Test Case Generation for Classes in Objects-Oriented Programming Using Grammatical Evolution

Jirawat Chaiareerat, Peraphon Sophatsathit and Chidchanok Lursinsap

Abstract This paper proposes a dynamic test case generation approach for Object-Oriented Programming classes, using evolutionary search to find test cases that would satisfy a branch coverage criteria. Grammatical Evolution (GE) is used to search for a solution in accordance to user-specified grammar, thus making the algorithm more flexible than the traditional genetic programming. Rather than generating test cases directly, source code for an Intermediate Test Script (ITS) is generated from the grammar. It is then evaluated and translated into source code by ITS interpreter. Such a provision makes it easy to produce test cases that have object and literal reference, whereby improve the performance of GE. We've tested the proposed method with several java classes from open source projects and yielded high code coverage results.

Keywords Test case generation · Code coverage · Object-oriented programming · Grammatical evolution · Intermediate test script

J. Chaiareerat (✉) · P. Sophatsathit · C. Lursinsap
Department of Mathematics and Computer Science, Faculty of Science,
Advanced Virtual and Intelligent Computing (AVIC) Center,
Chulalongkorn University, Bangkok, 10330 Thailand
e-mail: jirwat.c@student.chula.ac.th

P. Sophatsathit
e-mail: peraphon.s@chula.ac.th

C. Lursinsap
e-mail: lchidcha@chula.ac.th

1 Introduction

Test case generation is inherently a vital part of software testing. Because testing all possible test data is infeasible, test case generation therefore uses different criteria to select and generate only a subset of test data that guarantees a high quality of the test set. These criteria are called code coverage.

Previous works in test case generation can be classified into two groups, namely, static and dynamic approaches. The static test case generation relies on static analysis of the program source code. They usually apply the same concept as procedural test case generation by using symbolic execution and constrained solving. On the other hand, the dynamic approach models the test case generation as a search problem to find an optimal solution, which gives the highest code coverage. This approach executes the test object iteratively by tuning the test case until a satisfied result is reached. Various optimization techniques are used in dynamic test case generation including Genetic Algorithm [1–3], Simulated Annealing [2, 3], Memetic Algorithm [2, 3] and Strongly-Typed Genetic Programming [4, 5].

This paper proposes a dynamic approach test case generation for an object using GE to find test cases that would satisfy branch coverage criteria. Each test case is translated into Intermediate Test Script (ITS) format based on user-specified grammar. The idea is to find an executable program or program fragment that will yield a good fitness value for the given objective function.

2 Test Case Generation Framework

The Test Case Generation Framework consists of 4 modules: Code Instrumentation, Grammar Generator, Search, and Evaluation Module. Referring to Fig. 1, Java source code is instrumented by Code Instrumentation module. This process will insert certain code to the original source code which will be used for collecting branch coverage information during the evaluation of each test case. The instrumented source code is then compiled into a Java class file by java compiler. This compiled Java class will later be fed as an input to the Evaluation Module. The Grammar Generator Module, as stated by its name, will automatically generate grammar for test cases. At this step, users can review the generated grammar and modify it according to their specific grammar. The Search Module then derives these test cases through the defined grammar based on genetic algorithm. This process transforms the test cases into ITS which later are evaluated by the Evaluation Module.

The Evaluation Module processes the compiled Java class, together with Intermediate Test Script using its interpreter, to produce branch coverage information. The resulting branch coverage information is then fed back to the Search Module for selecting the best test case in each generation.

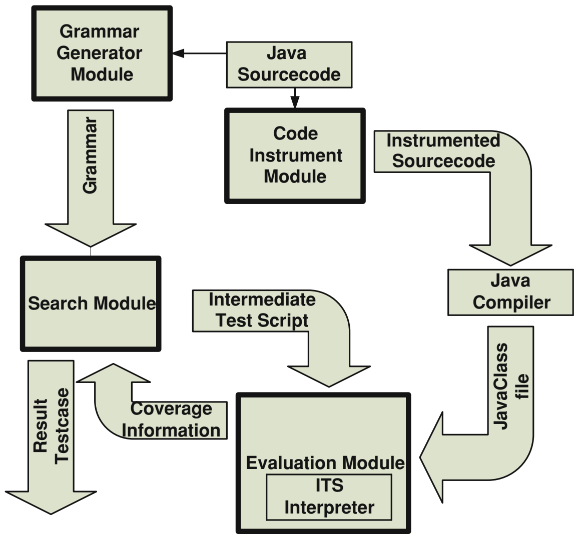

Fig. 1 Test case generation framework

3 Code Instrumentation

Code Instrumentation is the process of inserting certain code to the original source code, wherever there's a condition check and method declaration. The objective of code insertion is to let the newly inserted code collect information on path and coverage each time they are executed. Apart from collecting the mentioned information, they are also used for calculating branch distance. Branch distance information is used to evaluate how much each branch is closed to being taken. We used Java Parser for parsing and inserting new code to the original java source code automatically.

In a situation that two test cases have the same degree of coverage, branch distance is used to determine which one is a better and suitable for potential test case. The value of branch distance represents how much a branch is close to being taken. In this paper, branch distance is calculated, using the formula shown in Table 1. The lower branch distance means the branch is closer to be covered.

From Table 1, dist(x) is the branch distance of condition x and k is the smallest possible value of the branch distance (we used 0.0001 in this paper). The branch distance in the above table is then normalized by the following equation

$$\mathrm{dist}(A)_{\mathrm{normalize}} = 1 - (1 + 0.1)^{0-\mathrm{dist}(A)}. \tag{1}$$

Table 1 Branch distance calculation

Operation	Branch distance
dist(A == B)	Abs(A − B)
Dist(A! = B)	k
Dist(A >= B)	B − A
Dist(A > B)	B − A + k
Dist(A <= B)	A − B
Dist(A < B)	A − B + k
Dist(A and B)	Max[dist(A), dist(B)]
Dist(A or B)	min[dist(A), dist(B)]

4 Intermediate Test Script and Grammar

In this proposed method, each test case is represented in ITS Format. Grammar for the ITS of a class under test is automatically generated from the source code by the Grammar Generator.

The proposed framework generates test cases in Intermediate Test Script (ITS) format instead of a java source code. In ITS, each parameter of the method call can be referred to the previous created object or literal value. The advantage of using the same parameter helps improve the search process. Moreover, ITS also reduces the time for compiling the source code, since it can be run directly by ITS interpreter.

Grammar of ITS for the class under test is generated from a java source code. Tester can modified the grammar by inserting some heuristics, which can help reduce the search space, making it easier for the Search Module to discover the optimum solution. Using grammar to represent test cases have an advantage in flexibility since various types of parameter such as array, string and object reference can be created by grammar. In this paper, we used grammar in Backus-Naur Form (BNF) as an input of GE.

5 Search and Evaluation

Grammatical Evolution performs the process of searching for a test case that has the best code coverage for the class under test. In this paper, we used GEVA [6] for our experiment.

Traditionally, Genetic Algorithm (GA) is used to find the optimal solutions to a search problem. Genetic algorithm is classified as global search heuristics. Genetic Programming (GP) is for finding computer programs that satisfy user-defined criteria. Grammatical Evolution (GE) [7, 8], on the other hand, is an evolutionary computation technique that can be considered as a grammar-based GP. A program is represented by an ordered-list of integer. GE uses genotype-phenotype mapping process to map an ordered-list of integer to a computer program. With the help of

Table 2 Experimenta result (GE)

Class name	Total branches	Achievable branches	Branch coverage (mean)	Branch coverage (%)
Stack	10	10	10.00	100
StringTokenizer	40	39	39.00	97.50
Vector	128	123	123.00	96.09
LinkedList	130	122	122.00	93.84
BinTree	37	37	37.00	100.00
BinomialHeap	87	77	74.67	85.82
BrentSolver	29	28	28.00	96.55
SecantSolver	19	19	18.86	99.26
Complex	54	52	51.80	95.92

grammar, GE also provides a very flexible way to control an algorithm. The user can define a grammar that biases to produce very specific form of program, or can incorporate domain knowledge of the problem into the underlying grammar.

GE, GA, and GP use crossover and mutation operations to modify each individual and reproduce new populations. In this paper, a standard single point crossover; and nodal mutation [9] is used since it gives the best result among the three in our experiment. Fitness function will be calculated based on coverage information collected during the execution of each test case in the population. The following equation is used in the calculation:

$$\mathrm{fitness(t)} = (\mathrm{b} - \mathrm{cov}) + 1 - (1.1)^{0-\mathrm{bd}} \tag{2}$$

where b is the total number of branches, cov is the total branch coverage of test case t, and bd is the total branch distance.

6 Experimental Results

We used selected java class for our experiment including J2SDK version 1.4.2_12, Java Path Finder [10] version 1.3r1258 and Apache Commons Math version 1.1.

The experimental results are shown in Table 2 which compares the proposed algorithm with Acuri [3] and Wappler [5] in Tables 3 and 4 respectively. In the proposed algorithm, GE is executed for 200 generations with the number of population equals to 50. The probability of crossover and mutation of GE is configured to 0.9 and 0.1, respectively. Average performance execution of the algorithm is based on 50 runs. Branch Coverage also includes the number of method calls and try/catch statements. The branch coverage percentage is the ratio of mean branch coverage over total branch. The total branches were not stated in Acuri [3], while the achievable branches and branch coverage were not stated in Wappler [5].

Table 3 Experimenta result (Acuri's memetic method [3])

Class name	Total branches	Achievable branches	Branch coverage (mean)	Branch coverage (%)
Stack	–	10	10.00	–
StringTokenizer	–	–	–	–
Vector	–	100	100.00	–
LinkedList	–	84	84.00	–
BinTree	–	37	37.00	–
BinomialHeap	–	79	77.66	–
BrentSolver	–	–	–	–
SecantSolver	–	–	–	–
Complex	–	–	–	–

Table 4 Experimenta result (Wappler's EvoUnit [5])

Class name	Total branches	Achievable branches	Branch coverage (mean)	Branch coverage (%)
Stack	8	–	–	100
StringTokenizer	29	–	–	93.70
Vector	–	–	–	–
LinkedList	68	–	–	98.30
BinTree	–	–	–	–
BinomialHeap	–	–	–	–
BrentSolver	27	–	–	96.30
SecantSolver	17	–	–	100.00
Complex	51	–	–	90.20

The result shows that our algorithm is capable of generating almost all types of test cases by virtue of grammar which was evident by the high coverage outcome.

7 Conclusion

We propose a grammar-based test case generation for a class in Object-Oriented by using Grammatical Evolution and Intermediate Test Script. This method also supports various features of Object-Oriented Programming through the grammar. The results confirm that the proposed method can generate high coverage test cases. Moreover, being grammar-based, this technique has great flexibility and advantages in that it is possible to incorporate heuristics and characteristic of the class under test into the grammar to reduce the search space of the problem.

References

1. Tonella P (2004) Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and analysis (ISSTA'04), ACM, New York, pp 119–128
2. Acuri A, Yao X (2007) A memetic algorithm for test data generation of object-oriented software. In: IEEE congress on evolutionary computation (CEC 2007), pp 2048–2055, IEEE
3. Acuri A, Yao X (2008) Search based software testing of object-oriented containers: information sciences, vol 178, issue 15, pp 3075–3095, Elsevier Science, New York
4. Wappler S, Wegener J (2006) Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Proceedings of the 2006 conference on genetic and evolutionary computation (GECCO'06), ACM, New York, pp 1925–1932
5. Wappler S (2008) Automatic generation of object-oriented unit tests using genetic programming. PhD thesis, Technical University of Berlin
6. O'Neill M, Hemberg E, Gilligan C, Bartley E, McDermott J, Brabazon A (2008) GEVA: grammatical evolution in java. SIGEVOlution 3(2):17–22 ACM New York
7. O'Neill M, Ryan C (2001) Grammatical evolution. IEEE Trans Evol Comput 5(4):349–358
8. O'Neill M, Ryan C, Keijzer M, Cattolico M (2003) Crossover in grammatical evolution. Genetic programming and evolvable machines, vol 4, issue 1, pp 67–03, Academic Publishers, Kluwer
9. Byrne J, O'Neill M, McDermott J, Brabazon A (2009) Structural and nodal mutation in grammatical evolution. In: Proceedings of the 11th annual conference on genetic and evolutionary computation (GECCO), ACM New York, pp 1881–1882
10. Visser W, Pasareanu CS, Khurshid S (2004) Test input generation with java pathFinder. In: Proceedings of 2004 ACM SIGSOFT international symposium on software testing and analysis (ISSTA'04), ACM New York, pp 97–107
11. Buy U, Orso A, Pezze M (2000) Automated testing of classes. In: Proceedings of 2000 ACM SIGSOFT international symposium on software testing and analysis (ISSTA 2000), ACM New York, pp 39–48
12. Sen K, Agha G (2006) CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: 18th international conference on computer aided verification (CAV'06), LNCS vol 4144, pp 419–423, Springer, Berlin
13. Sen K, Marinov D, Agha G, (2005) CUTE: a concolic unit testing engine for C. In: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM New York, pp 263–272
14. Xie T, Marinov D, Shulte W, Notkin D (2005) Symstra: a framework for generating object-oriented unit tests using symbolic execution. In: Proceedings of the 11th international conference on tools and algorithms for the construction and analysis of systems (TACAS 05), LNCS vol 3440, pp 365–381, Springer, Berlin