A Method Dependence Relations Guided Genetic Algorithm

Ali Aburas and Alex Groce

Oregon State University, Corvallis OR 97330, USA

Abstract. Search based test generation approaches have already been shown to be effective for generating test data that achieves high code coverage for object-oriented programs. In this paper, we present a new search-based approach, called GAMDR, that uses a genetic algorithm (GA) to generate test data. GAMDR exploits method dependence relations (MDR) to narrow down the search space and direct mutation operators to the most beneficial regions for achieving high branch coverage. We compared GAMDR's effectiveness with random testing, EvoSuite, and a simple GA. The tests generated by GAMDR achieved higher branch coverage.

Keywords: SBST, Genetic Algorithm, Search Space Reduction, Java Testing

1 Introduction

Different search-based testing techniques have been proposed to automatically generate unit tests for object-oriented programs, e.g., *TestFul* [4] and *EvoSuite* [7]. A major problem with most of the existing search based software testing (SBST) approaches is that they consider the whole search space of possible input values and method calls to the class under test (CUT). Thus, finding critical calls can be a challenge due to the large size of the search space.

In genetic algorithms (GA), the mutation operator plays an essential role: it modifies individuals (here, test cases) with a relatively small probability. Mutation operations (e.g., modifying input values or inserting method calls) are randomly performed to preserve diversity of populations, and prevent the search from being trapped in a local optima [11]. Nevertheless, whenever mutation occurs, the chance of choosing the method calls or primitive values that are most beneficial is very low. Such random mutation has two problems. First, it lacks guidance as to inputs, causing unnecessary computational expense [11]. This is due to an inability to explore promising areas in the search space. Second, randomly flipping methods or manipulating an input primitive value may fail to generate high quality new individuals. This can lead to an increase in the chances of premature convergence due to lack of diversity in the population [11].

In this short paper, we introduce a fully automated search-based testing approach for Java, called GAMDR. GAMDR implements a Genetic Algorithm (GA) that aims to cover *all* target branches. This implementation accelerates the

search towards the global optimum because it does not waste time on infeasible branches [4, 7]. GAMDR also exploits Method Dependence Relations (MDR) [14] to narrow down the search space and direct mutation operators to the most beneficial regions in the search space, leading to high CUT branch coverage.

2 Related Work

Harman et al. [9] were the first to theoretically and empirically explore search space reduction for SBST. Their empirical study targeted procedural programs and showed that irrelevant input removal improved the performance of local, global, and hybrid search algorithms. Barsei et al. [4] proposed a semi-automated approach to augment the efficiency and speed-up test generation with the TestFul tool. This was achieved by requiring the user to provide data regarding the effects of each method of the CUT. Ribeiro et al. [12] leveraged purity analysis [13] to reduce the input space of object-oriented programs. Harman et al. [10] also proposed a domain reduction technique to exclude irrelevant parameters in the search space for aspect-oriented programs. They performed backward slicing to identify such irrelevant parameters, after which evolutionary testing was conducted only on the remaining relevant parameters. Aburas and Groce [1] proposed a memetic algorithm exploiting MDR to improve the effectiveness of a hill climbing (HC) technique.

In contrast to the aforementioned approaches, our approach uses GA to generate test data and applies a static analysis to precisely identify only those member fields or parameters of the method under test that would be relevant for covering uncovered branches. Then, it leverages MDR to automatically direct the mutation operations to generate a sequence of method calls that produce the desired values for member fields or parameters, based on impact on target branches. Combining GA with MDR has a number of advantages. **1)** it focuses on the root cause of the failure to cover target branches. **2)** it focuses only on the relevant parts of the individuals (i.e., test cases) that affect the execution of the target branches. **3)** it implements a domain reduction mechanism to speed search space exploration. Unlike previous search-based approaches, these strengths together enable the proposed approach to explore high complexity code in order to achieve high branch coverage.

3 GAMDR

GAMDR consists of three different components: the Instrumenter, Static Analyzer, and Genetic Tester components.

- Instrumenter Component: In this component, the original source code of the class under test (CUT) is instrumented at byte-code level to measure coverage values and calculate the fitness function. We use Soot[1] for analyzing and instrumenting Java byte-code.

[1] http://www.sable.mcgill.ca/

- Static Analyzer Component: The key idea behind our approach is to use lightweight static analysis to identify relevant methods for each target branch, and then use them during mutation operations. To this end, we perform backward analysis for each target branch, and precisely identify if a parameter of the method contains the target branch or if a member field of a class can help to cover the target branch. For each member field, we use MDR to identify the methods that modify the member field (the `write-read relation`). In addition, if a parameter of the target method affects the coverage of the target branch, we identify all the methods that write in the target method (the `read-write relation`). If the identified parameter is not a primitive type, we identify the methods' return as the same type object that can be passed as an argument to the target method (i.e., `accessed-data relation`).

- Genetic Tester Component: In our implementation, we use a similar GA to that used in previous work [3, 4], but extend it to implement MDR [14].

1. **Individual representation**: We use an individual representation similar to some previous work [4, 7] because it is easy to manipulate. Each individual consists of a set of statements of length N, which is set to 80. Each statement is a constructor, method call, field access, or array input.
2. **Fitness Function**: The fitness function uses branch distance (BD) and keeps track of how close an individual is to covering all reachable but not-yet-executed branches [3, 7].

$$f(i) = \sum_{b_j \in B} BD(b_j, i) \qquad \text{and} \qquad BD(b_j, i) = \begin{cases} 0 & \text{if branch } j \text{ is covered} \\ k & \text{if branch } j \text{ is reached} \\ 1 & \text{otherwise} \end{cases}$$

   The function $BD(b_j, i)$ shows how close an individual i is to cover the not-covered branch j. Here BD is all target branches and k is a normalizing function with value within [0,1]; we use the normalization function: $k = \frac{x}{x+1}$ [2], and x shows how far a predicate is from obtaining opposite value [7].
3. **Genetic Operations**: Our approach (GAMDR) implements common genetic operators: selection, crossover, mutation, and elitism, to manipulate and evolve successive populations. Following is a summary of these operators:

a) ***Selection***: GAMDR implements tournament selection [11]. However, if two individuals have the same fitness values, the shortest individual is selected to prevent *bloat* [6].

b) ***Crossover***: GAMDR implements a fixed single crossover point, where the two selected individuals are cut at the middle, to avoid generating long offspring [6].

c) ***Mutation***: After crossover, the individuals are subjected to mutation. Rather than just randomly changing statements of the chosen individuals, GAMDR uses MDR to direct the mutation operator towards relevant statements where changes may help to result in more fit individuals and increase exploration of the search space. Therefore, GAMDR randomly chooses a reached (but not covered) branch and analyzes its predicates. Then, GAMDR precisely identifies the relevant types of elements that are involved in execution of the target branch, e.g. member field, parameter method, or/and constant

values. Consequently, GAMDR directs the mutation operations to explore those identified relevant statements (constructors, methods, and parameters). Finally, for a chosen individual with a length n, GAMDR randomly applies one of the following operations with probability 1/3.

- **Remove**: All irrelevant statements are removed; additionally a chosen statement from the identified relevant statements is removed from the individual with a probability r, where $r = 0.01$.
- **Insert**: A random number r, where $1 \leq r \leq (N - n)$, of identified relevant statements are added at a random position in the chosen individual.
- **Change**: Each identified relevant statement and parameter is changed in the chosen individual with probability r, where $r = 0.01$.

d) ***Elitism***: The best individuals are copied to the next new generation. The population size is set to 100, and elitism rate is set to 10%.

4 Empirical Study

We compared the effectiveness of GAMDR in achieving branch coverage against three different approaches: a simple GA (without MDR enabled) [3, 4], pure random testing (RT) [5], and EvoSuite [7]. We used seven popular Java projects as test subjects (Table 1). These projects are taken from the literature discussing cases where SBST faces problems in achieving high branch coverage.

We used identical configurations for GAMDR and the simple GA to ensure as fair a comparison as possible. We also used EvoSuite version 20130910 with the default configuration. To compare RT with GAMDR, we adopted the proposed approach by Ciupa *et al.* [5]; the length of test cases in RT was set to 200 [8]. We ran each approach 30 times with a time limit of 5 minutes with different random seeds, and used JaCoCoVersion 0.7.5[2] to measure coverage during test generation.

Table 1. Details of the test subjects.

Test Subject	#Classes	NCSS	#Branches
Commons Codec	41	3,269	1,373
Commons CLI	11	677	288
Conzilla	13	377	120
jdom2	40	3,196	978
lang3	55	9,182	5052
NanoXML	26	1,984	571
Joda-Time	57	9,152	2,207
Total	**243**	**27,837**	**10,589**

4.1 Effectiveness of GAMDR

Table 2 summarizes the average branch coverage percentags for the 30 experiments. In the table, the highlighted values with bold text indicates that a particular testing approach obtained the highest coverage (with Mann-Whitney-Wilcoxon test p-value < 0.05) for that test subject. For Commons Codec, GAMDR was significantly better than EvoSuite and the pure GA, but not RT.

[2] http://eclemma.org/jacoco/

Table 2. Branch Coverage achieved at 5 minutes

Test Subject	RT(%)	EvoSuite(%)	GA(%)	GAMDR(%)
Commons Codec	89.71	89.28	87.76	90.47
Commons CLI	95.96	95.67	91.97	95.81
Conzilla	70.05	82.79	73.78	**91.85**
Jdom2	**83.58**	81.22	80.02	83.03
lang3	88.48	78.64	86.98	**89.43**
NanoXML	62.87	61.34	62.51	**69.88**
Joda-Time	79.52	83.19	79.95	**85.10**

Table 2 shows that GAMDR outperforms other test approaches on Conzilla, NanoXML and Joda-Time subjects. One major reason is that these subjects contain classes which have constructors that call superclasses. These constructors require calling methods that are in a correct order and have valid arguments. For example, in the NanoXML subject, the constructor of the class `CDATAReader` requires a valid `StdXMLReader` object, which is a concrete implementation of the interface class `IXMLReader`. As a result, a valid sequence of method calls requires a correct order to create the desired objects: a valid `StdXMLReader` object must be created before a `CDATAReader` object. Despite the fact that the class `CDATAReader` contains only 4 public methods, our experiment revealed that RT, EvoSuite, and GA could only achieve 66%, 68%, and 71% branch coverage of `CDATAReader`, respectively. This is because there is no guidance encoded in the fitness function identifying which constructors, methods, or parameters must be called to cover certain branches. In contrast, the static analysis used in GAMDR helps to identify all relevant methods based on the fields they write, and accessible constructors. For example, GAMDR identifies the `StdXMLReader` constructors and the method `stringReader` because they both return objects that can be used to replace the interface class type argument in the `CDATAReader` constructor, i.e., *accessed-data relation*. In addition, GAMDR identifies the `CDATAReader` constructor because it writes field `reader`, i.e., *write-read relation*. As a result, during the mutation phase, GAMDR tries to generate test data and method calls for these relevant methods and constructors instead of investing time on all constructors, methods and parameters. Our results show that GAMDR achieves 90% branch coverage of the class `CDATAReader`, which is 23%, 22%, and 19% higher than RT, EvoSuite, and GA, respectively.

The results also show GAMDR outperforms EvoSuite and GA on lang3, and improves some over RT, becasue lang3 contains classes that contain a large number of method calls. For example, the `ArrayUtils` class contains 229 different public methods to test, each of which takes primitive and/or array arguments. RT achieves 99%, EvoSuites 68%, and GA 88% branch coverage of the class. We speculate the low branch coverage of the EvoSuite and GA are because the number of public methods decreases the probability of mutations of relevant methods and parameters to cover certain branches. GAMDR uses MDR to increase the probability of useful mutations, and achieves 98% branch coverage.

The results indicate that MDR is indeed useful in helping to increase branch coverage by identifying relevant methods and parameters that need to be mutated in order to cover particular branches. The results also support the belief

that the applicability of the search-based test data generation techniques are limited not only when the search space is large, but also when the search does not take into account data dependencies within the class under test (CUT) [11].

5 Conclusion

This paper has introduced and evaluated GAMDR, which applies a genetic algorithm (GA) to cover all target branches at the same time, and uses method dependence relations (MDR) for improving choice of mutations. Our empirical study shows that GAMDR achieves higher branch coverage than RT, EvoSuite, and a simple GA, for complex hard-to-cover programs.

References

1. Aburas, A., Groce, A.: An improved memetic algorithm with method dependence relations (mamdr). In: Quality Software (QSIC). pp. 11–20. IEEE (2014)
2. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability 23(2), 119–147 (2013)
3. Arcuri, A., Yao, X.: A memetic algorithm for test data generation of object-oriented software. In: Evolutionary Computation, 2007. pp. 2048–2055. IEEE (2007)
4. Baresi, L., Lanzi, P.L., Miraz, M.: Testful: an evolutionary test approach for java. In: Software testing, verification and validation (ICST). pp. 185–194. IEEE (2010)
5. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo. In: Software Engineering, 2008. ICSE'08. pp. 71–80. IEEE (2008)
6. Fraser, G., Arcuri, A.: Handling test length bloat. Software Testing, Verification and Reliability 23(7), 553–582 (2013)
7. Fraser, G., Arcuri, A.: Whole test suite generation. Software Engineering, IEEE Transactions on 39(2), 276–291 (2013)
8. Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., Lopez, C.: Lightweight automated testing with adaptation-based programming. In: Software Reliability Engineering (ISSRE). pp. 161–170. IEEE (2012)
9. Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Wegener, J.: The impact of input domain reduction on search-based test data generation. In: Proceedings of the ACM SIGSOFT symposium. pp. 155–164. ACM (2007)
10. Harman, M., Islam, F., Xie, T., Wappler, S.: Automated test data generation for aspect-oriented programs. In: Proceedings of the Aspect-oriented software development. pp. 185–196. ACM (2009)
11. McMinn, P.: Search-based software test data generation: a survey. Software testing, Verification and reliability 14(2), 105–156 (2004)
12. Ribeiro, J.C.B., Zenha-Rela, M.A., de Vega, F.F.: Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. Information and Software Technology 51, 1534–1548 (2009)
13. Salcianu, A., Rinard, M.: Purity and side effect analysis for java programs. In: Verification, Model Checking, and Abstract Interpretation. vol. 3385, p. 199 (2005)
14. Zhang, S., Saff, D., Bu, Y., Ernst, M.D.: Combined static and dynamic automated test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 353–363. ACM (2011)