Systematic Extraction of Tests from Object-Oriented Programs

Mohammad Ghoreshi, Hassan Haghighi

Systematic Extraction of Tests from Object-oriented Programs

Mohammad Ghoreshi[0000−0002−6902−9405] and Hassan Haghighi[0000−0002−6145−4095]

Shahid Beheshti University, Tehran, Iran
{m_ghoreshi,h_haghighi}@sbu.ac.ir

Abstract. Existing program-based automated test techniques from object-oriented programs generate only test data or test cases, which are not equipped with effective oracle to reveal the logical errors in the program. In addition, these techniques often focus on conventional code coverage criteria and intra-method testing, and are less concerned with inter-method, intra-class and inter-class testing. In this paper, we propose an automated testing approach to cover the inter-method and intra-class test levels. This approach generates tests that are equipped with effective oracles in terms of expected outcomes to reveal logical errors in the program under test. In order to demonstrate the applicability of the proposed approach, we applied it to a case study containing 14 different classes implemented in Java. Furthermore, we created artificial faulty versions of our case study, and the proposed approach was able to extract tests that reveal failures in 74% of faulty cases

Keywords: Object-oriented Testing · Automated Test Generation · Intra-class Test Level

1 Introduction

The object-oriented methodology can make finding software faults difficult because it hides the state of objects from each other and increases the complexity of relationships between program elements [1, 2]. Therefore, due to considerable effort and cost in the test phase, we need to use effective automated or semi-automated testing techniques for object-oriented programs.

Various automated and semi-automated techniques, which have been proposed for object-oriented testing, fall into two categories: specification-based and program-based techniques [3]. In specification-based approaches, tests are extracted based on formal and semi-formal specifications of software systems. These methods are extract effective tests along with sufficient oracles in terms of expected outcomes that can reveal errors of the software under test [2, 3]. However, in practice, they are not supported by mature automated tools. In addition, formal specifications are rarely used in industrial software development processes, except in the development of critical systems.

In program-based approaches, test cases are designed according to the program source code. Most of program-based techniques can be categorized in two groups: random-based and structure-oriented techniques. Approaches in the first group generate random method sequences for a class and evaluate each sequence to exhibit illegal behavior. The second group consists of methods which try to find a set of test inputs in order to reach high structural code coverage. Due to lack of the class and program specification, in both kinds of approaches, some method sequences are not based on the class logic; therefore, some of the designed tests have invalid method sequences. Also, lack of specification leads to tests that are not well equipped with effective oracles which reveal bugs and identify correctness or incorrectness of the test result.

One resource that can compensate for the lack of the class specification in program-based approaches is statements used in the code for purposes such as validating class fields and method parameters [4]. These statements can be in form of annotation libraries or assert statements. In this article, we present a program-based approach for testing Java programs. We use validation annotations in order to guide the test process and provide oracles in terms of some kind of assertions which reveal failures in the class under test. Unlike structure-oriented approaches, the suggested approach focuses on testing classes and extracts tests with respect to the inter-method and intra-class levels.

In the next section, we briefly review some considerable related approaches. Then, in section 3, we introduce our approach. Section 4 contains the details of approach evaluation. Finally, section 5 concludes the paper.

2 Related Works

Two approaches were introduced in [5] and [2] to extract tests from Object-Z formal specifications. The former only extracts intra-class level tests, but, in addition to intra-class tests, the later extracts some inter-class level tests and reuses some test artifacts through inheritance. Unlike most specification-based approaches (including approaches in [2, 5]) that suffer from lack of fully automated tools, a framework was introduced in [6] for automated testing of Java programs based on JML specifications. This approach, however, only generates method level tests and does not cover inter-method and inter-class tests.

In [7], a random program-based technique, called JCrasher, was proposed to generate tests from Java programs. This technique generates random method sequences with the aim of causing runtime errors. Such random tests may not necessarily indicate errors in the program under test because they may be illegal themselves. Therefore, as the authors have suggested, this approach is more suitable for robustness testing. Another random approach using the feedback directed technique was presented in [8]. In this approach, unit tests in the format of JUnit are randomly generated for input classes. The approach has been implemented as a fully automated tool called Randoop. Because of the ease of use and scalability, this tool is known as a mature tool in the automated test case generation area, and used as a baseline for evaluating other approaches.

A structure-oriented approach, called "EvoSuite", was proposed in [9] using evolutionary algorithms in order to generate test data for programs in Java. The usability of this approach on large libraries and industrial applications has been demonstrated. It should be noted that, the focus of this approach is to generate intra-method tests that are generated in respect of structural coverage criteria (such as branch coverage), and other object-oriented test levels like intra-class and inter-class are less addressed.

In practice, usually there are no formal specifications of the classes and modules of an object-oriented software except in safety-critical software; hence, specification-based approaches are rarely used. On the other hand, in most program-based approaches, relationships between methods in the method call sequences are usually formed based on a random approach. Therefore, these sequences may not be consistent with the class and program logic and can lead to invalid tests. Moreover, due to lake of specification of the program under test, most program-based approaches are rarely equipped with effective oracles.

In the next section, we introduce an approach for testing object-oriented programs written in Java, which focuses on extracting different method call sequences consistent with the logic of a given class. This approach also covers inter-method and intra-class test levels.

3 The Proposed Approach

The proposed approach starts with a class in Java called the "class under test". First, the state variables (class fields) and related constraints and conditions are extracted from the class under test. The "Model Extractor Component" handles the extraction of class state variables as well as preconditions and postconditions of methods. Now based on this logical model and the class under test, the "Test Machine Component" extracts a test machine for that class. The test machine is a kind of state diagram whose states contain the class state space, and transitions correspond to class method calls. Finally, the "Test Case Generator Component" extracts test cases by mapping the paths of the extracted state machine to JUnit test units.

3.1 Model Extractor Component

The "Model Extractor Component" extracts a model contains the class state variables along with their constraints, as well as preconditions, and postconditions of each method, which are extracted in the form of logical Java expressions. These information can be gathered from several sources in the Java programming language. Some of these sources are annotation-based validation libraries and design by contract libraries in Java. This article uses the OVal annotation library [10], but the model component can be expanded to use other types of libraries, also in addition to the assert statement which is a built-in feature in Java. As an example, a simple stack class in Java with OVal annotations is demonstrated in the left side of the Fig. 1.

3.2 Test Machine Component

A test machine is a test artifact containing abstract information that can be used to extract test cases. Each test machine can be defined corresponding to one method, multiple methods, or a class. In the following, we first present the test machine concept, and then, we review its extraction process.

The Test Machine Concept A test machine is a type of state diagram that consists of several states and several transitions. Each state contains a set of class variables with conditions associated with them. Test machine states make it possible to display the entire state of a class in several abstract states. For example, if x is a class variable, a state of the test machine can be represented as $(x, x \geq 10)$, which represents all states of the class where the value of the variable x is greater than 10. Transitions in a test machine connect states together. A transition in a test machine means calling one of the class methods. So by calling a method, we transfer from one state to another. In a test machine, transitions are divided into two categories: A "valid transition" which is consistent with the logical model (i.e., the postconditions of the called method); and an "Invalid transition which is not consistent with the logical model, or in other words, the postconditions of the called method does not hold in the destination state.

The Method Test Machine In a "method test machine", states are defined on class variables (class fields) that are used in the body of the method. These variables are called effective variables. In addition, in a "method test machine", all transitions represent a call to the method for which the "method test machine" is being defined. The following steps describe how to extract a "method test machine" for a method in a class.

1. State variables partitioning: First, the input space of the effective variables are partitioned. To do this, first the type of each variable and the allowed values are queried from the logical model. Now, based on the type of the variable and its allowable values, partitions are created based on "input space partitioning strategies" [3]. For example, for a numerical variable x with allowable values $x \geq 0$, three partitions $x=0$, $x=1$ and $x>1$ are created according to the "boundary values analysis" [3].
2. Constructing a pool of random objects: In this step, a pool of random objects with the type of the class under test is built. In fact, the object pool is a finite set of objects with different values. This set can be created by different techniques that generate random objects from a class. In the proposed approach, a method inspired by the Randoop approach is used to construct random objects of the class under test.
3. Extracting the test machine: We consider all extracted partitions as test machine states. Now for each state as "input state", we do as follow:
   (a) Sample objects that are placed in the input state are selected from the objects pool.

(b) For each sample object, the method for which we are extracting the test machine is performed. Now, the state of the sample object is mapped to one of the test machine states called "destination state". Therefore, we put a transition in the test machine between the "input state" and the "output partition".

Merge Test Machines By merging two method test machines, a test machine can be obtained that produces different sequences of two method calls. Assuming two test machines A and B (corresponding to methods A and B), the following steps should be performed to combine these two test machines.

1. First, a copy of test machine A is considered as the resulting test machine.
2. For each state (called "candidate state") of the second machine (machine B), we perform the following steps:
   (a) If the candidate state is the same as one of the states in B: we ignore this state and copy its related transitions to the equivalent state in A.
   (b) If the candidate state does not have any overlap with states of machine B: we add the candidate state with its related transitions to machine A.
   (c) If the candidate state has overlap with some states of machine B: Three new states are formed based on the notion of "disjunctive normal form" and replaced the candidate state (say S1) and that state of machine B (say S2) with which the candidate state has overlap; see equation (1). Each conjunction in equation (1), like $(S1 \wedge \neg S2)$, indicates a new state in the resulting test machine. It is clear that the new states have no overlap with each other, and also, are logical equivalent to the two old states, i.e., S1 and S2.

$$S1 \vee S2 = (S1 \wedge \neg S2) \vee (S1 \wedge S2) \vee (\neg S1 \wedge S2) \quad (1)$$

The Class Test Machine After combining two method test machines, the resulting test machine can be merged with the third method test machine in a similar way. By applying this "incremental and iterative" process to other methods, we get the whole class test machine. As an example, for mentioned stack class, by merging test machines for methods push and pop, a test machine for the class stack can be created as shown in the right side of the Fig. 1.

3.3 Test Case Generator Component

Different test cases can be extracted from an extracted test machine. In general, each path in the test machine, as a set of method calls of the class under test, can be a test case. Test cases fall into two general categories:

- Error revealing test cases: In these test cases, there must be an "invalid transition", and performing this test will reveal a logical error in the program.
- Regression test cases: In these test cases, there are no "invalid transitions" in the corresponding test machine path. Tests of this kind contain a set of correct class interactions and method interactions that lead to a (functionally) correct state. Such test cases can be used for regression testing.

```
01 public abstract class Stack
02 {
03   @Assert(expr = "_value>=0")
04   private int size;
05
06   @Post(expr = "top()!=null")
07   @Post(expr = "_this.size==_old+1",old="_this.size")
08   public void push(Object x)
09
10   @Pre(expr = "this.size>0")
11   @Post(expr = "_this.size==_old-1",old="_this.size")
12   public Object pop()
13 }
```

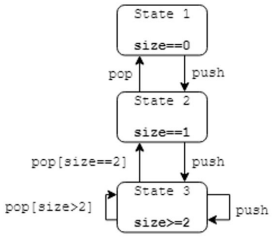

Fig. 1. A stack class with OVal annotations alongside with the extracted test machine.

4 Evaluation

In this section, our proposed approach is applied to a case study in order to demonstrate its applicability. The case study subject is a Java implementation of a famous puzzle game, called Tetris [11]. This implementation contains 14 classes. Using our approach, a test machine was extracted for each class. For comparison, our approach is compared to Randoop, which is a mature tool in the field of automated random test generation.

4.1 Effectiveness of Tests in Revealing Errors

To evaluate the effectiveness of tests in revealing errors, we first created 25 mutants versions of the Tetris program using the Pit tool [12]. Then, in order to generate automated tests based on each mutant (a mutant class alongside with other Tetris classes), it has been given as input to both the proposed approach and Randoop. The Randoop tool has only been able to generate error revealing test cases for 2 of 25 mutants. Our approach generated error revealing test cases for 18 mutants, i.e. 72% of the cases.

4.2 Revealing Real Error

There is a real bug in one of the source code versions of the Tetris game [11], which has been fixed in future versions. This bug is that, in certain circumstances, some blocks do not stick to the right or left wall of the game screen, and although the user wants to move the block one unit to the right or left, a gap between the block and the wall remains. Randoop could not generate error revealing tests for this version. By running our approach on this version, however, it generated different test cases that reveal failures of this type.

5 Conclusion

In this paper, we presented a new approach to generate test cases for object-oriented programs. Our approach leads the test case generation process by extracting some facts and conditions about the program under test. The approach

produces effective test cases for revealing program errors. To demonstrate the approach usability, it has been applied to the Tetris game, as our case study, containing 14 different classes. The proposed approach managed to detect a significant number of errors that had been seeded into the Tetris source code.

We are currently expanding our approach to support all popular validation libraries and built-in assert statements in Java. By this, our approach can be applicable for more Java programs, and thus, we can evaluate it on more case studies including industrial and open source software. As another direction for future work, by deriving relationships between test machines of classes, inter-class test cases can be extracted that examine relationships between classes.

References

1. Alexander, R. T., Offutt, A. J.: Criteria for testing polymorphic relationships. In: Proceedings 11th International Symposium on Software Reliability Engineering, pp. 15-23. IEEE (2000)
2. Ghoreshi M., Haghighi H.: An incremental method for extracting tests from object-oriented specification. Information and Software Technology (78), 1-26 (2016)
3. Ammann, P., Offutt, J: Introduction to software testing. Cambridge University Press (2016)
4. Shamshiri S., Just R., Rojas J.M., Fraser G., McMinn P., Arcuri A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 201-211 (2015)
5. Carrington, D., MacColl, I., McDonald, J., Murray, L., Strooper, P.: From Object-Z specifications to ClassBench test suites. Software Testing, Verification and Reliability 10(2), 111-137 (2000)
6. Milicevic, A., Misailovic, S., Marinov, D., Khurshid, S.: Korat: A tool for generating structurally complex test inputs. In: 29th International Conference on Software Engineering (ICSE'07), pp. 771-774. IEEE (2007)
7. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Software: Practice and Experience 34(11), 1025-1050. (2004)
8. Pacheco C, Lahiri SK, Ernst MD, Ball T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering, pp. 75-84. (2007)
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276-291 (2012)
10. Oval: The Object Validation Framework for Java, https://sebthom.github.io/oval/, last accessed 2020/11/15
11. JTetris Github Repository, https://github.com/sbu-test-lab/jtetris, last accessed 2020/11/15
12. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: Pit: a practical mutation testing tool for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 449-452. (2016)