A hybrid Black Box technique for the selection of optimum number of test cases

Bratin Bhattacharya
Department of MCA, MERI, GGSIP University, Delhi
E-mail:bratinb.10.11@gmail.com

Deepa Gangwani
Faculty, Department of MCA, MERI, GGSIP University, Delhi
E-mail:gangwanideepa@yahoo.com

ABSTRACT
Software Testing is the dominating verification technique used in industry today, and many man-hours and resources are invested in the testing of software products. Black box testing (also called functional testing) is a type of testing that ignores the internal mechanism of a system and focuses solely on the outputs generated in response to selected inputs and execution conditions. Various Black box testing techniques used by the industry today include Boundary Value Analysis (BVA), Equivalence partitioning, Decision table based testing, Cause-Effect graphing technique, Error guessing and Comparison testing. The most important prerequisite for a thorough software test using any technique is the design of relevant test cases, since they determine the kind and scope, and hence the quality of the test. Therefore, a reasonable and effective method for the selection and generation of test cases is urgently needed.

This article first introduces some traditional methods of Black box test case generation, and then proposes a better strategy of generating test cases such that maximum code coverage is attained with minimum test cases.

The proposed strategy, if implemented systematically, can lead to considerable reduction in the cost to a company by ensuring that errors surfacing at a stage are not propagated to later stages and also improve its reputation by reducing the need of post-release debugging.

KEYWORDS
Software testing, Black box testing, Functional testing, Boundary value analysis, Equivalence Class partitioning.

1. INTRODUCTION

1.1 Software testing defined
Software testing is defined as any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

1.2 What is Black Box Testing?
Also known as functional testing, black box testing is a software testing technique whereby the internal workings of the item being tested are not known by the tester. For example, in a black box test on software design the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. The tester does not ever examine the programming code and does not need any further knowledge of the program other than its specifications.

1.3 Black Box Testing vs. White Box Testing
Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing.

The main difference between black-box and white-box testing is the areas on which they choose to focus. In simplest terms, black-box testing is focused on results. If an action is taken and it produces the desired result then the process that was actually used to achieve that outcome is irrelevant. White-box testing, on the other hand, is concerned with the details. It focuses on the internal workings of a system and only when all avenues have been tested and the sum of an application’s parts can be shown to be contributing to the whole is testing complete.

1.4 Boundary Value Analysis (BVA)
It is based on the concept that the density of defects is more towards the boundaries. This is done due to the following reasons:

- Programmers are usually not able to decide whether they should use < operator or <= operator while making comparisons.
- Different terminating conditions of for loops, while loops and repeat loops may cause the defects to move around the boundary conditions.
The requirements themselves may not be clearly understood especially around the boundaries, thus causing even a correctly coded program to not perform in the correct way.

1.4.1 What is BVA?
[3]The basic idea of BVA is to use input variable values at their minimum (min), just above the minimum (min+), a nominal value (nom), just below their maximum (max -) and at their maximum (max).

1.4.2. The Critical Fault Assumption Theory
BVA is based on the Critical Fault Assumption. The assumption states that failures are only rarely the product of two or more simultaneous faults. Thus using this assumption, we vary the value of just 1 variable keeping the two or more simultaneous faults. Thus using this assumption, assumption states that failures are only rarely the product of BVA is based on the Critical Fault Assumption. The

1.2.1 Equivalence Class Partitioning explained
[3]In this technique, the input and output domain is divided into a finite number of equivalence classes. Then, we select one representative of each class and test our program against it. It is assumed by the tester that if one representative from a class is able to detect the errors, then why should he consider other cases. Furthermore, if this single representative test case did not detect any error then we assume that no other test case from this class can detect error. Thus Equivalence classes form a partition of mutually disjoint subsets (of valid & non valid cases) whose union is the entire set .Elements in an equivalence class are expected to cause the program to act the same. We would expect the program to react differently while operating on inputs from different classes.

The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. The fact that the entire set is represented provides a form of completeness .The disjointedness assures a form of non-redundancy. If the Equivalence classes are chosen wisely it greatly reduces redundancy among test cases.

[2]The key of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, this choice is made by “second guessing” the likely implementation, and thinking about the functional manipulations that must somehow be present in the implementation.

2. RESEARCH AND ANALYSIS

2.1 Limitations of BVA and ECP
I think the two techniques of Black box testing, namely, Boundary Value Analysis (BVA) & Equivalence Class Partitioning (ECP) have a few loopholes due to which they would not be able to accurately or exhaustively test the software. They are enumerated as follows:-

- BVA considers only the boundary values. Therefore it leaves a lot of data UNTouched.
- In ECP, although we cover the entire data by creating Equivalence Classes but from each Equivalence Class we randomly select a value. Now it’s possible that the particular value is an extreme value within that particular class and so does not represent the universe correctly.

In order to remove these inconsistencies I have developed my own algorithm which borrows the best features of BVA and ECP and uses its own concepts to provide an exhaustive and accurate technique.

2.2 Assumptions of my system
- The given function is such that it can be represented graphically. i.e. logical and Boolean valued functions cannot be tested using this technique.

- Preferably the input domain range should be large i.e. consisting of at least 1000 values to choose a sample from. Although this assumption does not mandatorily need to be followed and the system would work with small and medium sized programs as well, it is best suited for large programs.

2.3 Description of the algorithm
- The algorithm MainAlgo(i) takes as an input ‘i’ which is the total number of samples we will draw from the input domain of the particular problem.

- This number (i) is supplied by the tester based on the size of the input domain. An ideal number could be 1 representative value for every 100 units.

- The number of samples from the input domain and their corresponding values in the output domain are taken into two arrays of the same size, arr1 and arr2.

- In order to take samples from the input domain we make use of the LAW OF STATISTICAL REGULARITY which states that a comparatively small group of items chosen at random from a very large group will, on the average, represent the characteristics of the large group.

- Accordingly we use a rand() function which randomly generates numbers and stores in the array arr1.

- The elements thus stored in the array arr1 are sorted using merge sort algorithm.

- Why merge sort algorithm?
A hybrid Black Box technique for the selection of optimum number of test cases

- The only other alternative which could have been as good as merge sort is insertion sort. But for large inputs, merge sort would outperform insertion sort as it has complexity of $O(n \log n)$ compared to $O(n^2)$ for insertion sort and $n$ grows much faster than $\log n$. For small inputs insertion sort would only be marginally better ONLY if the list is more or less sorted which is never the case in the real world.

- After the elements are sorted we then compute $f(x)$ for each element $x$ in the sorted array $arr1$.

- These $f(x)$ values are stored in a separate array $arr2$. We could have also used a 2-dimensional array as a data structure instead of 2 1-dimensional arrays but internally even a 2-dimensional array is stored as a series of 1-dimensional arrays.

- Now we make a graph $G$ using the data from the two arrays, $arr1(x$-coordinate) and $arr2(y$-coordinate).

- This function takes as input the graph $G$ and partitions it into distinct graphs $G[1]$, $G[2]$, $G[3]$ and so on..... The logic it uses to partition the graph is that it starts traversing the graph from the extreme left and creates a partition at the point where the slope behavior changes. Thus in each partition of the graph $G$, the slope behavior is uniform (i.e. either ascending or descending or zero slope).

- For each partition $G[i]$, we call the function $EquivalenceClassMaker(G[i])$. This function takes as an input the particular graph-partition $G[i]$ and makes an equivalence class out of it consisting of the Cartesian product of 5 values(\(min, min+, median, max-\), \(max\)).Thus if we have ‘n’ equivalence classes, then we would have a total of $5^n$ test cases.

- The reason why I’m considering the median and not the common arithmetic mean is because mean gets affected by extreme values but median being the positional average overcomes this defect and since our array consists of ORDERED values therefore median is the best choice.

- Finally control returns to the main algorithm where we create two additional equivalence classes for the invalid values at the two ends of the input domain. An important point to note here is that unlike the other equivalence classes where we consider the Cartesian product of 5 values, in these two equivalence classes, we consider only one invalid value from each class.

2.4 The algorithm

$MainAlgo(i)$

```
{  int i,j, arr1[i], arr2[i];
  for(j=1, j<i, j++)
    do
      arr[j] <- rand();
}
```

```
Call MergeSort(arr,1,i);
for each element, x, in the sorted array arr1
  do
    {  compute $f(x)$;
      arr2[i] <- $f(x)$;  // $f(x)$ is the function
                        // which we are testing
    }
  make a graph, G
  do
    {  x-coordinate <- x-values;
       y-coordinate <- corresponding $f(x)$ values
    }
  Call graphPartition(G);
Make 2 equivalence classes for the invalid values at either end of the input spectrum;
}
```

```
graphPartition(G)
{  int i <- 1;
  while(EndofGraph G)
    do
      {  Traverse the graph G to the point where the slope behaviour changes(i.e. ascending-descending or vice-versa,ascending-zero slope or vice-versa,or descending-zero slope or vice-versa);
      }
  Create a partition G[i] at that point;
  Call EquivalenceClassMaker(G[i]);
    i <- i+1;   }
}
```

```
EquivalenceClassMaker(G[i])
{  int i;
  for the partition G[i] make an equivalence class consisting of 5 values(min, min+, median, max-\), max);
  take the Cartesian product of these 5 values;
  the result is the number of test cases;
}
```
MergeSort(A,p,r) //A,p,r store respectively the array //to be sorted, its starting index //and its ending index {
    if p < r
        then q ← ⌊(p+r)/2⌋
        MergeSort(A,p,q)
        MergeSort(A,q+1,r)
        Merge(A,p,q,r)
}

Merge(A,p,q,r)
{
    n₁ ← q – p + 1
    n₂ ← r – q
    create arrays L[1…… n₁ + 1] and R[1…… n₂ + 1]
    for i ← 1 to n₁
        do L[i] ← A[p + i – 1]
    for j ← 1 to n₂
        do R[j] ← A[q + j]
    L[n₁ + 1] ← ∞
    R[n₂ + 1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
            i ← i + 1
        else A[k] ← R[j]
            j ← j + 1
}

2.5 An example to explain the algorithm
Consider a problem in which we are given a function f(x) which behaves in the following manner with respect to the different values of the variable x.:-
{x: 1 <= x <= 17}

<table>
<thead>
<tr>
<th>Variable values(x)</th>
<th>Output(f(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

On applying merge sort to the input domain(x), we obtain the following list:

<table>
<thead>
<tr>
<th>Variable values(x)</th>
<th>Output(f(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>40</td>
</tr>
<tr>
<td>13</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>40</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

Now we plot a graph for our function as follows:-
A hybrid Black Box technique for the selection of optimum number of test cases

After having made the graph, Graph 1, we supply this graph to the function `graphPartition()` which takes this graph as input and makes 4 partitions in such a way that in each partition the slope behavior is uniform. Thus we obtain the following 4 partitions:

1. **Graph 1[i]**
2. **Graph 1[ii]**
3. **Graph 1[iii]**
4. **Graph 1[iv]**
Now we take each of the above 4 partitions and make an equivalence class consisting of the Cartesian product of 5 values. This exercise is shown for the first partition, Graph 1[i]. it is supplied to the function EquivalenceClassMaker computes the Cartesian product of the above mentioned 5 values as follows :-

\[ E_1 = \{x: x<1\} \]
\[ E_2 = \{x: x>17\} \]

This being a function of just one variable, x, there is no need of a Cartesian product and the test cases are the same as for weak normal equivalence class testing. Thus for the first equivalence class, we have 5 test cases as mentioned above.

Similarly we can make equivalence classes for the other partitions as well and generate test cases for each of those equivalence classes.

In addition to four equivalence classes for the four partitions of the graph, there would also be two additional equivalence classes for invalid values. The two classes are as follows:

\[ E_1 = \{x: x<1\} \]
\[ E_2 = \{x: x>17\} \]

From each of the classes E_1 & E_2 we will randomly take an invalid value and create a test case.

This completes our testing.

3. CONCLUSION

There are many contemporary researches that are going on at present which address different issues of Black box testing techniques. It can be seen that the technique which I have proposed works well for programs with a very large input domain. While boundary value analysis touches upon a very minuscule portion of the data and equivalence class partitioning would not be able to capture extreme variations in the data correctly if they existed, my system with its concept of partitioning the graph of the given function can exhaustively capture all the extremities in the data correctly and present the true picture. The system further aims to even out the irregularities by considering the median value instead of the arithmetic mean which gets affected by extreme values. By the Law of Averages I can say that my system would be better than the Strong Normal Equivalence Class Partitioning by 50% and better than Boundary Value Analysis by a much larger percentage. The value of 50% improvement over Strong Normal Equivalence Class Partitioning has been arrived from the assumption that on an average when we randomly choose a value from a class interval, the number could either be a median value or it could be a non-median value. So there is 0.5 probability (50%) that the value would be a non – median value. In other words we achieve a minimum 50% improvement with the new technique. In reality, this percentage improvement is much larger. For e.g.: if there are 100 values from 1-100. Here the median is 50. The probability of selecting a non-median value is 0.99 and the probability of selecting the median is only 0.01. So here we obtain a 99% improvement with my proposed technique.

Further the Cartesian product of the 5 values (min, min+, median, max-, max) guarantees exhaustive testing and a very high degree of accuracy. Lastly I conclude by stating that although complete testing is never feasible because of time and cost constraints as has already been shown in this article and as has it been rightfully said that there is “no absolute proof of correctness”, my technique intends to achieve maximum input data coverage.

4. FUTURE SCOPE

The proposed system can only work with functions whose graph can be drawn. Therefore it is not suited for functions giving logical or Boolean values as output. In the future, I intend to modify the current system such that it can work for these functions as well. The current implementation gives an extremely accurate measure but it tends to be a little biased towards large programs while for small programs or organizations with a small budget or those implementing the Rapid Application Development model for their projects, it might seem a little expensive and time consuming to implement. So, in the future, I intend to customize the existing system for small and medium sized projects.

REFERENCES