Test Case Redundancy Detection and Removal Using Code Coverage Analysis

Anurag Panday
Assistant Professor
Dept. of CS&IT, MIT Moradabad
Email: anu_mohan31@rediffmail.com

Manish Gupta
Assistant Professor
Dept. of CS&IT, MIT Moradabad
Email: manishymca2007@gmail.com

Manoj Kumar Singh
Assistant Professor
Dept. of CS&IT, MIT Moradabad
Email: manojaswal1982@gmail.com

Navase Ali
Assistant Professor
Dept. of CS&IT, MIT Moradabad
Email: navaselucky@gmail.com

ABSTRACT
Software testing is an important and costly process of software development. For testing a software, a test suite is generated according to the requirements of the software. The software or program under test is executed for each test case of test suite. The actual output is matched with the expected output. The software under test can evolve according to the change in its requirements and due to this size of test suite grows. In testing both test generation and program execution against a test case consume some cost. So when the size of test suite will grow, this cost will also grow. For a given test suite, it is important that it should cover all requirements of the given software or program otherwise it will be inadequate. So the size of a test suite plays an important role in testing. It is the need that find minimized test suite that provide same coverage as the original test suite.

Keywords: Software testing, Test suite reduction, Test requirement.

1. INTRODUCTION
Software testing is the process to determine whether there is any error in the program. It usually involves execution of software on a particular set of input and a comparison of the actual software output with the expected output. The set of input with expected output is called “test case” for program and set of test cases is called “test suite” or “test set”. For testing a software or program a test suite is generated.[1,2]

There are three important concepts in software testing:–

Error: An error is a programmer mistake at the time of coding of program.

Fault (or Bug): A fault is a defect in the source code due to error.

Failure: When a system produces wrong result, it is failure due to defect

1.1 Program Requirements and Test Adequacy
A program/ software contain a set of requirements that should be fulfilled.

Consider a program P written to meet a set R of requirements; P and R are denoted as (P, R). Let R contains n- requirements r1, r2, r3...rn. Suppose T be a test set containing k- test cases to test P to determine whether it meets all the requirements in R. The test adequacy criterion is defined as follows:-

“A test set T for program (P, R) is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r”. Adequacy of a test set is measured against a finite set of elements which are derived from the requirements of the program under test according to adequacy criterion. For each adequacy criterion C, a finite set is derived called “Coverage Domain Ce”.

1.2 Control Flow Graph (CFG)
A control flow graph captures the flow of control within a program. It is also known as a “flow graph” or “program graph”.

For constructing CFG, basic blocks of a program are found. A “Basic Block” or simply “Block” of a program P is a sequence of consecutive statements with a single entry and single exit point. These points are the first and last statements within a block.

1.3 Types of Coverage Criteria
Statement Coverage: It is ensure that every statement of the program is executed at least once.
**Decision Coverage or branch coverage:** It is ensure that every branch direction of the program is pass through, that every start point is raised at least once.

**Condition Coverage:** It is ensure that each condition in a branch takes on all possible result at least once.

**Decision-Condition Coverage:** It is ensure that both decision and condition coverage conditions are should be satisfied.

**Modified-Condition/decision Coverage:** It is ensure that all possible combinations of condition results within each branch are pass through at least once.

**Multiple condition coverage:** This criteria requires that all combinations of con-ditions inside each decision are tested.

**Path Coverage:** A Path is a unique sequence of branches from the function start to end. A path is finite order of nodes connected by edges or list of connected edges. Also known as predicate coverage. All possible paths in control flow graph (CFG) to be executed.

### 1.4 Test Suite Reduction

The test suite reduction problem may be stated as follows

**Given:** Test suite $T$, a set of test case requirements $r_1, r_2, ..., r_n$ that must be satisfied to provide the desired test coverage of the program, and subsets of $T$, $T_1, T_2, ..., T_n$, one associated with each of the $r_i$ s such that any one of the test cases $t_j$ belonging to $T_i$ can be used to test $r_i$.

**Problem:** Find a representative set of test cases from $T$ that satisfies all of the $r_i$ s.

The $r_i$ s in the foregoing statement can represent various test case requirements, such as source statements, decisions, definition-use associations, or specification items.

A representative set of test cases that satisfies all of the $r_i$ s must contain at least one test case from each $T_i$; such a set is called a hitting set of the group of sets $T$, $T_1$, $T_2$, ..., $T_n$. To achieve a maximum reduction, it is necessary to find the smallest representative set of test cases. However, this subset of the test suite is the minimum cardinality hitting set of the $T_i$s, and the problem of finding such a set is NP-complete. Thus, minimization techniques resort to heuristics.

The number of tests eliminated is given by $(T - T_{min})$, and the percentage of tests eliminated is given by $((T - T_{min})/T * 100)$.

**Literature Survey**

In the most of the approaches in the literature pertaining to test redundancy detection and reduction, coverage information is the basis [4, 5, 6, 7, 8, 9]. By using the coverage information, test case redundancy can be found but it does not guarantee to keep fault detection capability of a given test suite [10]. In [5], the authors mentioned all-definition coverage criterion on simple program. Although some existing test cases in the test suite may retest the modified software, the change in test case requirements may require new test cases and may also allow unnecessary test cases to be eliminated. There are two basic approaches to retesting after program changes. The control flow coverage criteria used in [4, 7, 9] are branch coverage in [4], statement in [7] and MC/DC in [9]. In [7], authors proposed three basic orderings for executing test cases, which lead to a total of 12 heuristic reduction procedures. These procedures run the tests in different orders, with the goal of increasing the difference between the original and subsequent orderings as much as possible. The three basic orderings therefore go from the beginning to the end, the end to the beginning and the middle to both ends. The authors were able to reduce 30% for mutation based test sets and 50% for state coverage based test sets. They were used Mothra mutation system for creating mutants and Godzilla as test case generator in the empirical evaluation and also calculated the cost for test reduction. The Systems Under Tests (SUTs) used were small scale. In [4], authors examined costs and benefits of test suite minimization based on the study of analysis Wong, Horgan, London, and Mathur [12] and named it WHML study. This study leaves many open questions and the following questions motivated in their work.

In [9], a case study was done to analyze “using coverage based criteria for reducing the size of test case is reasonable or not”. The following steps summarize the experimental methodology used in this work.

In [6], “presents an algorithm for test-suite reduction and prioritization that can be tailored effectively for use with MC/DC”. The main contribution of this paper:

1. Analyzing the problems that occur at the time when existing test-suite reduction and prioritization algorithms are used with MC/DC, so the new and effective techniques are necessary;
2. A description of general approach for new algorithm design;
3. In [13], to avoid redundancy test-cases can be transformed instead of discarding. Authors were used a model checker based test case generation approaches. Model checkers use Kripke structures as model formalism i.e., a Kripke structure $K$ is a tuple $K = (S, s_0, T, L)$, where $S$ is the set of states, $s_0$ 2 $S$ is the initial state, $T$ $S$ $S$ is the transition relation, and $L: S \rightarrow 2^{AP}$ is the labeling function that maps each state to a set of atomic propositions that hold in this state.

In [9], some experiments were performed to evaluate coverage-based test redundancy detection. This paper presents an approach using human computer interaction for inspecting the process of test suite reduction, and named as “tester assisted methodology”. The approach of this paper shows that “coverage information can’t be the only source of knowledge to precisely detect test redundancy.

Therefore semi automated approaches were come into existent and they are effective also [14; 15;]
3. PROPOSED APPROACH

The steps of our proposed approach are given in Figure 1.

Following information is given for program under test:

A program P, with requirement R. Test-suite TS = t1, t2, t3....

We now find the CFG (control flow graph) of the program P. From this graph we extract all possible paths p1, p2...etc.

For finding reduced test-suite, an association is needed between test cases and paths. This association provides information path coverage by test cases and redundancy between them. So we are introducing a coverage matrix. This matrix contains all paths in columns and all test cases in rows. We define matrix as:

\[ C(t_i; p_j) \] where ti 2 TS and pj is path;

The information in the matrix is all the digits of ‘0’ or ‘1’ which denote the path coverage information; here ‘1’ in the row i and column j means test case ti executed path pj and ‘0’ means test case ti did not executed path pj. For example, let a coverage matrix be shown in Figure 2.

Coverage score for each test case is calculated by using formula in equation (1).

\[ CS(t_i) = \frac{1}{8} = 0.12; \]
\[ CS(t_2) = \frac{3}{8} = 0.37; \]
\[ CS(t_3) = \frac{3}{8} = 0.37; \]

<table>
<thead>
<tr>
<th>Path</th>
<th>Subset Si</th>
<th>Coverage score</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>S1={t1, t11}</td>
<td>ti = 0.12, t11=0.25</td>
</tr>
<tr>
<td>P2</td>
<td>S2={t2}</td>
<td>t2=0.37</td>
</tr>
<tr>
<td>P3</td>
<td>S3={t2, t12}</td>
<td>t2=0.37, t12~0.12</td>
</tr>
<tr>
<td>P4</td>
<td>S4={t2, t13}</td>
<td>t2=0.37, t13~0.12</td>
</tr>
<tr>
<td>P5</td>
<td>S5={t3}</td>
<td>t3~0.12</td>
</tr>
<tr>
<td>P6</td>
<td>S6={t7, t8, t11}</td>
<td>t7<del>0.12, t8</del>0.12, t11~0.25</td>
</tr>
<tr>
<td>P7</td>
<td>S7={t9, t10}</td>
<td>t9<del>0.12, t10</del>0.12</td>
</tr>
<tr>
<td>P8</td>
<td>S8={t4, t5, t6}</td>
<td>t4<del>0.12, t5</del>0.12, t6~0.12</td>
</tr>
</tbody>
</table>

![Figure 1: Proposed model](image1)

![Figure 2: Coverage matrix](image2)

![Figure 3: Subsets of test cases according to paths](image3)
CS(t3) = 1/8 = 0.12;
CS(t4) = 1/8 = 0.12;
CS(t5) = 1/8 = 0.12;
CS(t6) = 1/8 = 0.12;
CS(t7) = 1/8 = 0.12;
CS(t8) = 1/8 = 0.12;
CS(t9) = 1/8 = 0.12;
CS(t10) = 1/8 = 0.12;
CS(t11) = 2/8 = 0.25;
CS(t12) = 1/8 = 0.12;
CS(t13) = 1/8 = 0.12;

Now on the basis of coverage matrix, subsets of test cases are made for each column according to paths. Table in Figure 3 shows these subsets.

First, all test cases from singleton subsets i.e., t2 and t3 from S1 and S5 are selected and included in primary test set Tp and related paths p2 and p5 are marked. Then, subsets that have these test cases are identified i.e., t2 is available in S3 and S4, related paths p3 and p4 are marked. Next, test cases from subsets with 2 elements from unmarked subsets are identified i.e., t1, t11 and t9, t10 from S1 and S7. Test cases with higher coverage score are selected. From S1, S11 is selected. But in S7, both have equal score, so by random choice, t9 is selected. Then, p1, p6 are marked and p6 is also marked because S6 contains t11. Same process is repeated for remaining subset S8, which is the only unmark subset. All test cases in S8 have equal score, so again by random choice, t4 is selected. In this, a reduced test-suite Tp = {t1; t3; t4; t9; t11} is created. This primary test set gives same coverage as original test-suite TS.

4. EXPERIMENTAL STUDY

Experiment: As shown in Figure 4, this program finds greater integer from given two positive integers and square that.

```
1. main()
2. { 
3. 1. Integer a, b, c;
4. 2. Input a, b;
5. 2. if (a>b)
6. 3. then c=a*a;
7. 4. else c=b*b;
8. 5. Output c; } 
9. 6. End
```

**Figure 4:** Program to find greater integer from two numbers

The control flow graph of the program under test is shown in Figure 5:

Now from the CFG, following possible paths are identified (a path begins from Start node S and end at End node E):

- p1: S-1-2-3-5-E
- p2: S-1-2-4-5-E

Using these paths and test cases, a coverage matrix is created and shown in Figure 6.

Using these paths and test cases, a coverage matrix is created and shown in Figure 6.

By using coverage matrix, coverage score for test cases are calculated. Then, subsets of test cases are identified and shown in Figure 7.

First, start from singleton set S1 and selects t1 and put it into primary set Tp = {t1}.

Since, no other subset contains t1, so move further after marking p1. In subset S2, every test case has equal score and then randomly t2 is chosen and included in Tp. Now for remaining test cases i.e. for t3 and t4, find which is useful for future aspect and which is useless. Both test cases follow path p2, so conditions from p2, i.e. c1 : a < b and c2: a = = b are extracted. Besides c1 and c2, no other condition is possible. Here, t4 satisfies c1 which is already in the program and hence no new condition is demanded by t4. For t3, which demands equality condition i.e. c2 and in future this condition may be used which will not violate requirement of program. Here t4 is useless and t3 is useful in future aspect, so t3 is included in Ts.
Finally two test sets are gained i.e. \( T_p = \{t_1, t_2\} \) and \( T_s = \{t_3\} \), and now \( t_4 \) can be removed. When program is changed then \( T_p \) is executed initially and if required then \( T_s \) is executed. If some more paths are remaining then new test cases are generated for them.

6. CONCLUSION AND FUTURE WORK

A 2-tier approach has been proposed for test suite reduction. It works on those redundant test cases which are although redundant but may be useful in future when program evolves. Many of the previous approaches either remove redundant test cases or find reduced set but do not remove any test cases. Both of these cases create confusion always for fault detection effectiveness. To overcome this problem, an approach is presented in the paper. It is tested on simple and middle size program, but it can be also effective for larger programs. This approach gives us a chance to save efforts of new test case generation process. Although this approach has been applied manually but if automated, it will provide better and faster result. It can also be integrated with automated test case redundancy detector.

REFERENCES