Pre and Post Test Suite Reduction Techniques: A Comparison Study

MOHAMMED AKOUR, REHAM BANI-YOUNIS, SOMAYYA ABO ALFOUL, 
SAJIDA MUSLEH 
CIS Department 
Yarmouk University 
University Street, Irbid 
JORDAN 
Mohammed.Akour@yu.edu.jo, banyyounis_reham@yahoo.com, sumayyatalat@gmail.com, mcsjust@gmail.com 

IMAN AKOUR 
MIS Department 
University of Sharjah 
University City, Sharjah 
UNITED ARAB EMIRATES 
iakour@sharjah.ac.ae 

Abstract: - Test suite reduction is a critical activity which occurs pre or post test cases generation process. As software keeps growing large amounts of new test cases will be generated and added to the test pool and others will be updated. Accordingly test suite size will increase. Test suite reduction techniques have been proposed to eliminate redundant or irrelevant test cases based on variant criteria, while seeking to maintain the total effectiveness of the reduced test suite. In this paper we address and compare four test case reduction techniques: CBR (case-based reasoning) deletion algorithms Technique, GE & GRE Heuristics and priority cost technique, Model-checker based technique and Base choice coverage criterion technique. The aim of the study is to provide a guideline for choosing the appropriate test suite reduction techniques.

Key-Words: Test suite reduction techniques; requirement test coverage.

1 Introduction

One of the most important phases of SDLC (Software development life cycle) is Software Testing. It is an important component of software quality assurance. There are many definitions available for Software Testing, but one can shortly define that as: A process of executing a program with goal of finding errors [2]. Some people get confused about the goal of testing, thinking that the goal is to check if a program is free from errors, while the goal is finding errors. So tests show the presence not the absence of defects. Miller gives a good description of testing in [3]: “The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances”.

Testing typically consumes 40–50% of development efforts, and consumes more effort for systems that require higher levels of reliability [4]. Although it is often impossible to find all errors in the program, the selection of right strategy at the right time will make the software testing efficient and effective [5].

The tester may or may not know the inside details of the software module under test, therefore either white-box testing or black-box testing can be used against the software module by generating a set of test cases [6]. A set of test cases is a set of (inputs, execution preconditions, and expected outcomes). This means that test cases check if a program for specified inputs gives the expected results. While a Test-Suite is a set of requirements and subsets of test cases, each requirement must be satisfied be at least one test case [1].

Our paper is organized as follows: in section 2, we present the problem under investigation, section 3 demonstrates the related works, in section 4, we describe each reduction technique and provide a
comparison between them, and finally we conclude the paper in section 5.

2 Problem Under Investigation
With a tremendous number of possible test cases available, especially in case of complex programs, testers have to generate appropriate test cases in a way that reduces the cost, time and efforts of executing and validating tests [8]. Another important aspect of software testing is the number of test cases that have a direct effect on the cost of testing, particularly that of regression testing [7] (testing activity that is performed to provide confidence that the changes made don’t harm the existing behavior of the software), it means the process of retesting the software after changes. So when tests must be run repeatedly for every change in the program, it is advantageous to have as small set of test cases as possible. Thus test case reduction aims to finding a minimal subset of the test-suite that can cover all possible. Thus test case reduction aims to finding a minimal subset of the test-suite that can cover all requirements [7]. Many techniques are available and have their own advantages and disadvantages and we can classify them into two types:

- Pre-process Reduction techniques (techniques reduce the test-suite before generation).
- Post-process Reduction techniques (techniques reduce the test-suite after generation).

The main goal of this article is to expose some of available pre and post test case reduction techniques and briefly manifest the mechanism for each technique. We compare these techniques by considering their advantages and disadvantages.

3 Related Works
Test suite minimization techniques (post-process) reduce the size of the test suite based on removing redundant test cases (unnecessary test cases) from it. There are many researchers who proposed a method to reduce unnecessary test cases, like Rothermel [17], McMaster [18] and Sampath [19]. These techniques aim to remove and minimize a size of test cases while maintaining the ability to detect faults. Previous works on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem. Horgan and London applied linear programming to the test case minimization problem in their implementation of a data-flow based testing tool, ATAC [21, 22]. Akour et al [33] provide test case reduction technique for adaptive software system. Their approach employed Change propagation theme to synchronizing component models and runtime test models and then removed the test cases that associated with a component targeted in reductive changes.

The use of a model-checker allows the detection of equivalent mutants. Therefore only non-equivalent mutants are used for the evaluation of a mutant score. A traditional minimization approach is applied to the model-checker scenario, similarly to Heimdahl and Devaraj [25].

A minimized subset of the test-suite achieving a criterion can be determined by calculating the covered properties for each test-case, and then iteratively selecting the test-case that covers the most yet uncovered properties. Black [24] proposed a test-case generation approach based on mutation of the reflected transition relation. The mutated, reflected properties can be used similarly to trap properties for test-case generation, to determine a kind of mutant score and also for minimization.

A domain of a program with mutually independent parameters is a set of all combinations of all values of these parameters. The input domain can be very big, so the main goal of domain testing methods is to achieve a test suite in which the size is considerably smaller than the count of all inputs of the program, and which effectively reveals failures of the program as much as possible [26]. There are two groups of domain testing methods – equivalence class testing (ECT) methods and boundary value testing (BVT) methods [26].

There are many methods that different authors call domain testing methods or domain analysis methods that take into account dependencies or interactions between input parameters [7, 27, and 28]. By these methods, the input domain often is seen as a geometrical shape and its edges – as boundaries. In most cases the domains with linear boundaries can be examined [7, 27], but there are some methods that allow to test nonlinear boundaries, too [29, 30].

4 Test Suite Reduction Technique
In this section we demonstrate and explain the main four pre and post test case reduction techniques.

4.1 CBR (Case-Based Reasoning) Deletion Algorithms Technique (Post-Process)
Removing all redundancy test cases is desirable, so many approaches introduced to reduce redundancy test cases. Applying an artificial intelligent concept in the test case reduction process is an innovated approach [9].

Case-based reasoning (CBR) is defined by Barry [16] as “one of the Artificial Intelligence-based algorithms, which solve the problems by searching through the case storage for the most similar cases. CBR has to store their solved cases back to their
memory or storage in order to learn from their experience.” “Case Base is a collection of cases in CBR, which can be defined as the following: Given a case - base \( C = \{c_1... c_n\} \), for \( c \in C \) whereas \( C = CBR \), \( c = \text{case} \) [16].

Here we introduce three reduction methods that apply CBR deletion algorithms: TTCF, TCIF and PCF methods. These methods aim to reduce a number of test cases generated by path-oriented test case generation technique. Path coverage is described by the control flow graph, which is derived from the source-code (program). Let us suppose that \( S = \{s_1, s_2, s_3, s_4, s_5\} \) to be a set of states in the control flow graph as in figure 1 below, where each state represents a block of code[9].

![Fig. 1 An Example of Control Flow Graph](image)

From the above figure, we assume that each state can reveal a fault. Thus, an ability to reveal faults of five states is equal to 5. Also, it is assumed that every single transaction must be tested. We will use this example in the three methods of CBR [9]. Let TCn = \( \{s_1, s_2, ... s_n\} \) where TC is a test case and \( s_n \) is a state or node in the path-oriented graph that is used to be tested. From the above figure, a set of test cases can be derived as showed in Table 1.

<table>
<thead>
<tr>
<th>Table 1 Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1 = ( {s_1, s_2} )</td>
</tr>
<tr>
<td>TC2 = ( {s_1, s_3} )</td>
</tr>
<tr>
<td>TC3 = ( {s_1, s_4} )</td>
</tr>
<tr>
<td>TC4 = ( {s_1, s_2, s_3} )</td>
</tr>
<tr>
<td>TC5 = ( {s_1, s_3, s_5} )</td>
</tr>
</tbody>
</table>

4.1.1 Test Case Complexity for Filtering (TCCF)

A complexity of test case is the significant criteria in this proposed method. It measures a number of states included in each test case. Let Cplx(TC) = \{High, Medium, Low\} where Cplx is a complexity of test case, TC is a test case. The complexity value can be measured as [9]:

- High when a number of states are greater than an average number of states in test suites.
- Medium when a number of states are equal to an average number of states in test suites.
- Low when a number of states are less than an average number of states in the test suites.

First, we should produce an auxiliary set from the test suite above. Auxiliary set removes test cases that don’t have a direct effect on the ability to reveal faults when it is removed. Therefore, the auxiliary set in our example is as follows [9]: Auxiliary set = \( \{TC1, TC2, TC3, TC4, TC5, TC6, TC9, TC10, TC11, TC12, TC13\} \)

We can notice that TC7 and TC8 are being removed. Afterward, the method computes a complexity value for all test cases in the above auxiliary set. From figure 1 and the test suite that contain 13 test cases, the average number of states is equal to 3. Therefore, the complexity value for each test case can be computed as follows:

- Cplx(TC1) = Low
- Cplx(TC2) = Low
- Cplx(TC3) = Low
- Cplx(TC4) = Medium
- Cplx(TC5) = Medium
- Cplx(TC6) = Medium
- Cplx(TC9) = Low
- Cplx(TC10) = Medium
- Cplx(TC11) = Low
- Cplx(TC12) = Low
- Cplx(TC13) = Medium

Finally, the last step removes test cases with minimum complexity value from the auxiliary set, which they are TC1, TC2, TC3, TC9, TC11 and TC12. Thus the reduced test suite will be: TC4, TC5, TC6, TC10 and TC13 [9].

4.1.2 Test Case Impact for Filtering (TCIF)

Due to the fact that defining and measuring a quality of software is important and difficult, the impact of inadequate testing must not be ignored. The impact of inadequate testing could be lead to the problem of poor quality, expensive costs and huge time-to-market. In conclusion, software testing engineers require identifying the impact of each test case in order to acknowledge and understand clearly the impact of ignoring some test cases. An impact value is considered here as an impact of test cases in term of the ability to detect faults if those test cases are removed and not be tested [9].

Let Imp(TC) = \{High, Medium, Low\} where Imp is an impact if a test case is removed, TC is a test case and the impact value can be measured as:

- High when the test case has revealed at least one fault for many times.
- Medium when the test case has revealed faults for only one time.
- Low when the test case has never revealed faults.

The procedure of this method is similar to the previous method. The only different is that this method aims to use an impact value instead of
complexity value. The impact value is computed for all test cases in the above auxiliary set, which is \{TC1, TC2, TC3, TC4, TC5, TC6, TC9, TC10, TC11, TC12, TC13\}. Based on figure 1, the impact value for each test case can be computed as follows:

\[
\begin{align*}
\text{Imp} (\text{TC1}) &= \text{Low}, \\
\text{Imp} (\text{TC2}) &= \text{High}, \\
\text{Imp} (\text{TC3}) &= \text{Medium}, \\
\text{Imp} (\text{TC4}) &= \text{Low}, \\
\text{Imp} (\text{TC5}) &= \text{High}, \\
\text{Imp} (\text{TC6}) &= \text{Medium}, \\
\text{Imp} (\text{TC9}) &= \text{Low}, \\
\text{Imp} (\text{TC10}) &= \text{Low}, \\
\text{Imp} (\text{TC11}) &= \text{Low}, \\
\text{Imp} (\text{TC12}) &= \text{Low}, \\
\text{Imp} (\text{TC13}) &= \text{Low}
\end{align*}
\]

Finally, test cases with minimum of impact value are removed from the auxiliary set. They are TC1, TC4, TC9, TC10, TC11, TC12 and TC13. Thus the reduced test suite will be: TC2, TC3, TC5, TC6 [9].

4.1.3 Path Coverage for Filtering (PCF)

The advantage of path coverage is that it takes responsible for all statements as well as branches across a method. It requires very thorough testing and used as a coverage value in this technique. The coverage value can specify how many nodes that the test case can cover. In other words, the coverage value is an indicator to measure nodes that each test case covers. It means that the higher coverage value is, the more nodes can be contained and covered in the test case.

Let Cov(n) = value, where Cov is a coverage value, value is a number of test cases in each coverage group and n is a coverage relationship.

The first step in this procedure is to identify a coverage set, which can be identified as follows (based on figure 1 above and the set of test cases that derived from it):

Coverage (1) = \{TC1\}
Coverage (2) = \{TC2\}
Coverage (3) = \{TC3\}
Coverage (4) = \{TC1, TC4, TC9\}
Coverage (5) = \{TC2, TC5, TC11\}
Coverage (6) = \{TC3, TC6, TC12\}
Coverage (7) = \{TC1, TC4, TC7, TC9, TC10, TC11\}
Coverage (8) = \{TC3, TC6, TC8, TC11, TC12, TC13\}
Coverage (9) = \{TC9\}
Coverage (10) = \{TC9, TC10, TC11\}
Coverage (11) = \{TC11\}
Coverage (12) = \{TC12\}
Coverage (13) = \{TC11, TC12, TC13\}

The next step is to calculate a coverage value based on a number of test cases in each coverage group. Therefore, the coverage value can be computed as follows:

Cov (1) = 1, Cov (2) = 1, Cov (3) = 1, Cov (4) = 3, Cov (5) = 3, Cov (6) = 3, Cov (7) = 6, Cov (8) = 6, Cov (9) = 1, Cov (10) = 3, Cov (11) = 1, Cov (12) = 1 and Cov (13) = 3.

The last step removes all test cases with minimum coverage value, in the potential removal set, that they are: TC1, TC2, TC3, TC9, TC11 and TC12. Thus the reduced test suite will be: TC4, TC5, TC6, TC7, TC8, TC10 and TC13 [9].

4.2 GE & GRE Heuristics and Priority Cost Technique (Post-Process)

GE and GRE heuristics algorithm have been proposed by Chen and Lau [20], Chen et al. defined essential test cases as the opposite of redundant test cases. If a test requirement ri can be satisfied by one and only one test case, the test case is an essential test case. On the other hand, if a test case satisfies only a subset of the test requirements satisfied by another test case, it is a redundant test case [10]. Based on these concepts, the GE and GRE heuristics can be summarized as follows [10]:

**GE heuristic:** first select all essential test cases in the test suite; for the remaining test requirements, we use the additional greedy algorithm, i.e. select the test case that satisfies the maximum number of unsatisfied test requirements.

**GRE heuristic:** first remove all redundant test cases in the test suite, which may make some test cases essential; then perform the GE heuristic on the reduced test suite. A mathematical formula is proposed to reduce the cost of testing by minimizing the size of the test suite using priority based cost. The priority factor will be calculated based on weighted set coverage, the cost of test requirements and test cases [11]. Let us consider the test cases \( T = \{t_1, t_2, t_3, t_4, t_5, t_6\} \) and let requirements of test cases are \( R = \{R_1, R_2, R_3, \ldots, R_{10}\} \).

Requirements according to the test cases (requirements satisfied by each test case) are:

- \( t_1 = \{R_1, R_2, R_3, R_5, R_6, R_10\} \)
- \( t_2 = \{R_1, R_2, R_4, R_5, R_{10}\} \)
- \( t_3 = \{R_6, R_8\} \)
- \( t_4 = \{R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9, R_{10}\} \)
- \( t_5 = \{R_3, R_5, R_7, R_8, R_{10}\} \)
- \( t_6 = \{R_3, R_4, R_5, R_6, R_8, R_9, R_{10}\} \)

After deriving the test cases from the test requirements, each requirement cost (C) is derived and computed from the summation of coverage (such as state coverage, edge coverage or branch coverage), high cost for a requirement means high degree of coverage.
From Table 2 [11], we calculate the cost of each test case, by taking the summation of cost of requirements (that it satisfies) as follows:

\[
\begin{align*}
\text{Cost (t1)} &= 2+1+3+2+1+3 = 12 \\
\text{Cost (t2)} &= 2+1+1+2+3 = 9 \\
\text{Cost (t3)} &= 1+3 = 4 \\
\text{Cost (t4)} &= 2+1+3+1+2+1+3+1+3 = 18 \\
\text{Cost (t5)} &= 3+2+1+3+3 = 12 \\
\text{Cost (t6)} &= 3+1+2+1+3+1+3 = 14.
\end{align*}
\]

Next, we check for unnecessary and redundant test cases, by applying GE and GRE heuristics as mentioned above. If not present, we then calculate the priority factor. We calculate the cardinality of the test cases (requirements satisfied by each test case)[11]:

\[
\begin{align*}
|\text{req}(t1)| &= 6, & |\text{req}(t2)| &= 5, & |\text{req}(t3)| &= 2, & |\text{req}(t4)| &= 10, & |\text{req}(t5)| &= 5, & |\text{req}(t6)| &= 7.
\end{align*}
\]

The priority of the test case \(ti\) is then calculated by this formula:

\[
\text{Priority (ti)} = \frac{\text{Cost (ti)}}{|\text{req}(ti)|}
\]

In our example, priorities for the sex test cases are:

\[
\begin{align*}
\text{Priority (t1)} &= 12/6 = 2, & \text{Priority (t2)} &= 9/5 = 1.8, & \text{Priority (t3)} &= 4/2 = 2, & \text{Priority (t4)} &= 18/10 = 1.8, & \text{Priority (t5)} &= 12/5 = 2.4, & \text{Priority (t6)} &= 14/7 = 2
\end{align*}
\]

Test cases with lower priority factor will be removed, so \(t2\) and \(t4\) are selected. Thus the reduced test suite will be: \(t1, t3, t5\) and \(t6\) [11].

4.3 Model-Checker Based Technique (Post-Process)

In this technique, we consider test-cases generated with model-checker based methods. A model-checker is a tool originally intended for formal verification. In general, a model-checker takes as input a finite-state model of a system and a temporal logic property and efficiently verifies the complete state space of the model in order to determine whether the property is fulfilled or not [12].

Redundancy is used to describe test-cases that are not needed in order to achieve a certain coverage criterion. As the removal of such test-cases leads to reduced fault detection ability, they are not really redundant in a generic way. In contrast, we say a test-case contains redundancy if part of the test-case does not contribute to the fault detection ability. We are going to identify such redundancy, and describe possibilities to reduce it [12].

Intuitively, identical test-cases are redundant. For any two test-cases \(t1, t2\) such that \(t1 = t2\), any fault that can be detected by \(t1\) is also identified by \(t2\) and vice versa, assuming the test-case execution framework assures identical preconditions for both tests. Similarly, the achieved coverage for any coverage criterion is identical for both \(t1\) and \(t2\).

Clearly, a test-suite does not need both \(t1\) and \(t2\) [12]. The same consideration applies to two test-cases \(t1\) and \(t2\), where \(t1\) is a prefix of \(t2\). \(t1\) is subsumed by \(t2\), therefore any fault that can be detected by \(t1\) is also detected by \(t2\) (but not vice versa). In this case, \(t1\) is redundant and is not needed in any test-suite that contains \(t2\). In model-based testing it is common practice to discard subsumed and identical test-cases at test-case generation time [12]. This kind of redundancy can be illustrated by representing a set of test-cases as a tree. The initial state that all test-cases share is the root-node of this tree. A sub-path is redundant if it occurs in more than one test-case. In the tree representation, any node below the root node that has more than one child node contains redundancy. If there are different initial states, then there is one tree for each initial state. The depth of the tree equals the length of the longest test-case in \(TS\). Children(x) denotes the set of child nodes of node \(x\).

Consider a test-suite consisting of three test-cases (letters represent distinct states):

\[
\begin{align*}
\text{t1: } &A\rightarrow B\rightarrow C, \\
\text{t2: } &A\rightarrow C\rightarrow B, \\
\text{t3: } &A\rightarrow C\rightarrow D\rightarrow E.
\end{align*}
\]

The execution tree representation of these test-cases can be seen in Figure 2(a) [12]. The rightmost C-state has two children, therefore the sub-path A-C is contained in two test-cases; it is redundant.

![Simple test-suite with redundancy represented as execution tree.](image)

Fig. 2: Simple test-suite with redundancy represented as execution tree.

The execution tree can be used to measure the redundancy \(R\) of test-suite \(TS\) based on the following relation:
R(TS) = 1/(n – 1) \sum_{x \in \text{children}(\text{root}(TS))} R(x) \quad (1)

The redundancy of the tree is the ratio of the sum of the redundancy values R for the children of the root-node and the number of arcs in the tree (n – 1, with n nodes).

The redundancy value R is defined recursively as following relation [12]:

\[ R(x) = (|\text{children}(x) – 1| + \sum_{c \in \text{children}(x)} R(c)) \text{ if } \text{children}(x) \neq \{\} \]
\[ 0 \text{ if } \text{children}(x) = \{\} \quad (2) \]

The example test-suite depicted as tree in Figure 2(a) has a total of 7 nodes, where one node besides the root node has more than one child, which is the node c. Therefore, the redundancy of this tree (based on relations 1 and 2) equals:

\[ R = \frac{1}{7-1} \cdot \sum_{x \in \text{children}(\text{root}(TS))} R(x) = \frac{1}{6} \cdot (0 + (1+0)) = \frac{1}{6} = 17\% \]

A test-suite contains no redundancy if for each initial state (root node) there are no test-cases with common prefixes, e.g., if there is only one test-case per initial-state. Figure 2(b) illustrates the result of an optimization applied to the Figure 2(a) [12] in order to remove redundancy. The test-cases A-C-B and A-C-D-E have the common prefix A-C, and there is a test-case ending in C, which is A-B-C. Therefore the postfix B of A-C-B is appended to A-B-C, resulting in A-B-C-B. Thus test suite with the three test cases is reduced to become test suite with two test cases after removing the redundancy [12].

4.4 Base Choice Coverage Criterion
Technique (Pre-Process)

The input domain to any program contains all the possible inputs to that program. In equivalence partitioning technique, the domain for each input is partitioned into regions (partitions), and each partition defines a set of blocks that must be pair wise disjoint (no overlap) and covers the domain of each partition (complete), as we can see in figure 3 [15].

![Fig. 3 three blocks for a partition which are disjoint and complete](image)

An important question would be: “How should we consider multiple partitions at the same time?” This is the same as asking “What combination of blocks should we choose values from?” The most obvious choice is to choose all combinations. However, using all combinations will be impractical when more than 2 or 3 partitions are defined [15]. For example, if we have three partitions with blocks [A, B], [1, 2, 3] and [x, y]. Table 3 shows the twelve test cases are needed for all combinations coverage.

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 : (A, 1, x)</td>
<td></td>
</tr>
<tr>
<td>t2 : (A, 1, y)</td>
<td></td>
</tr>
<tr>
<td>t3 : (A, 2, x)</td>
<td></td>
</tr>
<tr>
<td>t4 : (A, 2, y)</td>
<td></td>
</tr>
<tr>
<td>t5 : (A, 3, x)</td>
<td></td>
</tr>
<tr>
<td>t6 : (A, 3, y)</td>
<td></td>
</tr>
<tr>
<td>t7 : (B, 1, x)</td>
<td></td>
</tr>
<tr>
<td>t8 : (B, 1, y)</td>
<td></td>
</tr>
<tr>
<td>t9 : (B, 2, x)</td>
<td></td>
</tr>
<tr>
<td>t10 : (B, 2, y)</td>
<td></td>
</tr>
<tr>
<td>t11 : (B, 3, x)</td>
<td></td>
</tr>
<tr>
<td>t12 : (B, 3, y)</td>
<td></td>
</tr>
</tbody>
</table>

Ammann and Offutt [13] advocated base choice coverage criterion as the minimum adequate criterion. They argued that each system has a normal mode of operation and that normal mode corresponds to a particular choice in each category (partition). This particular choice (block) is called as base choice. Thus base – choice - coverage criterion requires that each choice in a category be tested by combining it with the base choice for all other categories. This causes each non-base choice to be used at least once, and the base choices to be used several times [14].

We simply ask: What is the most “important” block for each partition in our domain? This block is called the “base choice” [15]. For our example above, we suppose that base choice block in partition [A, B] is A, in partition [1, 2, 3] is 1 and in partition [x, y] is x. Then a base choice test case and additional test cases would be like the following [15]:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 : (A, 1, x)</td>
<td>which is called the base test</td>
</tr>
<tr>
<td>t2 : (B, 1, x)</td>
<td></td>
</tr>
<tr>
<td>t3 : (A, 2, x)</td>
<td></td>
</tr>
<tr>
<td>t4 : (A, 3, x)</td>
<td></td>
</tr>
<tr>
<td>t5 : (A, 1, y)</td>
<td></td>
</tr>
</tbody>
</table>

As we can see, in base choice coverage criterion the number of test cases are reduced compared with all combinations coverage criterion. This is because of choosing a base choice block for each partition we have. Which blocks are chosen for the base choices becomes a crucial step in test design that can greatly impact the resulting test [15].
Figure 4 represents an example of equivalence partitioning for a specific inputs, organized using the equivalence partitioning organizer [33] (written by Martin Keesen –version 0.5). The organizer allows us to create partitions with their valid and invalid values (blocks). In the above example, there are three partitions: Foreground color, Background color and Outlining partitions. Each has their own valid and invalid values. From the edit menu, we choose: Auto create test cases, and then the test cases above will be generated automatically based on a coverage criteria called: Each choice coverage, which requires that: one value from each block for each partition must be used in at least one test case[15]. So we notice that the four test cases (TC1-TC4) have covered all valid blocks each at least one and the last five test cases form (TC5-TC9), represent possible combination between one invalid block with two other valid blocks from the different partitions.

### 4.5 Pros and Cons of Test Case Reduction Techniques

There are many research challenges and gaps in the test case reduction area. Those challenges and gaps can give the research direction in this field. However, the research issues that motivated this study are: the too many redundancy test cases after reduction process, a decrease of test cases ability to reveal faults and the uncontrollable grow of test cases [9]. Table 4 summarizes the advantages and limitations of the aforementioned test suite reduction techniques:

<table>
<thead>
<tr>
<th>Technique / Algorithm</th>
<th>Advantages</th>
<th>Limitations</th>
</tr>
</thead>
</table>
| CBR algorithms        | - Preserving capability to detect faults after reduction (especially TCCF and TCF) [9].  
- Removing the redundancy and unnecessary test cases[11]  
- Controlling the growth of test cases [11]. | - Require a lot of time.(specially TCCF and TCF) [9].  
- The path coverage may be not an effective coverage factor for a huge system that contains million lines of code. This is because it requires an exhaustive time and cost for identifying coverage from a huge amount of codes [9]. |
- Reduce the redundant and unnecessary test cases [11]. | - The NP-complete problem [11].(that is no fast solution is known) |
| Model-Checker          | - A convenient tool for optimization purposes and removing redundancy, especially if it is already used for test-case generation in the first place [12].  
- Quality of the resulting test-suites does not suffer with regard to test coverage or fault detection ability [12]. | - Not an effective for a huge system that contains million lines of code [9]. Because this will be costly and time consuming. |
| Base choice criteria (equivalence partitioning) | - Fairly easy to get started, because it can be applied with no automation and very little training [15].  
- Simple to tune the technique to get more or fewer tests [15]. | - Quality of the resulting test-suite may suffer or be not efficient in revealing defects, because choosing base choices is crucial step that depends on the tester. |
5 Conclusion and Future Work

The size of a test-suite has a direct impact on the costs and the effort of software testing. Especially during regression testing, when software is re-tested after some modifications, the size of the test-suite is important. Common test-suite reduction techniques select subsets of test-suites that achieve given test requirements. Unfortunately, not only the test-suite size but also the fault detection ability is reduced as a consequence. Hence we must strive to produce an efficient test suite so that test-cases are not simply discarded, but the impact on the fault sensitivity must be minimal. As future work we intend to implemented the four test suite reduction techniques in our study and perform an experimental comparison of them in term of reducing the size of a test suite while maintaining its fault detection effectiveness.

References:

[9] Roongruangsuwan, S. & Daengdej, J., Test Case Reduction Methods by Using CBR, Autonomous System Research Laboratory Faculty of Science and Technology Assumption University, Thailand.


