Random Unit-Test Generation with MUT-aware Sequence Recommendation

Wujie Zheng, Qirun Zhang, Michael Lyu
Computer Science and Engineering
The Chinese University of Hong Kong, China
{wjzheng,qrzhang,lyu}@cse.cuhk.edu.hk

Tao Xie
Department of Computer Science
North Carolina State University, USA
xie@csc.ncsu.edu

ABSTRACT

A key component of automated object-oriented unit-test generation is to find method-call sequences that generate desired inputs of a method under test (MUT). Previous work cannot find desired sequences effectively due to the large search space of possible sequences. To address this issue, we present a MUT-aware sequence recommendation approach called RecGen to improve the effectiveness of random object-oriented unit-test generation. Unlike existing random testing approaches that select sequences without considering how a MUT may use inputs generated from sequences, RecGen analyzes object fields accessed by a MUT and recommends a short sequence that mutates these fields. In addition, for MUTs whose test generation keeps failing, RecGen recommends a set of sequences to cover all the methods that mutate object fields accessed by the MUT. This technique further improves the chance of generating desired inputs. We have implemented RecGen and evaluated it on three libraries. Evaluation results show that RecGen improves code coverage over previous random testing tools.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability

1. INTRODUCTION

Unit testing is one of the most commonly used techniques to assure high quality of software systems. A primary objective of unit testing is to achieve high structural coverage such as statement coverage. To this end, a method under test (MUT) needs to be executed with specific inputs. For object-oriented programs, the desired inputs including the receiver and arguments of a MUT are often objects that have specific values in their fields. As directly modifying object fields may violate class invariants, it is necessary to employ method-call sequences (in short as *sequences*) to create and mutate objects to generate desired inputs. Therefore, a key component of object-oriented unit-test generation is to find method-call sequences that generate desired inputs of a MUT.

There are two main approaches to generate desired method-call sequences: random sequence generation and bounded-exhaustive sequence generation. Random sequence generation approaches generate sequences randomly from the whole space. Csallner and Smaragdakis [8] developed a random testing tool named JCrasher to generate sequences randomly. Pacheco et al. [12] proposed a feedback-directed random test generation approach and implemented it in a tool named Randoop. Adaptive Random Testing (ART) [7, 10] approaches such as ARTGen [10] select inputs evenly across the input space. These random sequence generation approaches cannot generate desired sequences effectively from the large space of possible sequences. Bounded-exhaustive sequence generation approaches [13, 16, 17] generate sequences exhaustively up to a small bound of sequence length. However, generating desired inputs, including the receiver and arguments, often requires longer sequences beyond the small bound that can be handled by the bounded-exhaustive approaches.

To address the challenges faced by these existing sequence-generation approaches, in this paper, we propose a MUT-aware sequence recommendation approach called RecGen to improve the effectiveness of random object-oriented unit-test generation. Unlike existing random testing approaches that select sequences without considering how a MUT may use inputs generated from sequences, RecGen analyzes object fields accessed (i.e., read and write) by a MUT and recommends a short sequence that mutates these fields. The rationale is that a sequence that mutates object fields accessed by a MUT has a higher chance to explore different behaviors of the MUT. RecGen first analyzes the methods that mutate object fields accessed by a MUT. These methods are called *relevant methods* of the MUT. RecGen then recommends a short sequence that consists of *relevant methods* of the MUT for generating the receiver or an argument. Finally, for MUTs whose test generation keeps failing (i.e., the resulting sequence is duplicate or throws uncaught exceptions), RecGen recommends a set of sequences to cover the MUT's all *relevant methods*. This technique further improves the chance of generating desired inputs.

To illustrate the idea of RecGen, we present an example MUT named `openDatabase`, which belongs to the `Envi-`

ASE'10, September 20–24, 2010, Antwerp, Belgium.

ronment class in the Berkeley DB Java Edition [5], in Figure 1. The MUT calls setupDatabase, which subsequently calls getAllowCreate to check the field allowCreate of the DatabaseConfig argument. To achieve high structural coverage of the MUT (and the private methods called by it) needs various inputs. In particular, to cover the code under the true branch of dbConfig.getAllowCreate in the setupDatabase method and create a new database, the field allowCreate of the DatabaseConfig argument of the MUT should have the true value, which by default has the false value and can be mutated by invoking DatabaseConfig.setAllowCreate.

```
// Environment.java
public synchronized Database openDatabase(
                      Transaction txn,
                      String databaseName,
                      DatabaseConfig dbConfig)
throws DatabaseException {
    checkHandleIsValid();
    checkEnv();
    try {
        if (dbConfig == null) {
            dbConfig = DatabaseConfig.DEFAULT;
        }
        Database db = new Database(this);
        setupDatabase(txn, db, databaseName,
                      dbConfig,
                      false,
                      false,
                      envImpl.isReplicated());
        return db;
    } catch (Error E) {
        envImpl.invalidate(E);
        throw E;
    }
}

private void setupDatabase(..., DatabaseConfig
                           dbConfig, ...)
throws DatabaseException {
    ...
    if (databaseExists) {
        ...
    } else {
        ...
        /* No database.
           Create if we're allowed to. */
        if (dbConfig.getAllowCreate()) {
            ...
        }
    }
    ...
}

// DatabaseConfig.java
public boolean getAllowCreate() {
    return allowCreate;
}

public void setAllowCreate(boolean allowCreate) {
    this.allowCreate = allowCreate;
}
```

Figure 1: The openDatabase MUT and related code

When selecting previously generated sequences for the DatabaseConfig argument, existing random testing tools have a low chance to select a sequence that includes the DatabaseConfig.setAllowCreate method. On the other hand, RecGen identifies that DatabaseConfig.setAllowCreate is a *relevant method* of the MUT since it mutates the field allowCreate that is accessed by the MUT. RecGen then recommends short sequences with the DatabaseConfig.setAllowCreate method, and thus improves the chance of generating desired sequences.

We have implemented RecGen and evaluated it on three Java libraries. Evaluation results show that RecGen improves code coverage over previous random testing tools [8, 10, 12]. RecGen and the evaluation data are available at

https://sites.google.com/site/recgentool/

2. APPROACH

2.1 Overview

RecGen accepts a set of methods or classes under test. By default, RecGen filters out subclasses of java.lang.Exception, since these classes are often used to handle exceptional cases and may corrupt the program states for further testing. RecGen then repeats the following steps to test the MUTs up to a given time limit.

- Select a MUT randomly.
- For each input (the receiver or arguments) of the MUT
  - If the input is a primitive value, randomly select a primitive value from a fixed pool of values.
  - If the input is an object, select a sequence using MUT-aware sequence recommendation.
  - If the input is an array, construct a random-sized array whose elements (primitive values or objects) are selected as described earlier.
- Generate a new sequence for the MUT by concatenating selected sequences with the MUT at the end.
- Add test oracles (the same as those used in Randoop [12]) to the new sequence.
- Execute the new sequence; only valid and unique sequences are used for further sequence generation.

2.2 MUT-aware Sequence Recommendation

2.2.1 Identifying Relevant Methods

RecGen identifies *relevant methods* for the methods under test by analyzing the code using the Eclipse JDT Compiler [2]. Basically, RecGen checks whether two methods may access the same object fields. If so, we consider the two methods relevant; otherwise, we consider them non-relevant. Let $N(f)$ denote the set of object fields that a method f may access. The relevance between a method g and f is defined as follows.

$$relevance(f,g) = \begin{cases} 1, & if\ |N(f) \cap N(g)| > 0 \\ 0, & otherwise \end{cases}$$

$relevance(f,g)$ indicates whether g may affect the execution of f, and vice versa.

To compute the set of object fields that a method may access, RecGen first analyzes which object fields a method accesses directly in its method body. RecGen then merges the set of object fields accessed by a given method and the methods that the given method calls. Actually, we are interested in only methods that may write object fields accessed by a MUT. However, our current implementation of RecGen does not distinguish read and write operations.

2.2.2 Recommending a Single Sequence

Correlation Weights. RecGen recommends sequences that include more *relevant methods* of a MUT and fewer other methods. The correlation of a sequence *seq* to a method f is defined as follows.

$$weight\_corr(seq, f) = \sum_{i=1}^{n} relevance(seq_i, f)/n$$

where seq_i is the ith method in the sequence and n is the total number of method calls in the sequence.

Size Weights. RecGen also recommends shorter sequences so as to search the space of short sequences more thoroughly. RecGen assigns size weights to sequences as follows.

$$weight\_size(seq, f) = 1/n$$

where n is the total number of method calls in the sequence.

Overall Weights. RecGen employs a normalization which divides the correlation weight of each sequence for a MUT f by the sum of all the correlation weights. RecGen employs the same normalization for the size weights of sequences. RecGen then combines the weights equally as follows:

$$weight(seq, f) = (weight\_corr(seq, f) + weight\_size(seq, f))/2$$

RecGen randomly selects a sequence to generate the input of a MUT based on the overall weights of sequences. In particular, the chance of a sequence to be selected is proportional to the ratio of its weight over the total weights of sequences.

Priority of Receiver Sequences. Finally, RecGen recommends to use the receiver sequence, i.e., the sequence that is used to generate the receiver, to generate arguments, if possible. Given an argument to generate, if the receiver sequence can produce objects of the argument type, RecGen selects the receiver sequence to generate the argument with a predefined probability, which is 0.5 by default.

2.2.3 Recommending a Set of Sequences

If the test generation of a MUT keeps failing, i.e., returning invalid or duplicate sequences, RecGen identifies that this MUT is difficult to test. Given an input of such a MUT, RecGen recommends a set of sequences, which cover all *relevant methods* of the MUT appearing in the candidate sequences, for the input. This technique avoids the problem of missing a desired *relevant method* by chance.

3. EVALUATIONS

3.1 Evaluation Setup

To evaluate our approach, we apply our approach on three Java libraries: a database library Berkeley DB [5], a data structure library JDSL [3], and a science computation library JScience [4].

We compare RecGen with three well-known random test generation approaches, including JCrasher [8], Randoop [12], and ARTGen [10]. JCrasher uses an undirected random testing approach, Randoop applies a feedback-directed random testing approach, and ARTGen employs an adaptive random testing approach. We run these tools on each subject library for two minutes (JCrasher does not have a time limit option and we use its default options). The evaluations are conducted on a 2.80GHz Intel(R) Core (TM)2 PC with 3GB physical memory, running Ubuntu 9.04.

Table 1: Statement coverage (%) on Berkeley DB (LOC: lines of code, JCr: JCrasher, Rand: Randoop, ART: ARTGen, Rec: RecGen)

Package	#LOC	JCr	Rand	ART	Rec
com.sleepycat.je	4755	9.8	36.6	32.5	**44.3**
com.sleepycat.je.cleaner	2850	1.6	30.6	8.5	**52.8**
com.sleepycat.je.config	764	89.1	**95.9**	95.5	95.2
com.sleepycat.je.dbi	4401	10.4	40.0	27.9	**53.4**
com.sleepycat.je.evictor	456	0.0	**11.2**	0.2	8.6
com.sleepycat.je.incomp	318	0.3	**23.3**	0.3	16.0
com.sleepycat.je.jca.ra	278	0.0	0.0	0.0	0.0
com.sleepycat.je.jmx	441	49.2	58.3	57.8	**64.6**
com.sleepycat.je.latch	215	27.0	74.9	67.4	**76.7**
com.sleepycat.je.log	3789	9.6	36.3	15.1	**49.6**
com.sleepycat.je.log.entry	366	15.0	47.5	29.8	**65.6**
com.sleepycat.je.recovery	1954	7.0	33.9	7.8	**34.4**
com.sleepycat.je.tree	4398	9.3	34.8	22.0	**47.4**
com.sleepycat.je.txn	2608	6.6	37.6	22.1	**52.5**
com.sleepycat.je.util	1564	5.9	22.9	22.5	**34.6**
com.sleepycat.je.utilint	678	19.3	63.7	50.7	**64.5**
Total	29835	11.0	37.4	24.2	**48.4**

Table 2: Statement coverage (%) on JDSL (LOC: lines of code, JCr: JCrasher, Rand: Randoop, ART: ARTGen, Rec: RecGen)

Package	#LOC	JCr	Rand	ART	Rec
jdsl.core.algo.sorts	91	24.2	**48.4**	24.2	**48.4**
jdsl.core.algo.traversals	26	0.0	0.0	0.0	0.0
jdsl.core.api	62	69.4	**93.5**	90.3	25.8
jdsl.core.ref	2497	26.1	49.4	39.4	**67.4**
jdsl.core.util	60	**30.0**	6.7	6.7	1.7
jdsl.graph.algo	602	8.7	40.0	20.1	**41.4**
jdsl.graph.api	46	47.8	**89.1**	82.6	37.0
jdsl.graph.ref	541	15.7	29.6	25.9	**51.9**
Total	3925	23.2	45.5	35.2	**58.9**

We use statement coverage to measure the effectiveness of our approach, because tests that achieve high structural coverage often have a higher chance to reveal more bugs and improve the software reliability [6]. We apply a Java code coverage tool EclEmma [1] to collect the statement coverage of executing the generated tests.

3.2 Results

Tables 1, 2, and 3 show the statement coverage results of RecGen and other approaches on Berkeley DB, JDSL, and JScience, respectively. The counted lines of code include only lines that are not comments, blanks, standalone braces, or parenthesis.

The results show that JCrasher achieves low statement coverage. It is because JCrasher excludes void-returning methods, e.g., the `DatabaseConfig.setAllowCreate` method, to reduce the search space. But void-returning methods, which may mutate objects, are critical in generating desired inputs. Randoop improves the statement coverage much over JCrasher by using execution feedback to prune illegal and duplicate sequences. ARTGen performs comparably with Randoop on JScience, but worse than Randoop on Berkeley DB and JDSL. The poor performance of ARTGen

Table 3: Statement coverage (%) on JScience (LOC: lines of code, JCr: JCrasher, Rand: Randoop, ART: ARTGen, Rec: RecGen; GEO.COOR: geography.coordinates, MATH: mathematics)

Package	#LOC	JCr	Rand	ART	Rec
org.jscience.	396	3.0	4.5	**4.8**	**4.8**
org.jscience.economics.money	55	43.6	87.3	85.5	**96.4**
org.jscience.GEO.COOR	667	17.4	**61.9**	60.9	21.9
org.jscience.GEO.COOR.crs	198	52.5	**64.1**	61.6	61.1
org.jscience.MATH.function	692	32.8	32.7	37.3	**39.6**
org.jscience.MATH.number	1683	68.1	83.1	79.3	**86.1**
org.jscience.MATH.vector	1551	22.0	39.8	46.1	**82.8**
org.jscience.physics.amount	614	36.5	67.4	57.8	**70.5**
org.jscience.physics.model	60	58.3	96.7	96.7	**100**
Total	5916	37.7	56.1	56.0	**64.9**

may be caused by non-rigorous weights associated with object fields. RecGen improves the statement coverage much over Randoop. By focusing on *relevant methods*, RecGen improves the chance of generating desired sequences for each attempt and further improves the chance by covering all *relevant methods*. For some packages, RecGen achieves lower code coverage than Randoop. For example, the package `jdsl.core.api` of JDSL consists of mainly simple classes that are subclasses of `java.lang.Exception`. RecGen filters out the methods of these classes by default, and thus achieves lower code coverage for the package.

4. RELATED WORK

We have described the main sequence generation approaches in Section 1. Thummalapenta et al. [14] also proposed an approach named MSeqGen to generate desired sequences for a MUT using client code. But client code may not be available in testing code under development. Alternatively, RecGen assists random test generation by recommending sequences based on object-field-access information of the methods in the application under test, without requiring any client code.

Besides sequence generation, primitive argument generation is another key component of test generation. Random approaches [7, 8, 10, 12] generate primitive arguments from all possible values or a set of predefined values randomly. Symbolic execution approaches [9, 13, 15, 16, 18] generate primitive arguments systematically to cover all feasible paths in the MUTs. The symbolic execution approaches execute method sequences with symbolic parameters (unspecified arguments), builds path constraints on the parameters, and solves the constraints to create actual test inputs with concrete arguments. For large or complex programs, it is computationally intractable to precisely maintain and solve the constraints required for test generation. Dynamic symbolic execution approaches [13, 15] address this issue by substituting the symbolic parameters in the constraints with random concrete values. RecGen generates primitive arguments randomly, but it is possible to combine the sequence recommendation approach of RecGen with the symbolic execution approaches.

Our approach is inspired by an implementation-based API recommendation approach named Altair [11]. Altair recommends related methods for a given method in C programs according to the methods' shared data. Our approach extends Altair to recommend sequences for test generation.

5. CONCLUSIONS

In this paper, we propose a MUT-aware sequence recommendation approach, called RecGen, to improve the effectiveness of random object-oriented unit-test generation. The main idea of RecGen is to recommend short sequences that mutate object fields accessed by a MUT to generate the inputs. We have implemented RecGen in Java. Evaluation results show that RecGen improves code coverage over previous random testing tools. We plan to combine RecGen with test selection approaches [19] and symbolic execution approaches in future work.

6. ACKNOWLEDGMENTS

We would like to thank Linchun Sun, Xi Wang, Xin Xin, and Jackie Zhu for their useful feedback. The work described in this paper was fully supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4154/09E), and supported in part by USA NSF grants CCF-0725190, CCF-0845272, CCF-0915400, and CNS-0958235.

7. REFERENCES

[1] EclEmma: Java code coverage for Eclipse. `http://www.eclemma.org/`.
[2] Eclipse Java development tools (JDT). `http://www.eclipse.org/jdt/index.php`.
[3] JDSL: the data structures library in Java. `http://www.jdsl.org/`.
[4] JScience: Java tools and libraries for the advancement of sciences. `http://jscience.org/`.
[5] Oracle Berkeley DB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.
[6] M. Chen, M. R. Lyu, and E. Wong. Effect of code coverage on software reliability measurement. *IEEE Trans. on Reliability*, 50(2):165–170, 2001.
[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *ICSE*, pages 71–80, 2008.
[8] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw., Pract. Exper.*, 34(11):1025–1050, 2004.
[9] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
[10] Y. Lin, X. Tang, Y. Chen, and J. Zhao. A divergence-oriented approach to adaptive random testing of Java programs. In *ASE*, pages 221–232, 2009.
[11] F. Long, X. Wang, and Y. Cai. API hyperlinking via structural overlap. In *ESEC/SIGSOFT FSE*, pages 203–212, 2009.
[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.
[13] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.
[14] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: object-oriented unit-test generation via mining source code. In *ESEC/SIGSOFT FSE*, pages 193–202, 2009.
[15] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP*, pages 134–153, 2008.
[16] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java Pathfinder. In *ISSTA*, pages 97–107, 2004.
[17] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE*, pages 196–205, 2004.
[18] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.
[19] W. Zheng, M. R. Lyu, and T. Xie. Test selection for result inspection via mining predicate rules. In *ICSE Companion*, pages 219–222, 2009.