-box testing takes into account the internal mechanism of a system or component. White-box testing is also known as structural testing, clear box testing, and glass box testing. It involves testing of all logic of program, testing of loops, condition testing and data flow based testing. This helps in detecting errors even with unclear and incomplete software specification. The objective of white box testing is to ensure that the test cases exercise each path through a program [1]. The test cases also ensure that all independent paths within the program have been executed at least once. All internal data structures are exercised to ensure validity. All loops are executed at their boundaries and within operational bounds. Each branch is exercised at least once.
4. Experimental Work

In this paper we have developed a tool and implemented in PHP to generate the test cases automatically. In order to validate the functionality of the tool we have considered a straight line problem, with $(m_1, c_1)$ and $(m_2, c_2)$ defining the lines of the form $y = mx + c$ with following conditions:

(i) Parallel lines ($m_1 = m_2$, $c_1 \neq c_2$
(ii) Intersecting lines ($m_1 \neq m_2$
(iii) Coincident lines ($m_1 = m_2$, $c_1 = c_2$

(A) Boundary Value Analysis

In black box testing, test cases are derived on the basis of values that lie on the edge of the equivalence partition [5]. It is found that most of the errors occur at the boundary rather than the middle of the domain. If an input consists of certain values, then test cases should be able to exercise both the values at the boundaries of the range and the values that are just above and below the boundary values. For example for the range $[10, 100]$ the input values for a test would be $10, 11, 55, 99, 100$ Boundary Value Analysis focuses on the input variables of the function. Let us consider two variables $Y_1$ and $Y_2$, where $Y_1$ lies between $A$ and $B$ and $Y_2$ lie between $C$ and $D$.

$$A \leq Y_1 \leq B \ (1)$$

$$C \leq Y_2 \leq D \ (2)$$

We have summarized the total $4n+1$ number of test cases and the expected output in the table-I

<table>
<thead>
<tr>
<th>TEST CASE</th>
<th>$M_1$</th>
<th>$C_1$</th>
<th>$M_2$</th>
<th>$C_2$</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>3</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Coincident Lines</td>
</tr>
<tr>
<td>4</td>
<td>99</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>6</td>
<td>55</td>
<td>10</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>11</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>99</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>9</td>
<td>55</td>
<td>100</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>55</td>
<td>10</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>11</td>
<td>55</td>
<td>55</td>
<td>11</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>12</td>
<td>55</td>
<td>55</td>
<td>99</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>13</td>
<td>55</td>
<td>55</td>
<td>100</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>14</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>10</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>15</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>11</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>16</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>99</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>17</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>100</td>
<td>Parallel Lines</td>
</tr>
</tbody>
</table>

(B) Robustness Testing

A component is robust when it never fails or crashes, whatever the input is. Indeed, the failure of a single component may cause the failure of the entire system. IEEE defines robustness as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [7]. The value of detecting errors early in the development cycle, both in terms of cost and schedule, is well known. Boehm and Basili reported “Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase.” This statement is as true for robustness problems as it is for functionality defects. If we adapt our function $f$ to apply to Robustness testing we find the following equation:

$$f = 6n + 1$$

We have summarized the total number of test cases and the expected output in the table-02

<table>
<thead>
<tr>
<th>TEST CASE</th>
<th>$M_1$</th>
<th>$C_1$</th>
<th>$M_2$</th>
<th>$C_2$</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Coincident Lines</td>
</tr>
<tr>
<td>5</td>
<td>99</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>7</td>
<td>101</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td>9</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>9</td>
<td>55</td>
<td>10</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>10</td>
<td>55</td>
<td>11</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>11</td>
<td>55</td>
<td>99</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>12</td>
<td>55</td>
<td>100</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>13</td>
<td>55</td>
<td>101</td>
<td>55</td>
<td>55</td>
<td>Parallel Lines</td>
</tr>
<tr>
<td>14</td>
<td>55</td>
<td>55</td>
<td>10</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>15</td>
<td>55</td>
<td>55</td>
<td>11</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>16</td>
<td>55</td>
<td>55</td>
<td>99</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
<tr>
<td>17</td>
<td>55</td>
<td>55</td>
<td>100</td>
<td>55</td>
<td>Intersecting Lines</td>
</tr>
</tbody>
</table>

Following program shows the implementation of one of the module of the proposed tool.

```
<style>
.five {width: 125px; padding: 1px; display: inline-block ;}
.boundary form {text-align: centre ;}
Input {
    margin: 10px;
    padding: 5px;
    width: 250px;
}
</style>
<form method='post' action='' class="boundaryform">
    <input type="text" value="" placeholder="First Value" required>
    <input type="text" value="" placeholder="Second Value" required>
    Enter 1 for Boundary Value Analysis
    Enter 2 for Robustness Testing
    Enter 3 for Exit<br>
    <input type="text" value="" placeholder="Enter your Choice" name="choice" required>
    <input type="submit" value="enter">
</form>
```
$e=''$
$Scount=0;$
$Smid=''$
$Ssteps='';
$Sch=''$
$Sans='';
if (@$_POST['fboundary'] !== '' && @$_POST['sboundary'] !== '' && @$_POST['choice'] !== ''){
$s=$_POST['fboundary'];
$e=$_POST['sboundary'];
$ch=$_POST['choice'];
if($ch==1)
{
$mid=2;
$steps=5;
$a[0]=$s;
$a[1]=$s+1;
$a[2]=($s+$e)/2;
$a[3]=$e-1;
$a[4]=$e;
}
else if($ch==2)
{
$mid=3;
$steps=7;
$a[0]=$s-1;
$a[1]=
$s;
$a[2]=$s+1;
$a[3]=($s+$e)/2;
$a[4]=$e-1;
$a[5]=$e;
$a[6]=$e+1;
}

5. Output of Proposed Tool

In this section we have shown the various snapshots of the output of the proposed tool under different situations. We have delineated the comparison between BVA and RT is shown.
6. Summarized Test Cases

In table 3 showing total number of test cases generated.

**Table 3: Test Cases Summarized**

<table>
<thead>
<tr>
<th>S. No</th>
<th>Type of Test</th>
<th>No. of Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boundary Value Analysis</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>Robustness Testing</td>
<td>25</td>
</tr>
</tbody>
</table>

In Table 4 showing total number of Intersecting, Coincident & Parallel Lines in Boundary Value Testing and Robustness Testing.

**Table 4: Comparing Results**

<table>
<thead>
<tr>
<th>Type of Test</th>
<th>Intersecting</th>
<th>Parallel</th>
<th>Coincident</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary Value Analysis</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Robustness Testing</td>
<td>12</td>
<td>12</td>
<td>1</td>
</tr>
</tbody>
</table>

7. Conclusion & Future Scope

During the analysis of black box testing we conclude that the total number of test cases in case of BVA and Robustness are 17 and 25 respectively, and finally we have shown that the robustness technique is better than Boundary Value Analysis we can develop it and use it for other test case Designing techniques also. Like Decision Table, Equivalence Class Testing: We can develop it and use it for other test case Designing techniques also. Like Decision Table, Equivalence Class Testing.

References


