Chinese Journal of Electronics
Vol.24, No.1, Jan. 2015

Integrating Evolutionary Testing with Reinforcement Learning for Automated Test Generation of Object-Oriented Software*

HE Wei, ZHAO Ruilian and ZHU Qunxiong

(*Department of Computer Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China*)

Abstract — Recent advances in evolutionary test generation greatly facilitate the testing of Object-oriented (OO) software. Existing test generation approaches are still limited when the Software under test (SUT) includes Inherited class hierarchies (ICH) and Non-public methods (NPM). This paper presents an approach to generate test cases for OO software *via* integrating evolutionary testing with reinforcement learning. For OO software with ICH and NPM, two kinds of particular isomorphous substitution actions are presented and a Q-value matrix is maintained to assist the evolutionary test generation. A prototype called EvoQ is developed based on this approach and is applied to generate test cases for actual Java programs. Empirical results show that EvoQ can efficiently generate test cases for SUT with ICH and NPM and achieves higher branch coverage than two state-of-the-art test generation approaches within the same time budget.

Key words — Object-oriented software, Evolutionary testing, Reinforcement learning, Inherited class hierarchies, Non-public methods.

I. Introduction

Automated test generation of Object-oriented (OO) software has been a sustained topic of interest due to its benefit in terms of quality improvement and cost saving[1]. It aims at automatically creating a suite of test cases that achieves a specified coverage criterion, such as branch coverage for the Software under test (SUT), each test case consisting of a method call sequence that cover (or reach) a target branch. Although existing approaches, such as Random testing (RT)[2] and Evolutionary testing (ET)[3], have made notable progress in developing workable test generators for OO software, relatively little attention has been paid to test generation for SUT with special OO features, such as Inherited class hierarchies (ICH) and Non-public methods (NPM). Here, ICH refers to a set of collaborative classes that are organized hierarchically on account of the inheritance mechanism, whereas NPM refers to methods visible only within some classes as a result of the encapsulation mechanism.

In OO software, the inheritance mechanism allows one class to be reused as a basis for another class, thus classes are created in hierarchies, forming an ICH. A subclass at lower level defines a more specific data type than its superclass. When test generation is conducted for a branch which is guarded by a type checking (*i.e.* the "instance of" condition), it is essential to produce a method call sequence that constructs an object of the desired data type. Unfortunately, the amount of data types defined in an ICH maybe large and the inheritance relationships of these data types maybe intricate. Hence, it is hard to find a method call sequence that creates an object of required data type for covering the target branch.

As regards the encapsulation mechanism, details of classes are hidden and only public interfaces are exposed. If a method is not declared as public, namely an NPM, it cannot be freely invoked outside the class. Thus, when a tester wants to inspect a branch which is located in a non-public method, he must firstly look for some public methods that can call the non-public method. Typically, several public methods can invoke the non-public method, but maybe only a few of these public methods are able to enter the non-public method and lead to the target branch. Hence, it is difficult to produce such a method call sequence that invokes a specific public method which can call the non-public method and cover the target branch.

In test generation for SUT with ICH, most existing approaches only consider some frequently-used data types. For example, Refs.[3,4] simply dealt with four built-in data types and several user-defined data types. Meanwhile, current approaches provide little assistance to generate test cases for SUT with NPM. Researches like Refs.[2,3,5] neglected branches in non-public methods when producing test cases. Since ICH and NPM are widely used in OO software, these approaches are still defective in achieving high branch coverage.

Reinforcement learning (RL) is a popular machine learning strategy[6]. It concerns with identifying optimal (or near optimal) actions at different states to achieve a goal through

*Manuscript Received Dec. 2012; Accepted May 2013. This work is supported by the National Natural Science Foundation of China (No.61073035, No.61170082).
 DOI:10.1049/cje.2015.01.007

trial and error[7]. For OO software, invoking different methods can be viewed as performing distinct actions. Hence, taking a branch as a test goal, the task of creating a test case, namely a method call sequence for covering the branch, can be transformed into looking for a series of optimal actions to achieve the goal. Obviously, RL can be applied in test generation for OO software.

With the inspiration of RL, we introduce two kinds of particular actions, data type isomorphous substitution actions and access scope isomorphous substitution actions, to assist the test generation for SUT with ICH and NPM, respectively. For SUT with ICH, a superclass may have many subclasses, but only specific method call sequences can construct an object of the desired data type satisfying the type checking. So, data type isomorphous substitution action is presented to replace a method of a method call sequence with another method whose return type is a subclass of the original method's. Similarly, for SUT with NPM, there are usually some public methods that can invoke the non-public method containing the target branch, but only a few public methods with specific argument values can invoke the non-public method and reach the target branch. Thus, access scope isomorphous substitution action is proposed to replace a public method with another one that can call the non-public method. As a result of an isomorphous substitution action, a new method call sequence is produced. By trying out different isomorphous substitution actions at relevant call sites of a method call sequence, and maintaining the reinforcement knowledge that measures the quality of each action, a method call sequence for covering the target branch can be produced.

This paper presents an approach that integrates ET with RL to generate method call sequences as test cases for OO software. The general idea behind this approach is that ET evolves a population of candidate method call sequences with respect to target branches of SUT, if a target branch is guarded by a type checking and/or is located in a non-public method, an adapted RL process is employed to perform isomorphous substitution actions to produce new method call sequences. In particular, Genetic programming fulfills the role of evolving candidate method call sequences in ET. Q-learning, a popular RL technique, is adapted to perform isomorphous substitution actions and compute Q-values to measure the reinforcement knowledge about the quality of the actions to produce a method call sequence approaching to the target branch. The knowledge obtained from Q-learning is used to direct the search towards pertinent individuals. A corollary of this approach is that the performance of test generation can be greatly increased.

The rest of this paper is organized as follows. Section II provides the background underlying the research. Section III elaborates our approach to test generation. Empirical evaluations on this approach are described in Section IV. Section V reviews related work. Finally, Section VI concludes the paper.

II. Background

This section describes Evolutionary testing of OO software, Genetic programming, and Reinforcement learning, which our approach is based on.

1. Evolutionary testing of OO software

Evolutionary testing (ET) is a subcategory of search-based testing approaches[8]. It formulates the task of test generation as a search problem, and applies Evolutionary algorithms (EA), such as Genetic algorithms (GA)[9] or Genetic programming (GP)[10], to solve the problem.

To generate test cases for OO software, ET takes the SUT as input, and tries to find a group of method call sequences as test cases with respect to a selected coverage criterion such as branch coverage. This is to say, for each target branch, an initial population is constructed at first, where individuals are method call sequences generated randomly. Then, each individual is executed and evaluated by a fitness function that measures the difference between the execution trace of an individual and the target branch. If an individual reaches the target branch, a test case is produced and ET continues to search new test cases for covering the rest target branches. Otherwise, if the target branch is not covered by all candidate method call sequences, a new population is regenerated by genetic operations and evaluated by the fitness function. The cycle of regeneration and evaluation is repeated until the selected coverage criterion is satisfied or a pre-assigned termination condition is met.

Particularly, for path-oriented test generation, the fitness function usually combines a major component, called approach level, with a minor component, known as branch distance[8]. The former indicates how close the execution trace of an individual is to reaching the target branch, whereas the latter measures how far the execution trace of the individual takes the desired alternative branch at the first mismatching predicate. Quantitatively, the fitness function can be formulated by the approach level plus normalized branch distance[11]:

$$fitness = approach\_level + norm(branch\_distance) \quad (1)$$

where the branch distance is normalized with the formula:

$$norm(branch\_distance) = 1 - 1.001^{-branch\_distance} \quad (2)$$

ET tries to minimize the fitness value where 0 indicates that the individual covers the target branch.

2. Genetic programming

GP is a specialization of EA that aims at creating computer programs based on the principles of natural evolution[10]. When GP is employed to solve an actual problem, a set of candidate programs is mapped to a population of tree-shaped individuals. Every individual is represented with a finite tree structure, where specialized genetic operations can be performed.

For each individual of GP, the leaf nodes are called terminals, whereas the non-leaf nodes are called non-terminals. Terminals represent inputs of the program, constants or methods with no arguments. Non-terminals typically denote methods taking at least one argument.

GP has been used to solve a wide range of practical problems. According to the viewpoint of Refs.[4,12], a test case is a special kind of computer program. Thus, GP can be used in test generation.

3. Reinforcement learning

RL is a fundamental machine learning strategy. It is employed to acquire the optimal actions at each state to achieve a goal *via* maximizing cumulative reward[6,7].

Q-learning, a primary RL technique, works by learning from temporal rewards which measure the quality of each action at each state with respect to the goal[7,13]. For Q-learning, a set of states S and corresponding acceptable actions A should be defined at first. Besides, a matrix of Q-values is maintained to denote the cumulative reward, each element $Q[s,a]$ representing the quality of action a at state s if the action is performed in order to reach the goal. By trying out acceptable actions at different states gradually and assigning corresponding reward based on the impact of each action with respect to the goal, the Q-value matrix can be updated using Formula 3. Finally, a series of actions to reach the goal is identified according to the Q-value matrix.

$$Q[s,a] = \begin{cases} \alpha \times r, \text{ if } s' \text{ is reached for the first time} \\ (1-\alpha)Q[s,a] + \alpha \times \delta, \text{ otherwise} \end{cases} \quad (3)$$

where $$\delta = r + \gamma \max_{a'} Q[s',a'] - Q[s,a] \quad (4)$$

and γ is a user-defined discount factor.

III. Automated Test Generation for OO Software *via* Integrating ET with RL

This section details our approach to generate test cases for OO software. Taking a branch of the SUT as current target branch, a population of candidate method call sequences (represented by tree-shaped individuals) is randomly initialized. Then, a GP-based ET process starts. The fitness of each individual is evaluated by Eq.(1). If the target branch is not covered by any individual, an elite individual with highest fitness is selected for further refinement. Particularly, if the target branch is guarded by a type checking and/or is located in a non-public method, a Q-learning-based RL process is activated. Isomorphous substitution actions are performed and the corresponding Q-values are computed to measure the effects of the actions, trying to obtain a variant individual approaching to the target branch. Typically, there are multiple variant individuals generated since the isomorphous substitution actions can be performed at different call sites. An ideal variant individual is produced by an action with highest Q-value. After that, a poor individual with lowest fitness value in current population is replaced by the ideal variant individual. Then, general genetic operations, namely recombination and mutation, are implemented on current population including the ideal variant individual to generate a new population. Otherwise (*i.e.* the target branch does not involve ICH or NPM), the population is evolved by directly employing general genetic operations.

As a whole, this approach gears GP-based ET and Q-learning-based RL together to generate test cases for OO software. The knowledge obtained from Q-learning-based RL can help to direct GP-based ET towards pertinent individuals that may cover target branches, and thereby increases the performance of test generation for SUT with ICH and NPM. A high-level pseudocode for this approach is given in Algorithm 1.

1. Preliminary definitions

A method call sequence is made of a series of method invocations with some variables as the arguments. Either a variable or a method can be concisely represented by a signature[3]. A variable signature is composed of name, data type and value of the variable. A method signature consists of name and data types of its arguments (in order). Based on its signature, a variable or a method can be distinguished from others. However, the information on return type and access scope is not included in the signatures. This information is necessary in generating method call sequences for SUT with ICH and NPM. Therefore, in our approach extended variable signature and extended method signature are defined to indicate a variable and method, respectively.

Algorithm 1 Pseudocode for automated test generation

```
Input software under test SUT
Output test suite T
Instrument SUT for tracing executions;
Identify all variables V, methods M and branches B of SUT;
T = ∅;
foreach branch b ∈ B do
    Generate random tree-shaped population P with related V and M;
    foreach node of individual i ∈ P do
        Initialize Q[node, isa];
    repeat
        foreach individual i ∈ P do
            Decode i to method call sequence mcs;
            Execute mcs;
            if mcs covers b then
                Add mcs to T;
                break;
            Evaluate fitness of i;
        Get elite individual ei by Select(P);
        if b is guarded by type checking condition tcc then
            DataTypeISA(ei); // get variant individuals, update Q[node, isa]
        if b is located in non-public method npm and ei reaches npm then
            AccessScopeISA(ei); // get variant individuals, update Q[node, isa]
        Get ideal variant individual vi* produced with highest Q[node, isa];
        Replace a poor individual pi with vi* in P;
        GeneticOperations(P);
    until termination condition is met;
```

Definition 1 Extended variable signature (EVS) An $evs = (n, s, t, v)$ is a tuple, where n is a variable's name, s and t denote its access scope and data type, respectively, and v records the value of the variable. A set of all extended variable signatures can be represented by $EVS = (N, S, T, V)$.

Definition 2 Extended method signature (EMS) An $ems = (n, s, t_{rec}, t_{arg1}, \cdots, t_{argn}, t_{ret})$ is a tuple, where n is a method's name, s denotes its access scope, t_{rec} indicates the data type of its receiver object, $t_{arg1}, \cdots, t_{argn}$ corresponds to the data type of each argument, and t_{ret} refers to its return type. The notation $EMS = (N, S, T_{rec}, T_{arg1}, \cdots, T_{argn}, T_{ret})$ represents a set of all extended method signatures.

In EVS, the access scope information is new recruit; whereas in EMS, both the return type and the access scope

information are new recruits. All the information of *evs* and *ems* can be directly extracted from the declaration statements of the SUT by a static analysis.

2. Representation of method call sequences

GP is a specialization of EA with tree-shaped structure as individual representation. When GP is employed to solve an actual problem, each candidate solution needs to be mapped into a finite tree structure on which genetic operations can be performed.

A method call sequence is a series of method invocations $\langle m_1; m_2; \cdots; m_i; \cdots; m_n \rangle$, where method m_i, $i \in [1, \cdots, n]$, can be represented by an extended method signature along with some extended variable signatures as its arguments. Accordingly, each method call sequence can be expressed by a tree-shaped structure, referring to $GPTree = (Node,\ Edge,\ root)$, where a $node \in Node$ denotes either an *ems* or an *evs*, an $edge \in Edge$ from a parent *node* to a child *node* represents that the parent is dependent on the child (*i.e.* the method denoted by the parent takes the object created by the child as an receiver object or an argument), and the *root* is an entrance method. In other words, a method call sequence can be represented as a tree-shaped individual. *Vice versa*, an individual can be decoded into a method call sequence by a depth-first traversal.

3. Isomorphous substitution actions

As mentioned in Section I, given an SUT with ICH and NPM, existing approaches are deficient in producing test cases for covering target branches guarded by type checking conditions and/or located in non-public methods. Based on the idea of RL, two kinds of isomorphous substitution actions are proposed to aid the test generation for SUT with ICH and NPM, respectively. This section explains these isomorphous substitution actions in details.

1) Data type isomorphous substitution actions are presented to produce method call sequence for covering the target branches guarded by type checking conditions. The general idea is to replace a method of a method call sequence with another method (as well as its arguments) whose return type is a subclass of the original method's, hoping to obtain a new method call sequence approaching to current target branch. That is to say, for a *node* in a $GPTree$ individual that influences relevant type checking conditions of the target branch, its *ems* is replaced with an alternative ems' according to the ICH such that $t_{ret'} \triangleright t_{ret}$ ($t_{ret'}$ inherits t_{ret}), where t_{ret} and $t_{ret'}$ are return type of *ems* and ems', respectively. As a result, method call sequences that involve objects of inherited data types can be produced. Then, the qualtiy of each action is evaluated by computing its Q-value with respect to the target branch. The pseudocode for data type isomorphous substitution actions is displayed in Algorithm 2.

Note that, if the maximum depth of an ICH is D and the predefined maximum amount of *node* in a $GPTree$ individual is N, then the amount of variant individuals produced by data type isomorphous substitution actions is upper bounded by $D \times N$. As Booch suggested, class hierarchies should be no deeper than 7±2 in practice[14]. Hence, the computational effort of data type isomorphous substitution actions is feasible.

Algorithm 2 Pseudocode for data type isomorphous substitution actions

```
Input elite individual ei
foreach node of ei influencing type checking condition tcc do
    if ∃ data type isomorphous substitution action isa for node
      and rand[0,1) < Q[node, isa] then
        Substituting ems with ems' to produce variant individual
        vi, where t'_rct ▷ t_ret; decode vi to method call sequence
        mcs;
        Receive reward r by executing mcs;
        Update Q[node, isa] with r;
```

2) Access scope isomorphous substitution actions are introduced to produce method call sequence for covering target branches located in non-public methods. The basic idea is to replace a public method of a method call sequence with another public method (with proper arguments) that can invoke the non-public method *npm*. More specifically, the public methods that perhaps call the non-public method are obtained by employing a static analysis on the SUT beforehand. In a $GPTree$ individual, if a *node* invokes *npm* but does not reach the target branch, we replace *ems* with an alternative ems' which is another public method that calls *npm*. Then, Q-value of the action performed is also computed to measure the quality of the action with respect to the target branch. The pseudocode for access scope isomorphous substitution actions is presented in Algorithm 3.

Algorithm 3 Pseudocode for access scope isomorphous substitution actions

```
Input: elite individual ei
foreach node of ei invoking non-public method npm do
    if ∃ access scope isomorphous substitution action isa for
    node
      and rand[0,1) < Q[node, isa] then
        Substitute ems with ems' to produce variant individual
        vi, where ems' invokes npm;
        Decode vi to method call sequence mcs;
        Receive reward r by executing mcs;
        Update Q[node, isa] with r;
```

Given that the number of public methods that can invoke *npm* is M and the predefined maximum amount of *node* in a $GPTree$ individual is N, then the upper bound of variant individuals produced by access scope isomorphous substitution actions is $M \times N$, which is finite in real programs. So, the computational complexity is also manageable.

4. Q-value computation

In test generation, Q-learning is employed to try out acceptable isomorphous substitution actions at relevant call sites with respect to a target branch, maintains a matrix of Q-values indicating the effects of corresponding isomorphous substitution actions, and finally produce a new method call sequence that covers the target branch according to the matrix.

When an isomorphous substitution action *isa* is performed at a *node* to produce a variant individual *vi*, the matrix element Q[*node,isa*] is computed by Formula 5 to evaluate the quality of the action with respect to the target branch. More specifically, the Q-value is computed based on whether the resulting variant individual *vi* reaches a state that has been reached before. If the state is first reached, the corresponding Q-value Q[*node,isa*] is initialized by the temporal reward

r with a learning rate α. Otherwise (*i.e.* the state is reached again), the value of $Q[node,isa]$ is updated based on current $Q[node,isa]$ and the best Q-value that can be achieved with vi, *i.e.* $\max_{isa'} Q[node', isa']$, where isa' is any isomorphous substitution action that can be performed at $node'$ of vi. The higher $Q[node,isa]$ is, the isomorphous substitution action isa is better to produce a variant individual approaching to the target branch.

$$Q[node, isa] = \begin{cases} \alpha \times r, \text{ if the state is first reached} \\ (1-\alpha)Q[node, isa] + \alpha \times \delta, \text{ otherwise} \end{cases} \tag{5}$$

where

$$\delta = r + \gamma \max_{isa'} Q[node', isa'] - Q[node, isa] \tag{6}$$

In Eqs.(5) and (6), both the learning rate $\alpha \in [0, 1]$ and the discount factor $\gamma \in [0, 1]$ are user defined parameters. Typical values of $\alpha = 0.1$ and $\gamma = 0.9$ for Q-learning are used.

According to the Q-value matrix, an ideal variant individual vi^* approaching to the target branch can be produced by the action with highest Q-value (refer to Line 21 in Algorithm 1). A poor individual pi with lowest fitness value in current population is replaced by this variant individual vi^* so that direct the search towards pertinent individuals.

5. Genetic operations

In our approach, the GP-based ET process proceeds by applying two kinds of general genetic operations, namely recombination and mutation, to generate offspring individuals. The pseudocode for performing genetic operations is given in Algorithm 4.

Algorithm 4 Pseudocode for genetic operations

```
Input current population P
1 if rand[0,1) < recombination probability p_r then
2 |  Recombine(P);
3 if rand[0,1) < mutation probability p_m then
4 |  Mutate(P);
```

1) Recombination operation generates two offspring individuals on the basis of two parent individuals. One-point crossover strategy, a frequently-used recombination strategy, is employed in our GP-based ET process. In other words, a subtree of one parent is exchanged with another subtree of the other parent. To ensure that the resulting individuals are legal after recombination, the choice of crossover point is a little tricky: the two subtrees to be exchanged must be compatible, meaning that the return type and the access scope of the root methods in the two subtrees must be the same, respectively. If no such subtree exists in the parent individuals, recombination is not performed.

2) Mutation operation generates an offspring individual by altering a gene of a parent individual. More specifically, in a $GPTree$ individual, three mutation strategies can be employed. ① Promotion strategy moves a subtree of a parent to one descendant *node* with compatible return type and access scope. ② Demotion strategy moves a subtree of a parent to one ancestor *node* with compatible return type and access scope. ③ Change strategy turns a subtree of a parent to a new subtree with compatible return type and access scope. The meaning of "compatible" here is the same as in recombination. Similarly, if no compatible subtree exists in the parent individual, mutation is not performed.

IV. Empirical Evaluations

To evaluate the effectiveness of our approach, we developed a prototype called EvoQ (available online at http://code.google.com/p/evoq/), and compared it with Randoop (a feedback-directed random test generator), and eToc (a GA-based evolutionary test generator) on generating method call sequences as test cases for 8 Java standard library classes and 4 open source Java packages including many classes, respectively. All empirical evaluations were conducted on a HP ProLiant Server with Intel Xeon 2.40GHz×16 processors, 4GB×6 RAMs, 64-bit CentOS Linux 6.0 and JDK 1.6.0.

1. Experimental subjects

Two series of empirical experiments were conducted on Java programs. The first experiment was performed at class-level subjects (8 frequently-used classes of the Java runtime environment (JRE) as shown at the upper part of Table 1); whereas the second experiment was carried out at package-level subjects (4 open source Java packages that include more classes as shown at the lower part of Table 1).

Table 1. Statistics on experimental subjects

Subject		#Classes	#Methods		#Branches			LOC
			All	Non-public	All	Type-checking	Non-public	
JRE classes	ArrayList (AL)	1	27	4	78	0	12	218
	BitSet (BS)	1	41	10	168	2	52	388
	HashMap (HM)	9	70	28	148	8	63	512
	HashSet (HS)	1	15	3	10	2	6	78
	LinkedList (LL)	4	61	8	122	0	22	417
	StringTokenizer (ST)	1	13	4	72	0	18	153
	TreeMap (TM)	22	238	93	272	14	82	1455
	TreeSet (TS)	1	33	3	26	4	5	147
	Total	**40**	**498**	**153**	**896**	**30**	**260**	**3368**
whole packages	Commons collections (CC)	382	3182	130	6276	450	1004	26323
	Commons primitives (CP)	231	1756	217	1446	154	162	9836
	JTopas (JT)	63	719	53	1376	34	293	5361
	NanoXML (NX)	24	317	43	690	6	225	3279
	Total	**700**	**5874**	**443**	**9788**	**644**	**1684**	**44799**

Table 1 summarizes statistics of the experimental subjects in terms of the number of classes (#Classes), the number of all methods (#Methods: All) including non-public methods (#Methods: Non-public), the number of all branches (#Branches: All) including branches guarded by type checking conditions (#Branches: Type-checking) and branches located in non-public methods (#Branches: Non-public), and non-commenting lines of code (LOC).

Concerning the 8 JRE classes, the number of methods and branches therein varies from 13 to 238 and 10 to 272, respectively. Accordingly, the size of these subjects ranges from less than 100 LOC to more than 1400 LOC. Totally, these subjects contain more than 3K LOC.

As regards the 4 open source packages, the largest one CC consists of more than 20K LOC, up to nearly 400 classes, more than 3K methods and 6K branches. In total, the four packages include nearly 45K LOC.

2. Experimental setup

EvoQ was configured according to the recommended settings in Ref.[15]. In this regard, the time budget for test generation of each class was no more than 600 seconds. The GP population size was 100, and initial population was generated randomly. Generic operations were set to: rank selection with 1.7 bias; one-point crossover with probability of 0.75; mutation probability of 0.3 each for promotion, demotion and change.

For Q-learning, the learning rate α was 0.1, and the discount factor γ was 0.9. If a new state was reached with an isomorphous substitution action, the temporal reward r of the action was 0.1; otherwise, the action was punished with a penalty –0.05. If a target branch is reached after performing an isomorphous substitution action, the action was encouraged with an evident reward 0.5.

Randoop and eToc were configured with identical settings to EvoQ when needed. In addition, Randoop, eToc and EvoQ were all based on randomized algorithms, which might be strongly affected by chance. Thus, each experiment was repeated 30 times, and statistical methods were used to analyze the empirical results.

3. Empirical experiment at class level

In this section, we compare the effectiveness of EvoQ with Randoop and eToc at class level. For each subject, the branch coverage achieved by each test generator (averaged over the 30 runs) is reported in Table 2. The maximum branch coverage achieved for each subject is marked in bold. For each automated test generator, the average branch coverage achieved is given at the bottom. Notice that the branch coverage refers to the percentage of covered branches that are located not only in the public methods but also in the non-public methods.

It can be seen from Table 2 that EvoQ can generate method call sequences with highest branch coverage for all subjects except StringTokenizer: on average nearly 80%, with more than 22% improved from only 57.1% by Randoop, and with about 14% improved from 65.4% by eToc, respectively. Clearly, EvoQ outperforms Randoop and eToc on test generation for Java programs.

EvoQ outperforms Randoop mainly because Randoop is a random generator. It is well known that generating test cases randomly has little chance to reach a deep branch of the SUT. Besides, EvoQ performs better than eToc (except StringTokenizer). A possible reason is that EvoQ introduces isomorphous substitution actions for covering branches that involves ICH and NPM. Moreover, the improvement of EvoQ over Randoop and eToc is statistically significant at the 95% confidence level based on one-tailed Mann-Whitney U test.

Table 2. Branch coverage (in %) achieved by Randoop, eToc and EvoQ at class level

Subject	Randoop	eToc	EvoQ
AL	68.2	84.4	**87.8**
BS	77.7	53.9	**85.1**
HM	62.8	74.2	**90.3**
HS	40.0	40.0	**70.0**
LL	60.6	81.4	**96.2**
ST	59.4	**81.0**	62.8
TM	23.3	40.2	**57.8**
TS	64.7	68.2	**79.4**
Avg.	57.1	65.4	**79.4**

The special case is that eToc achieves higher branch coverage than EvoQ for StringTokenizer. It is largely due to the large amount of string objects in the StringTokenizer class. eToc is good at constructing complex string objects since it takes string as one of its built-in data types and generate string values without resorting to an explicit constructor invocation.

4. Empirical experiment at package level

In this section, we compared the test generation effectiveness of EvoQ with Randoop and eToc at package level by applying them to generate method call sequences as test cases for 4 open source Java packages, respectively (over all classes therein, each test generator are repeated 30 times). The empirical result is given in Table 3. Bold numbers represent the maximum branch coverage achieved for each subject.

From Table 3, it can be found that EvoQ works well on automated test generation, performing better than both Randoop and eToc for all of the four packages. The average branch coverage (also taken into account the branches located in non-public methods) achieved by EvoQ for each package ranges from more than 75% to nearly 90%. The average increment of EvoQ against Ranoop and eToc is up to nearly 47% and 18%, respectively. The improvement is also statistically significant at the 95% confidence level based on one-tailed Mann-Whitney U test.

Table 3. Branch coverage (in %) achieved by Randoop, eToc and EvoQ at package level

Subject	Randoop	eToc	EvoQ
CC	20.9	74.8	**89.5**
CP	62.8	75.4	**87.6**
JT	49.0	57.1	**75.2**
NX	15.8	56.7	**83.2**
Avg.	37.1	66.0	**83.9**

V. Related Work

Since manually creating test cases is hard and expensive, automated test generation has been a sustained topic of interest over the past decade. Generally speaking, existing approaches to automated test generation for object-oriented software can be classified into two major categories: Random testing[2,16,17] and Search-based testing[3,5,12,18] approaches.

Random testing (RT) approaches are simple and efficient to generate test cases[19]. They scale well and achieve fairly good code coverage. However, for OO software, RT generators such as Randoop[2], JCrasher[16] and RUTE-J[17] usually produce long and complex test cases that are hard to understand by human. Besides, many randomly generated test cases tend to cover the same target branch, thus some of them are redundant. Meanwhile, RT generators are hard to get particular test cases that cover some deep code fragments guarded by nested branch conditions.

Randoop[2] is one of the most advanced RT generators. It generates method call sequences incrementally for Java programs by randomly invoking methods of the SUT. For each randomly invoked method, Randoop uses random values and previously generated method call sequences for arguments of primitive data types and non-primitive data types, respectively.

In contrast, our EvoQ generates method call sequences for each target branch with the guidance of a fitness function and Q-values. It is more likely to cover deep branches of the SUT.

Search-based testing (SBT) approaches are believed promising for generating high quality test cases[8,20]. They formulate the task of automated test generation as a search problem and solve it *via* meta-heuristic search algorithms. As a subcategory of SBT, Evolutionary testing (ET) approaches have attracted much attention. Successful ET tools like eToc[3], EvoUnit[12], TestFul[18] and EvoSuite[5] have been developed to generate test cases for OO container classes and open source libraries. All these tools have made notable progress in automated test generation of OO software. However, automated test generation for OO software with ICH and NPM by ET is still challenging.

Among these tools, eToc is a representative ET generator. It uses GA to produce test cases for Java programs[3]. In eToc, individuals are directly represented as method call sequences with argument objects. Five special mutation operations are proposed to breed new offspring. After a mutation, adjustments are required to handle the conflicts among method invocations. It prescribes only four built-in data types and several user-defined data types. Additionally, branches inside non-public methods are not taken into consideration.

On the contrary, our EvoQ has implemented test generation for SUT with more general data types and a lot of non-public methods. The candidate method call sequences are evolved based on GP. Besides, Q-learning is employed to cover target branches guarded by type checking conditions or located in non-public methods. Thus, EvoQ is more flexible and effective than eToc.

VI. Conclusion and Future Work

OO methodology has been increasingly used in software development industry. While OO development techniques and tools have become relatively mature through long-term research and practice, testing of OO software is still challenging.

This paper proposes an automated test generation approach, which integrates ET with RL, to produce test cases for SUT with ICH and NPM. The new twist takes advantages of both ET and RL, and thereby improving the performance of test generation. To the best of our knowledge, there is no previous work on adapting RL to automated test generation.

Two empirical studies are conducted on various Java programs with totally over 48K LOC. The empirical results show that EvoQ can successfully generate test cases for SUT with ICH and NPM and achieves higher branch coverage than Randoop and eToc at both class level and package level.

There are two aspects that can be investigated in future research. First, in our empirical evaluations, parameters are configured according to a group of recommended settings. Tuning the parameters for distinct SUT may further improve the performance of EvoQ. Second, multi-threading is not considered currently. In the future, it can be extended into EvoQ to generate test cases for programs that involves multi-threading.

References

[1] A. Arcuri and X. Yao, "On test data generation of object-oriented software", *Proc. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp.72–76, 2007.

[2] C. Pacheco, S.K. Lahiri, M.D. Ernst and T. Ball, "Feedback-directed random test generation", *Proc. 29th International Conference on Software Engineering*, pp.75–84, 2007.

[3] P. Tonella, "Evolutionary testing of classes", *Proc. International Symposium on Software Testing and Analysis*, pp.119–128, 2004.

[4] A. Seesing and H.G. Grob, "A genetic programming approach to automated test generation for object-oriented software", *ITSSA*, Vol.1, No.2, pp.127–134, 2006.

[5] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software", *Proc. 19th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering*, pp.416–419, 2011.

[6] L.P. Kaelbling, M.L. Littman and A.W. Moore, "Reinforcement learning: A survey", *J. Artif. Int. Res.*, Vol.4, No.1, pp.237–285, 1996.

[7] R.S. Sutton and A.G. Barto, *Introduction to Reinforcement Learning*, Cambridge, MA, USA, 1998.

[8] P. McMinn, "Search-based software test data generation: A survey: Research articles", *Softw. Test. Verif. Reliab.*, Vol.14, No.2, pp.105–156, 2004.

[9] J.H. Holland, *Adaptation in Natural and Artificial Systems*, Cambridge, MA, USA, 1992.

[10] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA, USA, 1992.

[11] J. Wegener, A. Baresel and H. Sthamer, "Evolutionary test environment for automatic structural testing", *Information & Software Technology*, Vol.43, pp.841–854, 2001.

[12] S. Wappler and J. Wegener, "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming", *Proc. 8th Annual Conference on Genetic and Evolutionary Computation*, pp.1925–1932, 2006.

[13] C.J.C.H. Watkins and P. Dayan, "Q-learning", *Mach. Learn.*, Vol.8, No.3–4, pp.279–292, 1992.

[14] G. Booch, *Object-oriented Design with Applications*, Addison-Wesley, Boston, MA, USA, 2007.

[15] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering", *Proc. 3rd International Conference on Search Based Software Engineering*, pp.33–47, 2011.

[16] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java", *Softw. Pract. Exper.*, Vol.34, pp.1025–1050, 2004.

[17] J.H. Andrews, S. Haldar, Y. Lei and F.C.H. Li, "Tool support for randomized unit testing", *Proc. 1st International Workshop*

on Random Testing, pp.36–45, 2006.

[18] L. Baresi and M. Miraz, "Testful: Automatic unit-test generation for java classes", *Proc. 32nd International Conference on Software Engineering*, pp.281–284, 2010.

[19] 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, 2012.

[20] S. Ali, *et al.*, "A systematic review of the application and empirical investigation of search-based test case generation", *IEEE Trans. Softw. Eng.*, Vol.36, No.6, pp.742–762, 2010.

HE Wei was born in 1984. He received the 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. (Email: wei.he@live.cn)

ZHAO Ruilian (corresponding author) received the Ph.D. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences in 2001. She is now a professor and Ph.D. supervisor at Beijing University of Chemical Technology. Her primary research interests include software testing and fault-tolerant computing. (Email: rlzhao@mail.buct.edu.cn)

ZHU Qunxiong was born in Wuxi, China. Currently, he is a professor and Ph.D. supervisor at Beijing University of Chemical Technology. His research interests include intelligence engineering, fault diagnosis, *etc.* (Email: zhuqx@mail.buct.edu.cn)