Improving Test Data Generation for Java Programs using Search-based Approach and *d*-criterion

Mingyu Jin
School of Computing
KAIST
Daejeon, Republic of Korea
mgjin@se.kaist.ac.kr

Youlim Jung
School of Computing
KAIST
Daejeon, Republic of Korea
yljung@se.kaist.ac.kr

Jiyoung Song
School of Computing
KAIST
Daejeon, Republic of Korea
jysong@se.kaist.ac.kr

***Abstract*—Java program testing professionals have required effective test suites in fault detection within limited budget. To generate more effective test suites, they have evaluated test suites by mutation analysis. Therefore, researchers have been suggested more powerful mutation adequacy criterion. Also, they have studied on automatizing the process of test data generation and reducing the test suite size by prioritization to reduce the cost it takes to test programs. In this paper, we suggest an automated test suite generation approach using genetic algorithm and powerful mutation adequacy criterion. The proposed approach is implemented in the previous tool, *EvoSuite*. The test suite generated by our approach is up to four time more effective than test suites generated by original *EvoSuite*.**

***Keywords*-automated test generation; genetic algorithm; mutation analysis; d-criterion**

I. Introduction

Java program-testing professionals are always concern about the budget and the effectiveness of their test suites in fault detection. It takes relatively low cost to generate test suites thanks to automated test data generation tools such as *EvoSuite* [1]. But, Java program testers still require more powerful test suites within limited budget. To figure out how effective their test suites, the test suites have been evaluated by Mutation analysis: Artificial defects are injected into a target program and the fault-injected programs (mutants) are tested by test suites. To generate the more effective test suites, the more powerful mutation adequacy criterion is needed.

In general mutation adequacy criterion, it simply counts the number of killed mutants which give different outputs against the original program when test suites are executed on the original program and mutants. On the other hand, Shin et al. [2] proposed more powerful mutation adequacy criterion (*d-criterion*: distinguishing mutation adequacy criterion). According to *d-criterion*, mutants can be distinguished from each other by the set of tests that kill them. However, Shin et al.'s approach [2] generates large-sized test suites as they only tried to distinguish mutants but not to reduce the scale of test suites.

In this paper, we proposed an approach that automatically generates an effective test suite for a Java program not increasing the size of the test suite. Our approach uses the genetic algorithm and we utilize *d-criterion* as a fitness function to generate the effective test suite. The size of the test suite is handled by bloat controls which is applied generally to the genetic algorithm. We implemented our approach in *EvoSuite*. We have shown timeout issues, size of the test suite, and effectiveness of the test suite by answering three research questions. The results show that the generated test suite by our approach is up to four times more effective than a test suite generated by original *EvoSuite* not increasing the size of the test suite.

This paper is organized as follows. Section 2 describes the background for understanding the previous works such as the genetic algorithm and the d-criterion for mutation analysis. Section 3 presents our detailed test data generation approach using genetic algorithm and d-criterion. In section 4, we apply our approach to subject programs and conduct experiments. We conclude the paper in section 5.

II. Background

This section presents backgrounds for understanding our approach. The proposed approach used the genetic algorithm for generating a test suite and *d-criterion* is used as a fitness function which is utilized in the genetic algorithm. The illustrations on genetic algorithm and *d-criterion* is given on following subsections.

A. Genetic Algorithm in EvoSuite

EvoSuite [1] generates test data for Java programs using the genetic algorithm. The genetic algorithm is one method that gives optimal solutions by mimicking a natural selection process. The input of *EvoSuite* is a Java program. The genetic algorithm which is implemented in *EvoSuite* has five steps; initial population of test suites; evaluating fitness of population; selecting individuals as parents; creating offspring from parents; mutating offspring.

First, *EvoSuite* generates test suites as an initial population. To control the size of test suites, it determine the

Table I: An example showing behavioral difference of mutants against the original programs

Test	d(t_i,P_o,m_1)	d(t_i,P_o,m_2)	d(t_i,P_o,m_3)	d(t_i,P_o,m_4)
t_1	1	1	1	1
t_2	0	0	1	1
t_3	0	1	0	1

maximum number of test cases and their maximum length. Then it evaluates each individual's fitness. The fitness function which is used in *EvoSuite* is general mutation adequacy criterion (*k-criterion*: kill-criterion). *k-criterion* just counts the number of killed mutants and gives a mutation score of a test suite. When every individual has the fitness, *EvoSuite* selects individuals as parents repeatedly and crossover not letting the total test suite size of offspring bigger than that of parents. After creating offspring from parents, *EvoSuite* mutates its offspring. If offspring has the same fitness with their parents and the test suite size of offspring are bigger than that of parent, offspring is not included to next generation.

These five steps repeats until *EvoSuite* satisfies stopping conditions. The stopping conditions could be the iteration of generation or timeout conditions. The final output of *EvoSuite* is the most optimized test suite which has the smallest fitness value.

B. Distinguishing Mutation Adequacy Criterion

k-criterion focuses on how many mutants are killed by test suites. It does not consider the diversity of mutants. As we have different types in real-faults, there would be different types of mutants. If we can leverage that information, we can make a more powerful test suite. *d-criterion* distinguishes mutants along their kill-patterns.

When P is a set of programs which includes the program under test and T is a set of test, an original program $p_o \in P$, a mutant $m \in M \subseteq P$ mutated from P_o, a correct program $p_s \in P$, and a test $t \in T$ are essential to conduct mutation testing. We can formalize the behavioral difference of mutants against the original program when tests are executed on them as follows:

$$d(t, p_x, p_y) = \begin{cases} 1 \ (\textit{true}), & \text{if } p_x \text{ is different with } p_y \text{ for } t \\ 0 \ (\textit{false}), & \text{otherwise} \end{cases} \tag{1}$$

We can differentiate mutants on table I by using the above formula. If test t_i killed a mutant m_j, 1 is described on the table otherwise 0. Each column of behavioral differences shows unique pattern. In this case, we can say all mutants are distinguished. *d-criterion* indicates that the more mutants are distinguished by a test suite, the better the test suite.

III. Fitness Function Design

To concretize our approach, we have analyzed the existing fitness function of *EvoSuite*. There are two existing types of mutation adequacy criterion: weak mutation criterion and strong mutation criterion. A mutant is *weakly* killed when a state change is observed. It is also said *touched* as the state change becomes notable. However, a state change is not always propagated to an output. A mutant is said *strongly* killed when a difference in outputs of original program and a mutant is observable. We have focused on strong mutation criterion since strong mutation criterion subsumes the other.

The following subsections introduce the characteristics of fitness functions, the structure of strong mutation fitness, and the proposed fitness applied by *d-criterion*. We also explains how diversity-aware fitness is computed in our approach.

A. The Evaluation Strategy of EvoSuite

The concept of coverage criteria is a maximization problem since coverage is originally based on the notion of how much search space is covered. However, *EvoSuite* takes its fitness which is required to be minimized. A quick way to convert maximization problem into minimization problem is to see the size of uncovered space out of the whole space. We can simply define mutation fitness as the number of unkilled mutants. Rather than taking the ratio of unkilled mutants out of total mutants, the mutation fitness takes just the count of mutants so to be unbounded.

The strong mutation fitness of *EvoSuite* adds branch fitness to mutation fitness. A test suite is expected to cover at least the original program. Thus, this inclusion is meaningful as a sanity check. Mutation fitness also can leverage detailed information that how strongly a mutant is killed. The detailed information about branch fitness and mutation fitness used in the strong mutation fitness is introduced in the following subsections.

B. Branch Fitness

The value of branch fitness is the same with the value used for branch coverage criterion. It measures how many branches and methods of a original program a test suite covers. A branch consists of true and false predicates, and methods are classified according to the existence of any branches. The count of unreached predicates and uncalled methods is added to the fitness. When a branch is reached, branch distance [3] of the branch is computed, normalized, then added to the fitness. The range of branch fitness is unbounded but dependent on the size of the original program.

C. Mutation Fitness

The types of mutants run by a test suite can be divided into three groups: *alive*, *weakly killed*, *strongly killed*. The value of mutation fitness of a mutant differs according to how they are killed. When a mutant is alive against a test suite, it is penalized with three points to the fitness. A weakly killed mutant is penalized with infection distance [], adding points between one and two. A strongly killed mutant is

Table II: Distinguishment Symbols

State	Symbol	Description
Alive	0	a mutant is alive against a test
Weakly killed	1	a mutant is weakly killed by a test
Strongly killed	1	a mutant is strongly killed by a test
Timed-out	-	a mutant is unable to run within the timeout

penalized with impact distance (or propagation distance) [], adding points between zero and one to the fitness.

Mutation fitness makes the sum of fitness values of all mutants. The total value of mutation fitness is naturally bounded between zero and three times the number of mutants. Since the number of mutants varies with mutation operators and target programs, the maximum of the fitness cannot be fixed. This nature should be considered when designing additional fitness.

D. Diversity-Aware Fitness

We propose a fitness based on *d-criterion*. The notion of diversity can be translated into a fitness, counting the unique mutants. Those are uniquely distinguished mutants among all mutants. Similar to existing fitness, diversity-aware fitness is computed as the number of total mutants subtracted by the number of unique ones. To compute the fitness, we need to identify whether and how a test case kills a mutant.

The distinguishment symbols we have used to represent the state of a mutant are shown in Table II. We have distinguished one more state of a mutant from *alive* state, which is *timed-out*. In practice, some mutants are just timed-out because budget is limited as stopping condition. Following the spirit of diversity, those are distinguished from mutants which are literally alive. Mutants with identical patterns are grouped through the following notation, and the count of unique patterns is used to compute the value of diversity-aware fitness.

E. Building Fitness Functions

In order to evaluate the effectiveness of *d-criterion* adapted to genetic algorithm, we need to compare the effect of diversity-aware fitness with results of *k-criterion*-based fitness. We can simply add diversity-aware fitness to the original fitness of *EvoSuite*. However, it would be unfair to compare them directly because the original fitness is designed two-fold while the new fitness is designed three-fold. Additional kill-based fitness should be devised which takes its range as the same as diversity-aware fitness.

Similar to the structure of diversity-aware fitness, the number of unkilled mutants and the number of total mutants can be used to construct an additional kill-based fitness. We can define the fitness as the number of total mutants subtracted by the number of unkilled ones. These simple definitions reflects the key notion of each criterion.

Table III: Fitness Functions

Nickname	Formula
ORIGINAL	(Branch fitness) + (Mutation fitness)
KA(k-fitness, additional)	(BF) + (MF) + (Kill-based fitness)
DA(d-fitness, additional)	(BF) + (MF) + (Diversity-aware fitness)
KO(k-fitness, only)	(Kill-based fitness)
DO(d-fitness, only)	(Diversity-aware fitness)

We have constructed two fitness functions to see the effectiveness according to combination of different fitness. The kiil-based fitness added to the original one is denoted by *KA*, similarly the diversity-aware fitness added to the original one is denoted by *DA*. To see the pure impact of newly defined functions, we also made two versions of fitness function where original fitness is removed and contains only either kill-based fitness or diversity-aware fitness.

IV. Evaluation

In this section, we explain how we setup and execute the experiment and how we analyze and evaluate the result from the experiment.

A. Experiment Setup and Design

We modify the fitness function in *Evosuite* with applying *d*-criterion. To check the effectiveness of the modified version of *Evosuite*, we look into the program with three questions which are described below.

- **RQ1**. Timeout issue: *How much time would be enough for a test suite to evolve?*
- **RQ2**. Test suite size: *How different are sizes of the test suites generated by different fitness functions?*
- **RQ3**. Fault Detection Capability: *How effective are test suites generated by different fitness functions, in terms of fault detection capability?*

Evosuite has an timeout property to make stop the test suite evolution. There is a chance to be premature for a test suite during the evolution procedure because a subject program is too small to have enough time to get evolved. This makes an unfair situation to compare with others which have appropriately got evolved. Therefore, experiment for RQ1 can find a suitable timeout. Corresponding to [2], test case generation using *d*-criterion would make more test cases. Since we also think the criterion for distinguishing mutants need more constraints to satisfy, RQ2. will give us the answer about it. In addition, test suite generated using *d*-criterion have expected to perform better effectiveness in detecting faults. By resolving RQ3, we can identify the efficiency in practice.

Our experiments are proceeded in the environment detailed below.

Table IV: Subject Programs Used in Empirical Test

Program	Description	SLOC	# of Mutants
Pattern	Finds a string match	67	193
Sort	Collection of bubble, selection, insertion sorts	43	259
Statistic	Computes statistics information of an integer array.	57	273
Triangle	Decides the form of triangle with sides length info.	58	291
Stack	Simple implementation of stack data structure	20	50

Table V: Five Versions of *Evosuite* with Various Fitness Functions

Evosuite	Fitness	Function formula
Original	Original fitness	(Branch+Traditional mutant) fitness
KA	K-fitness (additional)	(Original+Kill-based) fitness
DA	D-fitness (additional)	(Original+Diversity) fitness
KO	K-fitness (Only)	Kill-based fitness
DO	D-fitness (Only)	Diversity fitness

- Environment: Microsoft Azure Virtual Machine
- CPU: 2.4GHz Intel Xeon® E5-2673 v3 (Haswell) quad cores
- RAM: 14GB
- Operating System: Ubuntu Server 16.04 (LTS)

Within this environment, we pick five programs to evaluate the modified *Evosuite* in test data generation. Programs are described in Table IV and collected from textbooks, Wikipedia, or tutorial documents. We use these programs as in-defective programs and make five faulty versions of each program on purpose which is replicating the real fault. *Evosuite* which we use to generate a test suite has five versions as illustrated in Table V. Five versions of *Evosuite* will provide these information: Original *Evosuite* provides a reference value and KA, DA versions provide the possibility that *d*-criterion can improve the fault detection capability, and then we can finally check how much the revised fitness can have ability of detection by itself with KO and DO.

Overall experimental procedure is as follows. First, we put five subject programs which are considered as non-faulty programs into original and modified *EvoSuite* (five versions of *Evosuite*) to generate test suite. Then we use five faulty programs which mimic the real fault based on subject programs to assure the generated test suites can detect the fault properly. We run the procedure 30 times to minimize variation.

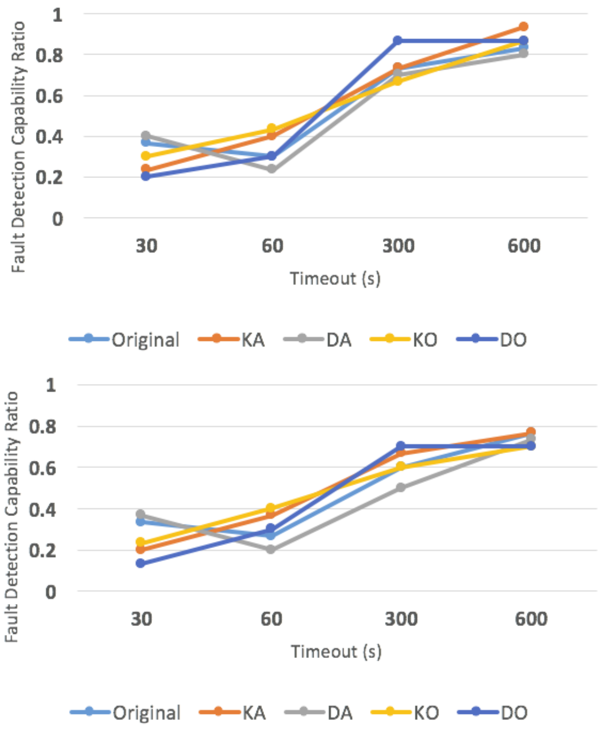

Figure 1: Fault detection capability Graph of 4th (upside) and 5th (downside) faulty program of Pattern

B. Results

RQ1. Timeout Issue

Before comparing the performance of various *Evosuite*, we have to find appropriate timeout for test suite to evolve and get matured. At first, we build four different times to stop the test suite evolution; 30, 60, 300 and 600sec. Next, as you can see the result in Fig.1., we calculate the fault detection capability (FDC) in how many test cases can detect with generated number of test cases in a test suite.

In Fig. 1., timeout 30sec is noticeably low in detection capability and we think that 30sec for evolution is not enough. Short evolution time can make a test suite leave premature and this means the test suite have high probability of being initial random suite. However, as it goes to 600sec, graph have increasing tendency. Since this incremental variation gets smaller continuously and converges, appropriate timeout to get matured can be decided to 600s.

RQ2. Test Suite Size

In [2], a test suite generated using *d*-criterion will be larger than existing methods. However, as we can recognize in Fig.2. and Fig. 3., there is not much correlation with fitness function neither by time nor by programs. In this report, results of other executions are too simple to have significant outcome, so we mainly explain the results with Statistic program which is relatively complex.

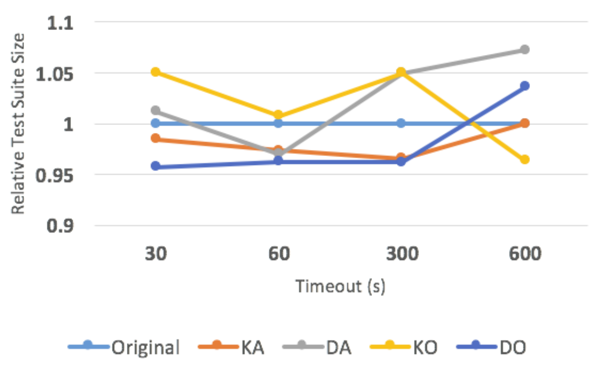

Figure 2: Relative Test Suite Size of Statistic

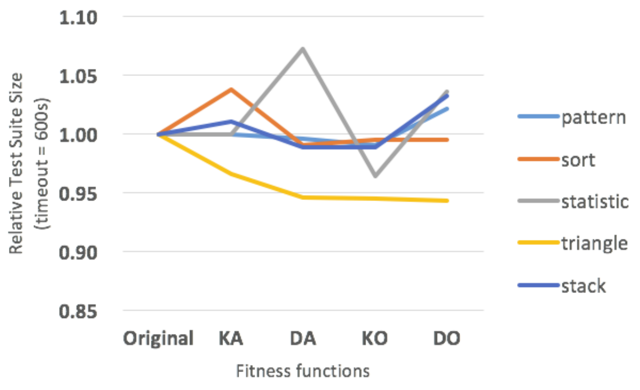

Figure 3: Relative Test Suite Size Information (Timeout=600s)

We expect the reason why the variety of test suite size has been occurred because of *whole test suite* generation [1]. *Evosuite* does not generate test suite by adding a test case into a test suite but make a *whole* suite evolve. Therefore, we can identify the test suite size may not have large size of suite because of using *d*-criterion through this result.

RQ3. Fault Detection Capability

Lastly, we analyze the result which use timeout as 600sec to figure out how much effective the *d*-criterion is. For the same reason described in RQ2., we only contain the result of Statistic program. In Fig.4., *Evosuite* with modified fitness function has better fault detection capability than original in second faulty version of Statistic. Moreover, we can see the detection abilities using *d*-criterion have about twice better performance than using *k*-criterion. In result, *d*-criterion can be an effective criterion to generation test data generation.

C. Threat to Validity

In our experiment, there are some remained properties to improve results.

About the timeout issue, we test four different time bound but there can be larger limit than ours. If there is an

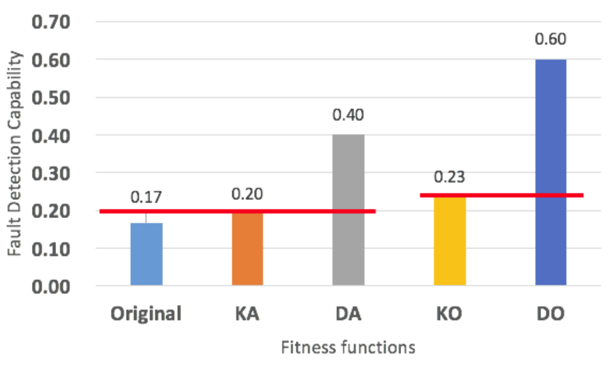

Figure 4: Fault Detection Capability in 2nd Faulty version of Statistic (timeout=600s)

experiment with larger values, we expect the result of fault detection ability shows dramatic increment. In addition, our subject programs are relatively small size programs, but larger programs can be used. However, in this case, programs or methods to analyze are needed. Finally, when we change the fitness function in *Evosuite*, we just add each fitness to existing or add after normalization. There can be various methods to weight on a fitness and the result will come out with significant clarity.

V. Conclusion

This paper has proposed improving test data generation approach for Java programs. We applied the distinguishing mutation adequacy criterion to *Evosuite* which generates test data for Java programs. We evaluated our approach with three research questions. The experimental results showed that the proposed approach is effective and the size of test suite is not getting bigger when d-criterion is applied due to the bloat control which is implemented in *Evosuite*. We plan to apply our approach to real-world programs and check whether the proposed approach can generate effective test data in fault detection or not.

References

[1] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[2] D. Shin, S. Yoo, and D.-H. Bae, "Diversity-aware mutation adequacy criterion for improving fault detection capability," in *International Conference on Software Testing, Verification and Validation Workshops*, 2016, pp. 122–131.

[3] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.