Using Program Slicing to Improve the Efficiency and Effectiveness of Cluster Test Selection

Zhenyu Chen, Yongwei Duan, Zhihong Zhao, Baowen Xu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, Jiangsu 210093, China

Ju Qian
College of Information Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210016, China

Cluster test selection is a new successful approach to select a subset of the existing test suite in regression testing. In this paper, program slicing is introduced to improve the efficiency and effectiveness of cluster test selection techniques. A static slice is computed on the modified code. The execution profile of each test case is filtered by the program slice to highlight the parts of software affected by modification, called slice filtering. The slice filtering reduces the data dimensions for cluster analysis, such that the cost of cluster test selection is saved dramatically. The experiment results show that the filtering techniques could reduce the cost of cluster test selection significantly and could also improve the effectiveness of cluster test selection modestly. Therefore, cluster test selection by filtering has more potential scalability to deal with large software.

Keywords: Regression testing; program slicing; cluster analysis; cluster test selection; dimensionality reduction

1. Introduction

Regression testing is an important phase in software development lifecycle. Developers use regression testing to validate modified software and confirm that no new fault has been introduced into previously verified code. Typically, regression testing involves executing a large number of test cases, such that it is very time-consuming. Therefore, regression testing is often performed with a time budget and it does not allow executing all test cases. Due to the expensiveness of regression testing, test selection techniques are employed to reduce the cost of retesting a modified program by selecting a subset of existing test suite to run.
In recent years, cluster analysis has been introduced in regression test selection, called Cluster Test Selection (CTS) [6, 7, 24, 30]. Cluster analysis is the assignment of a set of items into subsets (called clusters) so that these items in the same cluster are similar in some sense. In regression testing, cluster analysis is used on the execution profiles induced by the original test cases. The fundamental idea behind CTS is that test cases with similar behaviors would be grouped into the same cluster. It was expected that the fault-revealing test cases could be grouped into the same cluster. Finally, a certain sampling strategy is used to select a subset of the original test suite.

Many properties of the program’s executions could be profiled to characterize the behaviors of test cases. Such properties include: statement, block, function, data-flow, event sequence, etc. [14, 26]. Most of the existing efforts on CTS [6, 7, 24, 30] use a simple execution profile, function call profile, for cluster analysis. In some cases, function call profile may be difficult to capture the detailed behaviors of test cases [14]. On the other hand, a large software may contain ten thousands even millions of statements. If a detailed execution profile, such as statement coverage profile, is used, then it may result in an unacceptable performance of cluster analysis with high dimensional data.

The dimensionality reduction is a key issue of the application of many machine learning tasks, including cluster analysis. Most of the existing dimensionality reduction techniques are based on some mathematic approaches, which are independent of the application domain [8, 31]. These domain-independent techniques for dimensionality reduction are more general, but the efficiency and effectiveness may be limited in a special domain. In this paper, we would complete and extend the slice filtering technique presented in [9]. A domain-dependent technique for dimensionality reduction is proposed for CTS. A common program analysis technique, program slicing, is introduced to remove the parts of software irrelevant to modification, called slice filtering. The dimensions of the execution profiles, such as statements, can be reduced for efficient cluster analysis. It provides more potential scalability of CTS to deal with large software. On the other hand, program slicing can remove the code irrelevant to the modification, such that CTS can highlight the parts of software affected by modification. This will produce a better result of cluster analysis, as well as CTS.

The remainder of this paper is organized as follows. In the next section, a background of regression test selection is introduced. In Section 3, our approach is described and the key technique, slice filtering, is proposed. The detailed implementation of our approach is explained in Section 4. The experiment and its result analysis are done in Section 5. The related work is discussed in Section 6. The last section concludes this study.
2. Background

We use the following notations throughout the rest of this paper: $P$ denotes the original version of a program, $P'$ denotes its modified version, $T = \{t_1, t_2, \cdots, t_n\}$ denotes the original test suite for $P$, $T'$ denotes a subset of $T$ for $P'$. $P(t)$ and $P'(t)$ denote the test results of $P$ and $P'$ with the test case $t$, respectively. In order to guide the test selection, we will collect some entity coverage information. Let $E = \{e_1, e_2, \cdots, e_m\}$, in which $e_i$ denotes the $i$th entity, such as statement, block, etc. A coverage matrix $C = T \times E$ will be constructed after all test cases are executed. The $n \times m$ matrix is a 0-1 matrix, in which $c_{ij} = 1$ if the test case $t_i$ executed the entity $e_j$, otherwise $c_{ij} = 0$. $C_i$ denotes the $i$th row of $C$ and $C_i$ is called the feature of $t_i$.

Definition 1 (Regression Test Selection [17]). Given a program $P$ and its modified version $P'$, the problem is to select a subset $T'$ of the original test suite $T$, such that $T'$ can detect the faults introduced by the modification in $P'$. A test case is fault-revealing if it detects a failure in a program. A test case is modification-traversing if and only if it executes the modified code, or executes formerly executed code that has been deleted. There is an assumption in regression test selection: for each test case $t \in T$, $P(t)$ is correct. Therefore, $t$ is a fault-revealing test case if and only if $P(t) \neq P'(t)$. We use $T_F$ to denote the set of all fault-revealing test cases in $T$, $T_M$ to denote the set of all modification-traversing test cases in $T$. It is not difficult to see that $T_F \subseteq T_M \subseteq T$. That is, each fault-revealing test case is a modification-traversing test case. And a modification-traversing test case may not be a fault-revealing test case.

A test selection technique is called safe if it can select all fault-revealing test cases, that is it can select a subset $T' \supseteq T_F$. It is clear that $T_M$ is a safe one because $T_F \subseteq T_M$. In order to select the $T_M$, some program analysis techniques of the source code are used. Rothermel et. al proposed a safe selection technique [18], which constructed control flow graphs for a program $P$ and its modified version $P'$, and used these graphs to select $T_M$ from $T$. In some cases, $T_M$ is much larger than $T_F$. There are many redundant test cases in $T_M$. Running all test cases in $T_M$ with a time limit is still infeasible in some cases.

3. Approach

A main challenge of regression test selection is how to select $T'$ approximating to $T_F$. There are many existing efforts to address this problem [9, 16, 17, 18, 19]. Cluster Test Selection (CTS) is one of the successful techniques [6, 7, 9, 24, 26, 30]. Please note that CTS is not safe due to the indeterminacy of cluster analysis and randomness in a sampling strategy. This un-safeness means our approach may miss some fault-revealing test cases in $T_F$. However, in a resource-limited testing scenario, running all the test cases selected by a safe regression test selection technique can be impractical. On the other hand, an appropriate subset of all fault-revealing test
cases is sufficient to detect and debug the faults in many cases. In this paper, we adopt the framework of CTS [30] and filter the features by program slicing to improve the efficiency and effectiveness of CTS.

3.1. Framework

CTS introduces a clustering algorithm to separate $T$ into some groups. The fundamental idea behind CTS is that the similar test cases could be grouped into the same cluster by some certain behavior features. And then a few test cases will be sampled from each cluster to form a regression test suite $T'$. As shown in Figure 1, CTS contains the following steps:

1. **Running $T$**: All test cases in $T$ are executed for the program $P$. The execution profiles of test cases will be collected for the next steps. In this step, many types of profiles, such as function call profile, statement/block coverage profile, etc., could be generated for each test case.

2. **Filtering**: The execution profile contains a set of entities (statement or block, etc.) that is executed during a run. For a large software, the dimension of feature $|C_i|$, i.e. $m$, can be huge, such that it cannot be handled in practice. Therefore, each feature $C_i$ will be filtered to a small subset to improve the performance of cluster analysis in next step. In this paper, we use program slicing to filter the feature of each test case.

3. **Cluster analysis**: The inputs to cluster analysis are the features filtered in step 2. The distance of each pair of features should be calculated before clustering. A clustering algorithm is chosen to group the test cases in $T$ into some clusters. Clustering techniques put test cases with similar execution profiles into the same cluster.
4. **Sampling**: An effective sampling strategy, called adaptive sampling, is used in this paper. It first randomly selects a few of test cases from each cluster. Then, these test cases are executed to verify whether they are passed or failed. If a test case is failed, then all the other test cases in the same cluster will be selected into $T'$. Previous research shows that adaptive sampling strategy is more effective than other sampling strategies [6, 7, 24].

3.2. **Slice Filtering**

For each test case $t_i$ in $T$, there is a corresponding feature $C_i$, in which $C_i = \{c_{i1}, c_{i2}, \ldots, c_{in}\}$. $c_{ij}$ is 1 or 0 to represent it is executed by $t_i$ or not. The efficiency and effectiveness of cluster analysis largely depends on the size and the quality of features. We use statement coverage profiles and block coverage profiles to construct the features of test cases, that is the entity $c_{ij}$ in $C_i$ represents a statement or a block. However, there is a huge number of entities in a large software. A natural question is whether all entities are useful to CTS with regard to modification. This also motivates us to remove some useless entities, such that the efficiency and effectiveness of CTS can be improved.

<table>
<thead>
<tr>
<th>No.</th>
<th>Statement</th>
<th>Slice</th>
<th>$t_1$ $t_2$ $t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void f (int m, int n)</td>
<td>*</td>
<td>(1,6) ● ● ● ● ●</td>
</tr>
<tr>
<td>2</td>
<td>int a=0, b=0, c=0, d=0;</td>
<td>●</td>
<td>● ● ● ● ●</td>
</tr>
<tr>
<td>3</td>
<td>if (m&lt;n)</td>
<td>*</td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>4</td>
<td>a=1;</td>
<td>*</td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>5</td>
<td>b=1;</td>
<td></td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>6</td>
<td>while(a&gt;0){ /* (a&gt;0)*/</td>
<td>*</td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>7</td>
<td>c=c+a;</td>
<td>*</td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>8</td>
<td>a=a-1;}</td>
<td>*</td>
<td>(2,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>9</td>
<td>printf(&quot;%s&quot;,c);</td>
<td>*</td>
<td>(7,0) ● ● ● ● ●</td>
</tr>
<tr>
<td>10</td>
<td>if(a&gt;0){</td>
<td>○</td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
<tr>
<td>11</td>
<td>b=a-5;</td>
<td>○</td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
<tr>
<td>12</td>
<td>while(b&gt;0){</td>
<td></td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
<tr>
<td>13</td>
<td>d=d+b;</td>
<td>○</td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
<tr>
<td>14</td>
<td>b=b-1;}</td>
<td>○</td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
<tr>
<td>14</td>
<td>printf(&quot;%s&quot;,d);}</td>
<td></td>
<td>(7,0) ○ ○ ○ ○</td>
</tr>
</tbody>
</table>

**Note**: The modification is in the statement $s_5$. * denotes the statement in the program slice. ● denotes the statement remained by slice filtering. ○ denotes the statement removed by slice filtering.

Program slicing is one of the most useful techniques for simplifying programs by focusing on selected aspects of computing. The process of program slicing deletes the parts of the program which have no effect upon the aspects of interest. In this paper,
we use static slice to filter the features of test cases [2, 23, 29]. For a statement s and a set of variables \( \{v\} \), the slice \( S \) of program \( P \) with respect to the slicing criterion \(< s, \{v\} >\) includes only those statements of \( P \) needed to capture the computing of \( \{v\} \) at \( s \). In other words, the slice of program includes those statements that are directly or indirectly dependent on the values of variables in \( \{v\} \) at statement \( s \). In this paper, we construct two forms of slice: backward slice and forward slice. A backward slice contains the statements of the program which can have some effect on the slicing criterion. A forward slice contains those statements of the program which are affected by the slicing criterion [29]. To determine the impact of modifications, we use the union of the backward slice and the forward slice as a final slice \( S \). It is clear that \( S \) is a subset of all statements.

We consider 3 different kinds of changes [22] in \( P' \): define change, use change and control change. Informally, a define change is a change on the left-hand side of an assignment statement; a use change is a change on the right-hand side of an assignment. Given a statement \( s_i \) in \( P \): \( x = z \) is changed to \( y = z \), the union set of the following 3 slices is used: forward slice at \(< s_i, \{x\} >\), forward slice at \(< s_i, \{y\} >\) and backward slice at \(< s_i, \{z\} >\). In the case of use change, if a statement \( s_i \) \( x = y \) in \( P \) is changed to \( x = z \) in \( P' \), then the union set of the forward slice at \(< s_i, \{x\} >\) and backward slice at \(< s_i, \{z\} >\) is used. Control change is defined as follows: a change is a control change if it affects the original control-flow. Control changes require special treatment. We compute the slice of control change in 3 steps: (1) We compute the backward slice of changed statement. (2) We compute all the control dependent of the changed statement and compute the corresponding forward slices. (3) we add in the modified statement if it is not included in step 2.

We construct a simple program to demonstrate the necessity of slice filtering using program slice, as is shown in Table 1. Suppose the statement \( s_5 “ \text{while} \ (a \geq 0) “ \) is changed to \( “ \text{while} \ (a > 0) “ \). The program slice \( S \) with the slice criterion \(< s_5, \{a\} >\) is the set of statements \( S = \{s_1, s_2, s_3, s_5, s_6, s_7, s_8\} \). Given three test cases: \( t_1 = (1, 6), t_2 = (2, 0), t_3 = (7, 6) \), the execution profiles (statement coverage profiles here) of test cases are shown in Table 1. \( t_1 \) and \( t_3 \) are similar since they share many statements in their execution profiles. \( t_2 \) is dissimilar to both \( t_1 \) and \( t_3 \). Hence, \( t_1 \) and \( t_3 \) are more similar than \( t_2 \) and \( t_3 \) in cluster analysis. However, if we focus on the program slice, not all statements in program, then the result is quite different. We compute the intersection of \( S \) and each feature \( C_i \), as the “•” shown in Table 1. The features of \( t_2 \) and \( t_3 \) are actually the same by filtering, and the difference between \( t_2 \) and \( t_3 \) is magnified since those statements irrelevant to the statement \( s_5 \) are filtered out. Since \( S \) is a subset of statements in the program with regard to the modification, it is excepted that the dimension of feature can be reduced and the affected parts are still remained.

The program slice \( S \) could be considered as the representation for the impact...
of modification. Hence, a natural idea is to use $S$ to filter the execution profiles of test cases, such that the features become more compact and they are related to the modification. To filter out the unaffected statements with regard to the modification, we use the intersection of the feature $C_i$ and the slice $S$, denoted by $C_i'$. This intersection of static slice and the execution profile could be considered as a lightweight dynamic slice. The exact dynamic slice is more precise than static slice. But the computation of dynamic slice is usually more complex and time-consuming than static slice. The execution profiles, such as covered statements, are collected during regression testing. Assisted by a low cost static slice, the parts of software affected by modification are computed easily.

4. Implementation

In this section, we describe the detailed implementation of our CTS techniques with filtering, including execution profiles collection, filtering, cluster analysis and sampling. The setting of some key parameters in our experiment will also be explained.

4.1. Execution Profile Collection

We implemented our approach based on the statement coverage and the block coverage respectively. The original test suite $T$ was executed on the program $P$. The coverage information was collected using GNU call-coverage profiler, gcov [11]. This coverage information was then stored to form a coverage matrix $M$, respectively. CTS techniques conducted on the original matrix or on the resulting matrix by filtering. We would use the size of matrix to evaluate the efficiency of various CTS techniques in the next section. We also stored the mapping information from the statements to the blocks for further analysis. For simplicity, we mainly discuss the implementation based on the statement coverage. The implementation based on the block coverage is similar.

4.2. Filtering

The size of coverage matrix plays a key role in the performance of CTS techniques. And the quality of coverage matrix plays a key role in the effectiveness of CTS techniques. In order to achieve a sufficient filtering result, we implemented the whole filtering process in three steps: the preprocess of filtering, slice filtering and the postprocess of filtering.

4.2.1. Preprocess of Filtering

As described in Section 2, $T_F \subseteq T_M$, that is a test case not modification-traversing must not be a fault-revealing one. Hence, these test cases could be removed safely. This is so-called safe regression test selection [18, 19]. The process of safe selection is this:
(1) The control flow graphs of the original and modified programs are constructed.
(2) The two graphs are traversed synchronously in a depth-first order to identify the modified nodes, which means the corresponding edges are risky.
(3) Test cases traversing the risky edges are chosen.

As shown in [30], there are many redundant test cases using such safe selection techniques in most cases, although it contains all fault-revealing test cases. In our experiment, we used the tool DejaVu [18,19] to select the modification-traversing test cases as a preprocess of filtering. In this step, we got a new matrix $M_1$, which was from $M$ by safe selection. That is, we only kept modification-traversing test cases and their features in $M$.

4.2.2. Slice Filtering

In this step, we used program slice to do two types of filtering: column filtering and row filtering. Given the modified statements in $P'$, we set the corresponding slice criterion for each statement $<s_i,\{v\}>$. The forward slice and the backward slice were computed respectively. And then the union of the forward slice and the backward slice forms the program slice $S$. The matrix $M_1$ was filtered in columns by $S$. That is, some columns were removed from the matrix $M_1$ if the corresponding statements are not in $S$. Then we got a new matrix $M_2$, which was from $M_1$ by column filtering with $S$. We would do a further row filtering for $M_2$. The common parts between the program slice $S$ and the feature $C_i$ indicate the fault detection capability of $t_i$ in some sense. We set a threshold, called filtering rate $FR$ [9], to determine the operation of filtering. If the intersection of the program slice $S$ and the execution profile $C_i$ is low, then $t_i$ is lowly relevant to the modification. Formally, if $|S \cap C_i|/|S| < FR$, then $C_i$ would be filtered out from the matrix. In our experiment, we set $FR = 0.3$ based on the experience from [9]. Then we got a new matrix $M_3$, which was from $M_2$ by row filtering.

4.2.3. Postprocess of Filtering

In this step, we conducted a simple column filtering to further compact the matrix. We observed that some statements were executed by all test cases (such as the statement $s_1$ in Table 1) and some statements were not executed by any test case. Using such statements as feature elements could not distinguish the behaviors among test cases in some sense. Therefore, these statements were filtered out from the matrix. That is, if the numbers in a column are all 1 or are all 0, this column would be removed from the matrix. Finally, we got a new $n' \times m'$ matrix $M'$, which was from $M_3$ by column filtering.
4.3. Cluster Analysis

In the cluster analysis phase, a test case $t_i$ is represented as a feature $C_i'$. Euclidean distance was used as the distance function to each pair of features $C_i'$ and $C_j'$ as follows:

$$D(C_i', C_j') = \sqrt{\sum_{k=1}^{m'} (c_{ik}' - c_{jk}')^2}$$  \(1\)

The simple K-means algorithm was employed as the cluster algorithm in our experiment. We adopted a widely used machine learning tool Weka [27] to implement the cluster analysis. The K-means algorithm requires the initial cluster number as a parameter. In our approach, this number was set according to the numbers of test cases, i.e. the number of rows in $M'$, $n'$. Let $CN$ denote this initial cluster number, then $CN = n' \cdot p$, where $0 < p < 1$. In our experiment, we set $p = 0.025$ from the experience in [30].

4.4. Sampling

It was expected that similar test cases were grouped into the same cluster by cluster analysis. According to previous studies [6, 7, 14, 30], if we used appropriate distance metrics to describe each pair of features, then test cases with unusual behaviors tended to be isolated within the same cluster. And the next step was to sample out a subset of test cases which mostly consists of failed test cases.

We used a popular strategy, namely adaptive sampling strategy [6, 7], in our approach. We first sampled a certain number of test cases with a pre-defined sampling rate, denoted by $SR$ hereinafter. Then the sampled test cases with those selected features were run to check whether they were fault-revealing. If a fault-revealing test case was found, the entire cluster from which the test cases were sampled was selected. This strategy favors small clusters and had high probability to select fault-revealing test cases. In our experiment, we set $SR = 0.1$ and repeat the sampling task 10 times to calculate the corresponding average value.

5. Experiment

We use a subject program, space, from Software-artifact Infrastructure Repository (SIR) [20], as our subject program. Space has 5902 lines of code, 1533 basic-blocks. A total of 38 modified versions along with a base program. 8 versions were excluded because no or a handful of test cases could detect these faults. 30 modified versions were used in our experiment. For each version, the base program is augmented with a real fault. A total of 13585 test cases are supplied.

CTS techniques have certain limitation when dealing with small-sized program [30]. This is because execution profiles of test cases on small programs tend to be similar, such that they are difficult to distinguish by execution profiles. Therefore,
the Siemens programs in SIR [20] are not adopted in our experiment because their sizes are too small.

5.1. Efficiency Analysis

In the process of CTS, the cost of cluster analysis is the main part of the whole cost, especially for large software. K-means was used in our CTS techniques. The time complexity of K-means is modest, basically linear in \( n \), the number of items (i.e. test cases here) and \( m \), the number of attributes (i.e. statements or blocks here). It is expected that \( n \) and \( m \) can be reduced greatly by the filtering techniques.

5.1.1. Measures

Two simple reduction measures of cluster analysis, dimensionality reduction and matrix reduction, are introduced to evaluate the efficiency of CTS using filtering techniques. The dimensionality reduction was widely used in the applications of cluster analysis. But for CTS, the matrix reduction is more suitable, because the time complexity of K-means is linear in the size of coverage matrix, i.e. \( n \times m \).

**Dimensionality Reduction:** \( m \) is the number of original entities. \( m' \) is the number of resulting entities by filtering. Then the dimensionality reduction \( DR \) can be calculated as follows:

\[
DR = \frac{m'}{m} \times 100\%
\]  

(2)

Lower dimensionality reduction \( DR \) means that more cost can be saved in cluster analysis, as well as in CTS.

**Matrix Reduction:** The size of the original coverage matrix \( M \) is \( n \times m \). The size of the resulting coverage matrix \( M' \) by filtering is \( n' \times m' \). Then the matrix reduction \( MR \) can be calculated as follows:

\[
MR = \frac{n' \times m'}{n \times m} \times 100\%
\]  

(3)

Lower matrix reduction \( MR \) means that more cost can be saved in cluster analysis, as well as in CTS.

5.1.2. Results

Figure 2 and 3 respectively give the \( DRs \) and \( MRs \) using statement coverage in CTS. The experiment results show that the filtering techniques could achieve a great reduction for both \( DR \) and \( MR \). The mean of \( DR \) and \( MR \) are 24.77% and 10.55% respectively. Most \( DRs \) are from 15% to 30%. The \( DRs \) of some versions (1, 21 and 25) are larger than 40% but they are still modest. The dimensions of some versions (2, 3 and 4) are reduced greatly and they are smaller than 10%. 
The experiment results indicate that the size of coverage matrix could be reduced dramatically. Although the \( MRs \) of two versions (1 and 25) are larger than 40\%, most \( MRs \) are smaller than 10\% and even some (version 3, 5, 6, 16, 17, 19 and 29) are smaller than 1\%.

![Figure 2](image2.png)

**Fig. 2.** DR of Statement: Mean 24.77\%, Max 48.61\%, Min 2.93\%

![Figure 3](image3.png)

**Fig. 3.** MR of Statement: Mean 10.55\%, Max 48.50\%, Min 0.33\%

Figure 4 and 5 respectively give the \( DRs \) and \( MRs \) using block coverage in CTS. The experiment results show that the filtering techniques could also achieve a great
reduction for both $DR$ and $MR$. The mean of $DR$ and $MR$ are 39.75% and 16.56% respectively. Most $DR$s are smaller than 50% and two $DR$s (version 2 and 3) are smaller than 10%. Most $MR$s are smaller than 10% and even some (version 5, 6, 16 and 17) are smaller than 1%.

As the results shown in Figure 2-5, the matrix reduction $MR$ is smaller than the dimensionality reduction $DR$ for each version, and for both statement and block.
$DR$ only benefits from the column filtering and $MR$ benefits from both the row filtering and the column filtering. The results of statement are better than the results of block. However, we cannot claim that the filtering techniques are more useful for statement coverage than for block coverage. The number of statements 5902 is much larger than the number of blocks 1533. A large number of entities has potential chances for reduction. Hence, we believe that the filtering techniques are more useful for large software.

5.2. Effectiveness Analysis

In the previous subsection, the experiment results show that the filtering techniques could greatly reduce the cost of cluster analysis, as well as the cost of CTS. On the other hand, it is expected that the effectiveness of CTS could also be improved, at least the same as the CTS not using filtering techniques.

5.2.1. Measures

To evaluate CTS techniques with filtering, as well as other CTS techniques, we constructed two evaluation measures, one for test suite reduction, the other for comprehensive assessment of fault detection capability.

**Test Suite Reduction:** Regression test selection techniques aim to reduce the size of test suite, such that the cost of execution and maintenance could be saved. If an original test suite $T$ and a subset $T'$ is selected by a certain technique, the test suite reduction can be calculated as this:

$$TR = \frac{|T'|}{|T|}$$

A small $TR$ means a great reduction of test suite. Please note that $TR$ is different from $DR$ and $MR$ in the previous subsection. $TR$ is used to evaluate the effectiveness of CTS techniques. $DR$ and $MR$ are used to evaluate the efficiency of CTS techniques.

**F-measure:** Another main goal of regression test selection is to select a subset with strong fault detection capability. $T_F$ is the set of all fault-revealing test cases. If a CTS technique selects a subset of fault-revealing test cases, denoted by $T'_F$. Then the recall measure is defined as $recall = \frac{|T'_F|}{|T_F|}$, which indicates the fault detection capability. The precision measure is defined as $precision = \frac{|T'_F|}{|T'|}$, which is used to limit the number of redundant test cases. A higher recall and precision means a better result of CTS. However, the recall and the precision are naturally in conflict. Improving one usually hurts the other. We used F-measure to evaluate the integrative benefit of CTS techniques. F-measure relies on the traditional information retrieval measures, adapted for evaluating the results by considering the corresponding recall and precision pairs in this paper:
F = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \quad (5)

Since the recall and the precision are both in the range of [0, 1], the F-measure is also in the range of [0, 1]. And it is easy to find that the F-measure increases as the recall or the precision increases. As shown before, the recall and the precision need to be relatively large, so we want the F-measure to be large since it is relevant to both of them, monotonically increasing. Therefore, the F-measure can well demonstrate the integrative benefit of the techniques.

5.2.2. Results

In this subsection, we will make a comparison between CTS techniques with filtering and CTS techniques without filtering with regard to effectiveness. However, the original statement coverage profile is too large (13585×5902) to accomplish the task of CTS for 30 versions within 24 hours. Therefore, we discarded the results of this CTS technique in our experiment. We will compare the effectiveness of CTS techniques using block, block+filtering and statement+filtering.

Figure 6 gave a comparison of test suite reduction $TR$ of CTS techniques using block, block+filtering and statement+filtering. The experiment results showed that almost half of $TR$s were about the same, and half of $TR$s of filtering were better (i.e. lower) than $TR$s of no filtering. Overall, CTS techniques with filtering achieved better effectiveness than CTS techniques without filtering modestly for test suite reduction. CTS techniques using statement coverage profiles achieved
similar effectiveness as CTS techniques using block coverage profiles. A reasonable explanation of the results is that the statements in the same block have similar ability to distinguish the behaviors of test cases.

Figure 7 gave a comparison of F-measure of CTS techniques using block, block+filtering and statement+filtering. The experiment results showed that most of F-measures of filtering were better (i.e. higher) than F-measures of no filtering. The F-measures of some versions achieved much better improvement, such as version 5, 7, 8, 9, 10, 16, 17 and 29. CTS techniques using statement coverage profiles achieved similar effectiveness as CTS techniques using block coverage profiles, because the statements in a same block have similar ability to distinguish the behaviors of test cases. Overall, The filtering techniques could greatly improve the effectiveness of CTS techniques with regard to F-measure, which stands for fault detection capability.

5.3. Threats to Validity

5.3.1. Internal Validity

Threats to internal validity are uncontrolled factors that are also responsible for our results. The threat is that there are defects in our implementation of CTS techniques. To reduce this threat, we reviewed all the implementation code before conducting our experiment. We also used some popular tools, such as gcov [11] and Weka [27], in our implementation to reduce the threats.
5.3.2. External Validity

Threats to external validity are mainly the representativeness of our subject program space. The space is widely used in many existing studies. And 30 versions with different faults and a large number of test cases were used in our experiment to reduce this threat. We would conduct a family experiment with more subject programs, such that the results have more generality in the future.

5.3.3. Construct Validity

Threats to construct validity are mainly concerned with the evaluation measures in our experiment. In order to reduce the threat, we used common measures in related research fields. Dimensionality reduction is widely studied in cluster analysis and other machine learning fields. F-measure is commonly used in information retrieval fields. Test suite reduction is a common term in software testing.

6. Related Work

Test selection techniques could be classified into three categories: regression test selection, test suite minimization, and test case prioritization [28]. This paper studied a novel approach for regression test selection, which is performed between two different versions of software in order to provide confidence that changes of software do not interfere with the existing features. Many regression test selection techniques have been proposed. Fischer [10] used the systems of linear equations to select test suites that yield segment coverage of modified code. Yau and Kishimoto [25] used input partitions and data-driven symbolic execution techniques to achieve test case selection. Leung and White [16] presented a technique that placed a firewall around modified modules and selected unit tests for modified modules that lay within the firewall and integration tests for groups of interaction modules that lay within the firewall. Chen [5] developed a technique to select test cases that traverse the modified code entities such as functions, variables, etc. Rothermel et al. [19] proposed a safe test selection technique. A selection method is safe if it selects all the fault-revealing test cases. All the above regression test selection techniques are quite different from the cluster test selection techniques in this paper.

Cluster test selection (CTS) is a novel approach for regression test selection in the past years. CTS was firstly used in the observation-based testing [6, 7]. Dickinson [6, 7, 15] et al. used cluster analysis on execution profiles induced by the original test cases. And then one or more test cases were sampled from each cluster and manually checked whether they are fault-revealing test cases. Yan et al. [24] proposed a new test case sampling strategy by leveraging test case spectra information to select test cases that are most likely to be failed. Zhang et al. [30] introduce cluster analysis in regression testing to improve the precision of test selection techniques. All the above CTS techniques used function call profiles and we used statement/block execution profiles. Furthermore, they did not discuss how to reduce the dimensions
of test cases such that it can be used for larger software. Recently, Chen et al. introduce semi-supervised learning to reduce the dimensions of test cases and improve the effectiveness of test selection [4]. However, semi-supervised learning techniques require human participation and the slice filtering could be implemented automatically. More related efforts on regression test selection could be found in the recent survey paper [28].

An other topic related to our approach is the application of program slicing in regression testing. Harrold et al. used static slicing to detect definition-use associations that were affected by a program change in regression testing [12]. Agrawal [1] discussed the application of execution slice in regression test selection. The set of statements executed under a test case is referred to as the execution slice of the program with respect to a test case. During the off-line processing, the execution slices of the program with respect to all test cases in the regression test suite are collected. After the program is modified, only those test cases whose execution slices contain a modified statement are selected to rerun. Jeffrey and Gupta used the relevant slices of the outputs to prioritize test cases in regression testing [13]. Three different heuristics based on relevant slicing were implemented and a detail experiment was conducted. All the above approaches directly used program slicing in regression testing. And the program slicing was indirectly used here to filter out some irrelevant feature elements, such that the efficiency and effectiveness could be improved.

7. Conclusion

In this paper, we proposed a novel method with program slicing to improve CTS techniques. The program slicing technique was used to filter out some irrelevant feature elements and further some irrelevant test cases. An experiment with a widely used subject program \textit{space} was designed and implemented. The experiment results show that:

(1) The filtering techniques could significantly reduce the cost of CTS for both dimensionality reduction and matrix reduction.
(2) The filtering techniques could modestly improve the effectiveness of CTS for test suite reduction.
(3) The filtering techniques could greatly improve the effectiveness of CTS for fault detection capability.

Based on the above observations, CTS techniques with filtering have potential scalability to deal with large software. We will perform a more comprehensive empirical study to compare the efficiency and effectiveness with other test selection techniques in the future.
Acknowledgement

The work described in this article was partially supported by the National Natural Science Foundation of China (90818027, 60803007, 61003024, 91018005), the Major State Basic Research Development Program of China (973 Program: 2009CB320703), and the Fundamental Research Funds for the Central Universities.

References