Sequential Pattern Mining Based Test Case Regeneration

Wei He, Ruilian Zhao*
Department of Computer Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China
Email: wei.he.stkm@gmail.com, rlzhao@mail.buct.edu.cn

***Abstract*— Automated test generation for object-oriented programs is an essential and yet a difficult task. Many automated test generation approaches produce test cases entirely from the program under test, without considering useful information from already created test cases. This paper presents an approach to regenerate test cases via exploiting frequently-used method call sequences from test repository. Particularly, for an object-oriented program under test, a sequential pattern mining strategy is employed to obtain frequent subsequences of method invocations as sequential patterns from corresponding test repository, and then a GA-based test case regeneration strategy is used to produce new test cases on the basis of the sequential patterns. A prototype called SPM-RGN is developed and is applied to generate test cases for actual Java programs. Empirical results show that SPM-RGN can achieve 47.5%, 11.2% and 4.5% higher branch coverage than three existing automated test generators. Besides, SPM-RGN produces 85.1%, 28.1% and 27.4% shorter test cases than those test generators. Therefore, the test cases generated by SPM-RGN are more effective and easier to understand.**

***Index Terms*— test case regeneration, object-oriented software, sequential pattern, Genetic Algorithms, test repository**

I. Introduction

Software testing is nowadays a dominant way to assure the quality of software products, but finding suitable sets of test cases is a challenging task [1]. During the past decades, the researches on automated test generation have received an enormous amount of attention due to its benefit in quality improvement and cost saving [2]–[6].

However, many existing approaches generate test cases for the programs under test, implicitly assuming no previous test cases are available. If this assumption does not hold, these approaches will miss the opportunity of getting useful information from already created test cases, especially relevant manual test cases. A recent study introduces a test data regeneration strategy to generate new test data from existing test data for procedural programs [7]. Their test results indicate that the knowledge of existing test data can help to improve the quality of new generated test data.

Moreover, most programs developed currently are object-oriented (OO). As regards OO programs, each test case consists of a method call sequence instead of primitive values which form test inputs for procedural programs. Hence, automated test generation for OO programs is much more complex than for procedural programs [8].

Sequential pattern mining is a widely used data mining technique to discover subsequences that appears no less than a specified frequency in a sequence database [9]. For OO programs, each test case is composed of a method call sequence (i.e. a sequence of method invocations), thus a test repository for an OO program can be considered as a database with numerous method call sequences. Then, sequential pattern mining can be used to find frequently-used subsequences of method invocations for regenerating new test cases.

In this paper, we propose an approach to regenerate test cases for OO programs by integrating sequential pattern mining with meta-heuristic test generation. Rather than generating test cases entirely from the programs under test, our approach employs sequential pattern mining to exploit frequently-used subsequences of method invocations as sequential patterns from test repository, and uses these sequential patterns in meta-heuristic test generation to regenerate new test cases. More specifically, providing that a test repository is available, BI-Directional Extension (BIDE) strategy [10], an efficient sequential pattern mining strategy, is adapted to find sequential patterns from the test repository at first. Then, the sequential patterns are used in GA-based test generation in order to produce new test cases for covering test targets in OO programs.

We develop a prototype, called SPM-RGN (Sequential Pattern Mining based test case ReGeNerator), to implement the approach. This prototype can successfully regenerate new test cases from existing test cases for Java programs. What's more, if the test repository is created manually, the resulting test cases produced by SPM-RGN are easier to understand than those generated thoroughly from the program under test.

The main contributions of this work are as follows:

A test case regeneration approach based on sequential pattern mining is proposed to produce test cases from existing test repositories for OO programs. This approach combines 1) a sequential pattern mining strategy to obtain frequent subsequences of method invocations as sequential patterns from available test repositories, and 2) a GA-based test case regeneration strategy to produce new test cases for OO programs on the basis of the sequential patterns.

* Corresponding author.

Empirical evaluations are conducted by comparing SPM-RGN with two popular test generators, Randoop and EvoSuite, as well as our early developed RND-RGN on 4 open source Java projects (with totally more than 40K LOC), in which Randoop is a feedback-directed random test generator [11], EvoSuite is a GA-based test generator with random initial population [12] and RND-RGN is a GA-based test case regenerator with randomly taking existing test cases as initial population. The empirical results show that our SPM-RGN is effective in achieving high branch coverage and producing short test cases for all these projects. More specifically, SPM-RGN can achieve up to 47.5%, 11.2% and 4.5% higher branch coverage than Randoop, EvoSuite and RND-RGN, respectively. Besides, the average length of test cases generated by SPM-RGN is 85.1%, 28.1% and 27.4% shorter than by Randoop, EvoSuite and RND-RGN, respectively. Hence, it can be concluded that the test cases generated by SPM-RGN are more effective and easier to understand.

The rest of this paper is organized as follows. Section II provides the background underlying our work. Section III elaborates our test case regeneration approach based on sequential pattern mining. Empirical evaluations are described in Section IV. Section V reviews related work. Finally, Section VI concludes the paper and lists future research directions.

II. Background

This section describes the automated test generation for OO programs and the sequential pattern mining technique, which our approach is based upon.

A. Automated test generation for OO programs

Automated test generation for OO programs has been a sustained topic of interest over the last decade because it is helpful to improve the quality and to reduce the cost in the testing phase [8]. It aims at automatically creating a suite of test cases that can achieve a specified coverage criterion, such as branch coverage for the program under test, each test case consisting of a method call sequence that covers a target branch. In structural testing of OO programs, test cases are often produced randomly [11], [13], [14] or heuristically [12], [15], [16].

Random test generation is simple and scales well [13]. However, for OO programs, random test generators such as Randoop [11] usually produce long and complex test cases [14]. Besides, random test generators are hard to produce particular test cases that can cover the code fragments guarded by nested branch conditions [14].

Meta-heuristic test generation is believed as a promising approach to generate high quality test cases [15], [16]. It formulates the task of automated test generation as a search problem and solves it via meta-heuristic search techniques. As a subcategory, Evolutionary Testing (ET) has attracted much attention, which applies evolutionary algorithms such as Genetic Algorithms (GA) [17] to search for required test cases. Successful ET tools like EvoSuite [12] have been developed to generate test cases for OO programs.

All these approaches have made notable progress in automated test generation for OO programs. However, they have not considered valuable information from test repository if it is available.

B. Sequential pattern mining

Sequential pattern mining is a data mining technique that is introduced to find frequently occurring ordered subsequences in a sequence database [9]. The identified subsequences can often highlight useful information underlying the database and can be used for further application.

A sequence database contains a number of *records*, where each record is made of a sequence of ordered *items*. If a subsequence appears in the sequence database with frequency no less than a *minimum support threshold* θ, it is referred to a *frequent sequential pattern* (or a *sequential pattern* for short). The number of items in a sequential pattern is called its *length*, and a sequential pattern with length l is named as a *frequent l-sequence*.

To obtain frequent sequential patterns in a sequence database, some sequential pattern mining strategies, such as BI-Directional Extension (BIDE for short) [10], are developed. Specifically, BIDE finds all frequent 1-sequences from the sequence database at first. Next, for each frequent 1-sequence, BIDE collects the items appearing just before the 1-sequence, and attempts to extend the frequent 1-sequence to a 2-sequence by inserting such an item before the 1-sequence, namely backward extension. If the resulting sequence meets the frequency θ, i.e. occurring with frequency no less than θ in the database, then a frequent 2-sequence is obtained. Similarly, 2-sequence may be extended to a 3-sequence, and so on. The sequence is extended backwards until the resulting sequence appears with frequency less than θ in the database. After that, BIDE gathers the items occurring just after the sequence, and then extends the sequence by appending such an item after the sequence, namely forward extension. BIDE extends the sequence forwards till the resulting sequence does not satisfy θ. As a result, a frequent sequential pattern of the sequence database is obtained. Due to each 1-sequence may have a lot of adjacent items in a sequence database, a set of frequent sequential patterns is produced from the database.

With over a decade of substantial and fruitful research, it is believed that sequential pattern mining has the potential to significantly help software testing [18]. In this paper, we consider a test repository for an OO program as a sequence database. Thereupon, sequential pattern mining can be used to obtain frequent subsequences from the test repository, and then meta-heuristic test generation is applied to regenerate new test cases on the basis of these frequent subsequences.

III. Test Case Regeneration Based on Sequential Pattern Mining

As we know, a test case is made up of a method call sequence mcs for OO programs. Accordingly, given an OO program under test p, a coverage criterion c, and a repository of test cases T, each t of T consisting of an mcs, then the task of test case regeneration based on sequential pattern mining can be formulated as:

1) Find a set of frequent sequential patterns S from T with a user-defined minimum support threshold θ, where each s of S is a mcs subsequence (i.e. $s \subseteq mcs$) and occurs no fewer than $\theta|T|$ times ($|T|$ is the total number of test cases in T);
2) Produce a set of new test cases T based on S by using GA-based test generation to achieve the coverage criterion c for program p.

In the following section, we elaborate how to perform the test case regeneration with the help of sequential pattern mining strategy.

A. Mining sequential patterns from test repository

For a given test repository T, a BIDE mining strategy can be employed to find frequent sequential patterns S in a "pattern growth" manner. For this purpose, we define *prefix sequence* and *suffix sequence* as follows.

Definition 1: Prefix Sequence. Given a method call sequence $=<m_1\ m_2\ \cdots\ m_i\ \cdots\ m_n>$, then subsequence $\beta=<m_1\ m_2\ \cdots\ m_{i\ 1}>$ is called its prefix sequence with respect to method m_i.

Definition 2: Suffix Sequence. Given a method call sequence $=<m_1\ m_2\ \cdots\ m_i\ \cdots\ m_n>$, then subsequence $\gamma=<m_{i+1}\ m_{i+2}\ \cdots\ m_n>$ is called its suffix sequence with respect to method m_i.

According to a test repository T, all frequent 1-sequences S_1 (subsequences have a length l=1 and occur with frequency no less than a minimum support threshold θ) are identified at first by scanning every mcs in T. For a 1-sequence s_1 of S_1, there are more than one prefix sequences and suffix sequences with respect to s_1 since s_1 occurs at least $\theta|T|$ times in T. Here, a collection of its prefix sequences and that of suffix sequences are denoted by P_{s_1} and S_{s_1}, respectively.

For each 1-sequence s_1, we attempt to extend it backwards by inserting the methods of its prefix sequence one by one before s_1 to get longer frequent sequences. This is to say, for a prefix sequence of s_1, for example $p_{s_1}=<m_1\ \cdots\ m_{j\ 1}\ m_j>\in P_{s_1}$, we insert the last method m_j of p_{s_1} just before s_1. If the resulting sequence $<m_j\ s_1>$ appears no less than $\theta|T|$ times in T, a frequent 2-sequence $s_2=<m_j\ s_1>$ is produced and s_2 is tried to extend backwards to a 3-sequence by adding $m_{j\ 1}$ before s_2. The sequence is lengthened in this way until the resulting sequence appears fewer than $\theta|T|$ times in T. As a result, we obtain a backward frequent sequence s with respect to s_1. After that, we try to extend s forwards to obtain longer frequent sequences. More specifically, for a suffix sequence $s_{s_1}=<m_k\ m_{k+1}\ \cdots\ m_n>\in S_{s_1}$, we append the first method m_k of s_{s_1} just after s . If the resulting sequence $<s\ \ m_k>$ appears no fewer than $\theta|T|$ times in T, then $<s\ \ m_k>$ is tried to extend forwards by adding m_{k+1} after it. The sequence is extended likewise until the resulting sequence appears fewer than $\theta|T|$ times in T. Thus, an entire frequent sequence s with respect to s_1 is gotten. As mentioned above, there are many prefix sequences and suffix sequences with respect to each s_1, therefore a set of entire frequent sequences, named as frequent sequential patterns S, can be produced from T. The pseudo-code for obtaining S from T is given in Algorithm 1.

Note that, the minimum support threshold θ is a key parameter in mining sequential patterns from a test repository T. To choose a proper θ with respect to T, we define *Averaged Method Invocation Frequency* (*AMIF* for short) as follows.

Definition 3: Averaged Method Invocation Frequency (AMIF). Given a test repository T containing $|T|$ test cases that invoke n methods in total and each method m_i is invoked $|m_i|$ times in T, then the averaged method invocation frequency with respect to the test repository is defined as

$$AMIF = (\sum_{i=1}^{n} \frac{|m_i|}{|T|})\ \ n = \frac{\sum_{i=1}^{n} |m_i|}{n|T|} \tag{1}$$

AMIF reflects how frequently the methods are invoked in a test repository. If a test repository has a relatively small *AMIF*, its methods are rarely invoked. Consequently, a relatively small amount of sequential patterns is produced. In contrast, as for a test repository with relatively large *AMIF*, its methods are frequently invoked. Thus, it is likely to yield plenty of sequential patterns.

In order to obtain an appropriate amount of sequential patterns within reasonable mining cost, we assign different values to θ in accord with *AMIF*. More specifically, if *AMIF* is small for a test repository, then we choose a relatively low θ value; conversely, a relatively high θ value is used.

B. Regenerating test cases based on sequential patterns

For an OO program p, after sequential patterns S are obtained from its corresponding test repository using the above technique, GA-based ET is employed to regenerate test cases according to S. In other words, GA-based ET takes the program p and the sequential patterns S as inputs, and seeks for a group of mcs as test cases with respect to a selected coverage criterion c like branch coverage.

To be more specific, p is analyzed statically to find all methods M and branches B at first, and some statements are instrumented to record dynamic execution traces of p. Then, taking a branch b of B as a target, a population P of candidate executable mcs is initialized, in which each individual i is constructed by a sequential pattern s

Algorithm 1: Get sequential patterns from a test repository

```
Input: A test repository of test cases $T$
Input: A minimum support threshold $\theta$
Output: A set of sequential patterns $S$
1  $S = \emptyset$;
2  $S_1$ = all frequent 1-sequences of $T$;
3  foreach 1-sequence $s_1 \in S_1$ do
4      $P_{s_1}$ = all prefix sequences with respect to $s_1$;
5      $S_{s_1}$ = all suffix sequences with respect to $s_1$;
6      foreach prefix sequence $p_{s_1} \in P_{s_1}$ do
7          $s = s_1$;
8          $m_l$ = the last method of $p_{s_1}$;
9          while $<m_l\ s>$ appears no less than $\theta|T|$ times in $T$ do
10             $s = <m_l\ s>$;
11             $p_{s_1} = p_{s_1} - m_l$;
12             $m_l$ = the last method of $p_{s_1}$;
13         foreach suffix sequence $s_{s_1} \in S_{s_1}$ do
14             $s = s$ ;
15             $m_f$ = the first method of $s_{s_1}$;
16             while $<s\ m_f>$ appears no less than $\theta|T|$ times in $T$ do
17                 $s = <s\ m_f>$;
18                 $s_{s_1} = s_{s_1} - m_f$;
19                 $m_f$ = the first method of $s_{s_1}$;
20             $S = S \cup \{s\}$;
21 return $S$;
```

Algorithm 2: Provide argument values for an individual

```
Input: An individual $i$
1  $ARG$ = the arguments of $i$;
2  while $ARG \neq \emptyset$ do
3      foreach argument $arg \in ARG$ do
4          if $arg$ refers to a primitive value then
5              $arg$ = a random number;
6          else
               // arg refers to an object
7              $m^\star$ = a method of $M$ such that $T_{m^\star} \sqsubseteq T_{arg}$;
8              $arg$ = the return object of $m^\star$;
9      update $ARG$;
```

of S. To make i executable, some arguments of s may need to be assigned values. If an argument arg refers to a primitive value such as an integer, a random number is assigned; otherwise, i.e. arg corresponds to an object, M is scanned to find a method $m^\star$ whose return type $T_{m^\star}$ is either the same as or a subtype of the declared type T_{arg} of arg, namely $T_{m^\star} \sqsubseteq T_{arg}$. Then the return object of $m^\star$ is assigned to arg. Similarly, the argument values of $m^\star$ are also supplied if needed, and so forth until no argument value is required any more. So far, an individual i is generated. The pseudo-code for providing argument values for an individual is depicted in Algorithm 2.

After P is initialized, each individual i of P is dynamically executed. If i reaches branch b, a new test case is produced and ET continues for covering another target branch b . Otherwise, i is evaluated using a fitness function to measure the distance between the execution trace of i and the target branch b. If b is not covered by any individual of P, two elite individuals, represented by i_1 and i_2, are selected from P; and genetic operations, namely crossover and mutation, are performed on i_1 and i_2 to produce offspring individuals. In these operations, one-point crossover is used with probability p_c to exchange a subsequence of i_1 with another subsequence of i_2. In addition, change mutation is employed with probability p_m to turn a subsequence of i_1 and i_2 into a new subsequence separately. More concretely, two sequential patterns $\tilde{s}_1$ and $\tilde{s}_2$ are randomly selected from S to substitute a random subsequence of i_1 and i_2, respectively. After performing change mutation, i_1 and i_2 may require new argument values. Subsequently, the argument values are produced according to Algorithm 2, and executable offspring individuals i_1 and i_2 are obtained.

After that, a new population P is generated by replacing two low-fitness individuals of P with i_1 and i_2 and evaluated by the fitness function. The process of generating new population and evaluating the population is repeated until a pre-assigned termination condition is met (e.g. all branches are covered or the time budget is consumed). Thus, a group of new test cases T is produced. As a whole, the pseudo-code for regenerating test cases based on sequential patterns is described in Algorithm 3.

IV. Empirical Evaluations

To evaluate the approach presented in this paper, we implemented it with a prototype called SPM-RGN (Sequential Pattern Mining based test case ReGeNerator), and conducted a set of experiments on Java programs.

First, we examined the performance of our approach on mining sequential patterns from different test repositories. In particular, we studied (1) the amount of frequent sequential patterns obtained from distinct test repositories, and (2) the corresponding cost of mining the test repositories, including the time consumption and the memory usage.

After that, we compared our SPM-RGN with Randoop, EvoSuite and RND-RGN, where Randoop is an advanced random test generator [11], EvoSuite is a GA-based test generator with random initial population [12], and RND-RGN is our early developed GA-based test case regenerator which randomly takes existing test cases as initial population. More specifically, these test generators are compared in two aspects: (1) the branch coverage achieved by these test generators, and (2) the understandability of the test cases produced by these test generators.

Algorithm 3: Regenerate new test cases based on sequential patterns

```
Input: An OO program under test $p$
Input: A set of sequential patterns $S$
Output: A set of new test cases $T$
1  $B$ = all branches of $p$;
2  $M$ = all methods of $p$;
3  instrument $p$ to trace executions;
4  $T = \emptyset$;
5  foreach branch $b \in B$ do
6      population $P = \emptyset$;
7      while $|P| <$ population size pop_size do
8          individual $i$ = a random sequential pattern $s \in S$;
9          call ProvideArgumentValues($i$);
10         $P = P \cup \{i\}$;
11     repeat
12         foreach individual $i \in P$ do
13             execute $i$;
14             if $i$ covers $b$ then
15                 $T = T \cup \{i\}$;
16                 break;
17             evaluate fitness of $i$;
18         select two elite individuals $i_1$ and $i_2$ from $P$;
19         if $rand[0\ 1) <$ crossover probability $p_c$ then
20             crossover $i_1$ and $i_2$;
21         if $rand[0\ 1) <$ mutation probability $p_m$ then
22             $\tilde{s}_1$ = a random sequential pattern of $S$;
23             $i_1$ = change a random subsequence of $i_1$ with $\tilde{s}_1$;
24             $\tilde{s}_2$ = a random sequential pattern of $S$;
25             $i_2$ = change a random subsequence of $i_2$ with $\tilde{s}_2$;
26         call ProvideArgumentValues($i_1$);
27         call ProvideArgumentValues($i_2$);
28         replace two low-fitness individuals of $P$ with $i_1$ and $i_2$;
29     until termination condition is met;
30 return $T$ ;
```

All empirical evaluations were conducted on an HP ProLiant Server with Intel Xeon 2.40GHz*16 processors, 4GB*6 RAMs, 64-bit CentOS Linux 6.0 and JDK 1.6.0.

A. Subject programs

In the empirical evaluations, 4 open source Java projects are used as experimental subjects: Commons Collections (CC) and Commons Primitives (CP) are from the Apache Commons Proper repository, a repository of Java libraries; while JTopas (JT) and NanoXML (NX) come from the Software-artifact Infrastructure Repository, a repository for rigorous control experiments [19]. Among these projects, the largest one CC consists of more than 20K LOC. In total, the 4 projects include nearly 45K LOC. We use the manual test cases released along with each project as its test repository.

Table I summarizes statistics of the experimental subjects in terms of the number of classes (*#Classes*), the number of all methods (*#Methods*), the number of all branches (*#Branches*), non-commented lines of code (*LOC*), the number of test cases in corresponding test repository (*#Tests*) and the averaged method invocation frequency (*AMIF*) of each test repository .

B. Parameter settings

As mentioned in Section III, our sequential pattern mining based test case regeneration approach is composed of two main stages: the sequential pattern mining stage and the test case regeneration stage. For these two stages, there are several parameters to be configured.

With respect to the sequential pattern mining stage, minimum support threshold θ is an important parameter. For each test repository, we assign different values to θ according to the *AMIF* of the test repository, in order to obtain an appropriate amount of frequent sequential patterns within reasonable mining cost. In our experiments, the *AMIF* of the test repository for CC is nearly 0.5%. Accordingly, we tried out low minimum support thresholds varying from 1% to 5% for this test repository. Similarly, the *AMIF* of the test repository for CP is merely 0.4%, and we also used minimum support thresholds varying from 1% to 5%. Different from the former two test repositories, as regards the test repository for JT, whose *AMIF* is up to 7.6%, we used relatively high minimum support thresholds from 10% to 50%. With respect to the test repository for NX, whose AMIF is 2.4%, we tried out more comprehensive minimum support thresholds ranging from 1% to 50%.

As regards the test case regeneration stage, EvoSuite, RND-RGN and our SPM-RGN were all configured according to the recommended settings in the literature [20]. In particular, the budget for test generation of each class was set to be no more than 600 seconds. The population size was 100. Genetic operations were set to: rank selection with 1.7 biases; one-point crossover with a probability of 0.75; change mutation probability of 0.3. Randoop was configured with identical settings to EvoSuite, RND-RGN and SPM-RGN when needed.

EvoSuite generated initial population entirely at random. RND-RGN took test cases randomly from corresponding test repository as initial population. Our SPM-RGN constructed initial population by using the sequential patterns.

In addition, there were randomness to some extent in Randoop, EvoSuite, RND-RGN and SPM-RGN. Thus, each experiment was repeated 30 times, and statistical methods were used to analyze the results.

C. Experiments on mining sequential patterns from test repository

Using the sequential pattern mining technique proposed in Section III-A, sequential patterns can be successfully

TABLE I.
STATISTICS ON THE EXPERIMENTAL SUBJECT PROGRAMS

Subjects	#Classes	#Methods	#Branches	LOC	#Tests	AMIF
CC	382	3182	6276	26323	1151	0.5%
CP	231	1756	1446	9836	732	0.4%
JT	63	719	1376	5361	52	7.6%
NX	24	317	690	3279	76	2.4%
Total	**700**	**5874**	**9788**	**44799**	**2011**	**–**

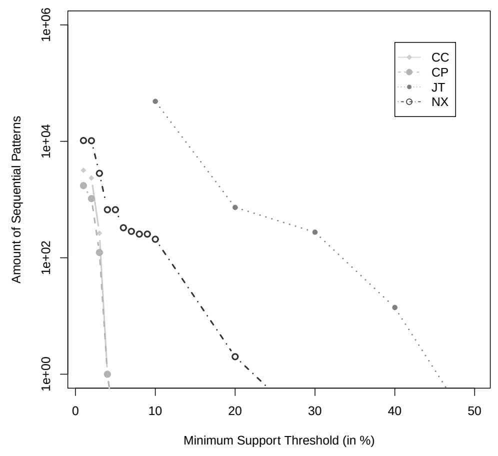

Figure 1. Amount of sequential patterns obtained from the test repositories

obtained from corresponding test repository. This section presents the amount of resulting sequential patterns and the cost of whole sequential pattern mining stage.

1) Amount of sequential patterns: Figure 1 depicts the amount of sequential patterns obtained from test repositories using different minimum support thresholds. From Figure 1, we can see that thousands of sequential patterns can be discovered from these test repositories. Generally speaking, the amount of sequential patterns obtained from each test repository decreases logarithmically with respect to the minimum support thresholds. In particular, regarding the test repositories for CC and CP, sequential patterns can be discovered with relatively low minimum support thresholds. For example, at support 1%, 3181 sequential patterns can be obtained from the test repository for CC. In contrast, large amounts of sequential patterns can even be obtained from the test repository for JT using relatively high minimum support thresholds. For example, at support 10%, 48603 sequential patterns can be obtained from the test repository for JT. Concerning the test repository for NX, whose *AMIF* is moderate, sequential patterns can be obtained with minimum support thresholds varying from 1% to 20%.

2) Mining cost: Figure 2 demonstrates the time and memory cost in mining sequential patterns from test repositories with different minimum support thresholds. We can see that, both the time consumption and memory usage also decrease logarithmically with respect to the minimum support thresholds. Besides, under the same minimum support threshold, mining sequential patterns

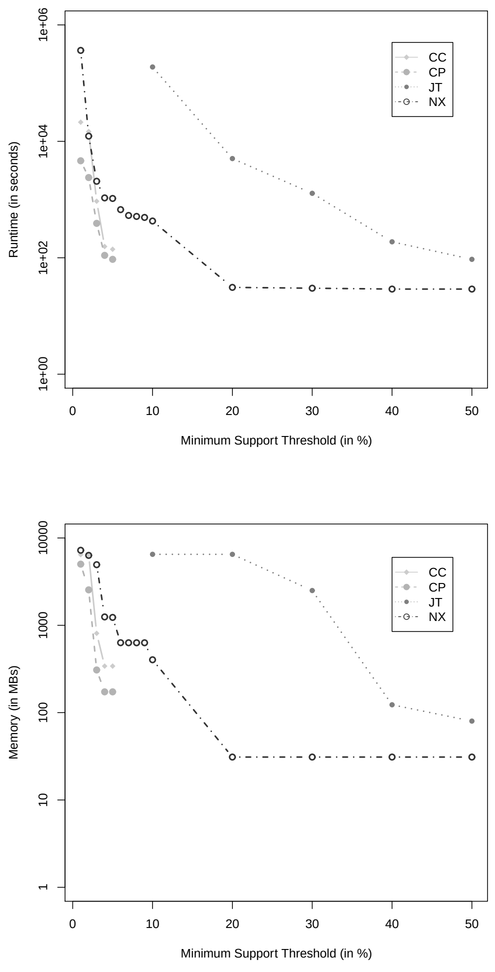

Figure 2. Cost in mining sequential patterns from the test repositories

from a test repository with small *AMIF* always runs faster and consumes less memory than from a test repository with large *AMIF*. For example, mining sequential patterns from the test repository for CC can be nearly an order of magnitude faster than for NX, while it only uses nearly an order of magnitude less memory. Similarly, mining sequential patterns from the test repository for NX can be over an order of magnitude faster than for JT, while it only uses over an order of magnitude less memory.

3) Summary: Generally speaking, mining a certain test repository with a smaller minimum support threshold value leads to more sequential patterns but consumes more time and memory. Thus, in practice, a proper minimum support threshold should be chosen according to the *AMIF*

TABLE II.
BRANCH COVERAGE (IN %) ACHIEVED BY RANDOOP, EVOSUITE, RND-RGN AND SPM-RGN

Subjects	Randoop	EvoSuite	RND-RGN	SPM-RGN
CC	20.9	83.1	85.3	**94.1**
CP	62.8	70.1	87.3	**90.7**
JT	49.0	73.7	77.1	**80.2**
NX	15.8	66.8	70.6	**73.2**
Avg.	37.1	73.4	80.1	84.6

of the corresponding test repository, in order to make a good trade-off between the amount of resulting sequential patterns and the mining cost.

D. Experiments on regenerating test cases based on sequential patterns

Using the approach proposed in Section III-B, new test cases can be regenerated based on obtained sequential patterns. This section compares the effectiveness of our SPM-RGN with Randoop, EvoSuite, and RND-RGN by applying them to generate test cases for 4 open source Java projects, respectively (over all classes therein, each test generator is repeated 30 times).

1) Branch coverage: For each subject program, the branch coverage achieved by each test generator (averaged over the 30 runs) is summarized in Table II. The maximum branch coverage achieved for each subject program is marked in bold. For each test generator, the average branch coverage it achieved is given at the bottom.

It can be seen from Table II, that SPM-RGN can successfully regenerate new test cases with highest branch coverage for all subject programs. The average branch coverage achieved by SPM-RGN ranges from more than 70% to nearly 95%. The average increment of SPM-RGN over Randoop, EvoSuite and RND-RGN is up to 47.5%, 11.2% and 4.5%, respectively. Moreover, the improvement of SPM-RGN over Randoop, EvoSuite and RND-RGN is statistically significant at the 95% confidence level according to one-tailed Mann-Whitney U test.

Figure 3 presents the distribution of branch coverage achieved for the subject programs by each test generator. We can see that the branch coverage achieved by Randoop diverges a lot in different runs; whereas the branch coverage achieved by EvoSuite, RND-RGN and SPM-RGN varies comparatively not much in different runs.

2) Test case understandability: We have also measured the test case understandability in terms of average length of the test cases produced by each test generator (each test generator is repeated 30 times). The result is summarized in Table III, where bold numbers denote the shortest average length of the test cases produced for each subject program. For each test generator, the average length of the test cases it produced is presented at the bottom.

From Table III, it can be found that SPM-RGN produces shortest test cases among the test generators: on average about 20.7 lines of code, with 85.1%, 28.1% and 27.4% shorter than Randoop, EvoSuite and RND-RGN, respectively. The reduction is also statistically significant

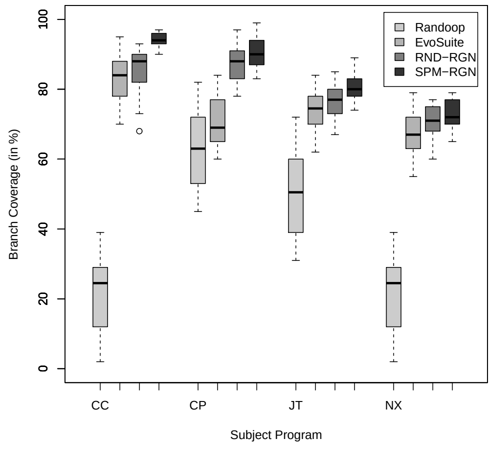

Figure 3. Distribution of branch coverage (in %) achieved by Randoop, EvoSuite, RND-RGN and SPM-RGN

TABLE III.
AVERAGE LENGTH OF THE TEST CASES PRODUCED BY RANDOOP, EVOSUITE, RND-RGN AND SPM-RGN

Subjects	Randoop	EvoSuite	RND-RGN	SPM-RGN
CC	110.8	19.6	19.1	**18.9**
CP	98.2	16.3	14.6	**13.7**
JT	111.7	32.6	48.5	**25.1**
NX	235.9	46.5	31.8	**25.2**
Avg.	139.1	28.8	28.5	20.7

at the 95% confidence level according to one-tailed Mann-Whitney U test.

Figure 4 illustrates the distribution of average length of the test cases produced for the subject programs by each test generator. We can see that, in different runs, the average length of the test cases achieved by Randoop also diverges most; whereas our SPM-RGN diverges least.

3) Summary: The above results suggest that our SPM-RGN can successfully regenerate new test cases for OO programs based on sequential patterns. In general, it can achieve good branch coverage. Besides, the test cases generated by SPM-RGN are comparatively short, thus are likely easier to understand by human.

V. RELATED WORK

Existing approaches to structural test generation for OO programs can be broadly classified into two major categories: implementation-based and usage-based approaches [21].

A. Implementation-based approaches

As discussed in Section II-A, structural test generation for OO programs is commonly based on the code implementation of programs under test all the way, either randomly or heuristically.

Among existing tools, Randoop [11] is a random test generator that incrementally generates method call sequences for Java programs by randomly invoking

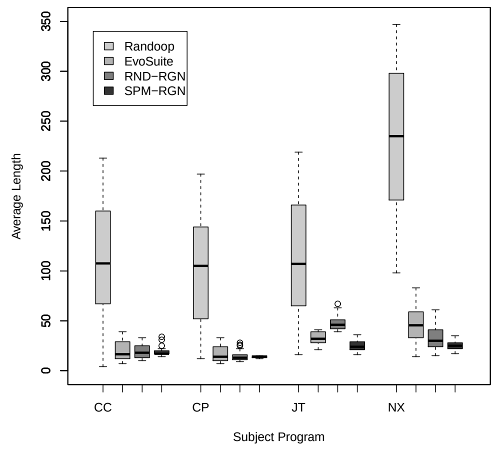

Figure 4. Distribution of average length of the test cases produced by Randoop, EvoSuite, RND-RGN and SPM-RGN

methods of the program under test. However, due to the large amount of all possible methods, Randoop is hard to generate test cases for deep target branches. In contrast, our SPM-RGN exploits common method usage from existing test cases. Besides, SPM-RGN employs GA to direct the search of new test cases towards target branches. Thus, within limited test generation time, SPM-RGN is more likely to generate test cases that cover deep branches of the program under test. This paper compares the performance of our SPM-RGN with Randoop in the empirical evaluations.

Besides, EvoSuite [12] is a GA-based test generator that heuristically generates test cases for Java programs. In EvoSuite, individuals are directly represented as method call sequences with argument objects. EvoSuite generates initial population at random and evolves the population with genetic operations. If there is a conflict among method invocations after a genetic operation, adjustment is required to handle the conflict. On the contrary, our SPM-RGN generates test cases based on sequential patterns of existing test cases, and thus can reduce the time consumption in adjustment because in general there is few conflict in existing test cases. The performance of our SPM-RGN and EvoSuite is also compared in the empirical evaluations.

B. Usage-based approaches

In recent years, some approaches turn to generate test cases for OO programs by exploiting usage information from code bases beyond the program under test [18]. Generally speaking, our approach belongs to this category. In particular, our approach generates new test cases on the basis of method-usage information, i.e. sequential patterns, obtained from a test repository.

Among the usage-based approaches, Yoo's work on test data regeneration [7] is perhaps the most related one to ours. In [7], a test data regeneration approach is proposed for producing test data for procedural programs. It takes existing test data of the program under test as starting points and employs a hill-climbing algorithm to seek for new test data. Similarly, our approach attempts to regenerate test cases based on existing test cases. However, there are three main differences between Yoo's approach and ours. First of all, Yoo's approach aims to regenerate primitive test data as test inputs for procedural programs, whereas our approach regenerates method call sequences as test cases for OO programs. Secondly, Yoo's approach uses hill climbing to search new tests, whereas our approach employs GA. Last but not least, Yoo's approach uses existing test data directly as initial solutions in test data regeneration, whereas our approach mines sequential patterns from test repositories and regenerates test cases based on the sequential patterns. According to the idea of [7], we implemented a test generator called RND-RGN that randomly takes available test cases as initial population to search new test cases for OO programs, and compared the performance of our SPM-RGN with RND-RGN in the empirical evaluations.

Moreover, our approach is also related to several other usage-based approaches that generate test cases for OO programs [21]–[24]. Among those approaches, DyGen [21], MSeqGen [22] and OCAT [23] capture object instances and serialize them into a file for further usage; whereas [24] infers a graphic model to represent the common usage of objects. Our approach is different from these approaches in two main aspects. Firstly, these approaches obtain object-usage information from massive code bases, whereas our approach mines test repositories particularly. Secondly, our approach uses meta-heuristic test generation technique to produce new test cases, which is usually more effective than random test generation or dynamic symbolic execution adopted in those usage-based approaches.

VI. Conclusion and Future Research

In software testing, it is a challenging task to automatically generate desirable test cases for OO programs. Unlike many existing approaches that generate test cases entirely from the program under test, this paper proposes an approach that regenerates test cases based on the frequent sequential patterns of test repository. As a result, this approach can effectively generate new test cases that achieve good structural coverage for OO programs. What's more, if the test repository consists of manual test cases, the resulting test cases are more understandable than those generated entirely from the program under test.

In future work, we plan to tune the parameter settings for distinct programs under test, rather than use the general "best-practice" settings recommended in the literature, to further improve the performance of the approach. Besides, we want to study sequential pattern recommendation strategies so that sequential patterns can be chosen intelligently when there is more than one sequential pattern for a position of an individual during GA-based test case regeneration.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Fund of China under Grant No.61073035 and No.61170082. The authors are grateful to the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper.

REFERENCES

[1] J. A. Whittaker, "What is software testing? and why is it so hard?" *IEEE Softw.*, vol. 17, no. 1, pp. 70–79, Jan. 2000.

[2] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 85–103.

[3] L. Chen, "Automatic test cases generation for statechart specifications from semantics to algorithm." *JCP*, vol. 6, no. 4, pp. 769–775, 2011.

[4] S. Wang, Y. Ji, and S. Yang, "A micro-kernel test engine for automatic test system," *JCP*, vol. 6, no. 1, pp. 3–10, 2011.

[5] L. Zhang and J. Kuang, "A new test data compression scheme," *JCP*, vol. 6, no. 7, pp. 1297–1301, 2011.

[6] Z. Li, X. Li, G. Ye, and W. Yao, "Research on runtime environment of spacecraft testing system," *JCP*, vol. 6, no. 4, pp. 657–663, 2011.

[7] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Softw. Test. Verif. Reliab.*, vol. 22, no. 3, pp. 171–201, May 2012.

[8] A. Arcuri and X. Yao, "On test data generation of object-oriented software," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 72–76.

[9] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Min. Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, Aug. 2007.

[10] J. Wang, J. Han, and C. Li, "Frequent closed sequence mining without candidate maintenance," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 8, pp. 1042–1056, Aug. 2007.

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[12] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419.

[13] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, Mar. 2012.

[14] R. Gerlich, R. Gerlich, and T. Ball, "Random testing: from the classical approach to a global view and full test automation," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, ser. RT '07. New York, NY, USA: ACM, 2007, pp. 30–37.

[15] P. McMinn, "Search-based software test data generation: a survey: Research articles," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, June 2004.

[16] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, Mar. 2010.

[17] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA, USA: MIT Press, 1992.

[18] L. C. Briand, "Novel applications of machine learning in software testing," in *Proceedings of the 2008 The Eighth International Conference on Quality Software*, ser. QSIC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 3–10.

[19] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, Oct. 2005.

[20] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proceedings of the Third international conference on Search based software engineering*, ser. SSBSE'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 33–47.

[21] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth, "DyGen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces," in *Proceedings of the 4th international conference on Tests and proofs*, ser. TAP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 77–93.

[22] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "MSeqGen: object-oriented unit-test generation via mining source code," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 193–202.

[23] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: object capture-based automated testing," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 159–170.

[24] G. Fraser and A. Zeller, "Exploiting common object usage in test case generation," in *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 80–89.

Wei He was born in 1984. He received his B.S. degree in computer science and technology from Beijing University of Chemical Technology, China, in 2006. Currently, he is a Ph.D. candidate at the same university. His research interests include program analysis and software testing.

Ruilian Zhao received her B.S. and M.S. degree in computer science from North China Industry University in 1985 and 1990, and Ph.D. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences in 2001. She is now a professor at Department of Computer Science, Beijing University of Chemical Technology. Her primary research interests include software testing and fault-tolerant computing.