International Journal of Computer Science and Software Engineering (IJCSSE), Volume 5, Issue 6, June 2016
ISSN (Online): 2409-4285 www.IJCSSE.org Page: 97-107

A Fitness Function for Search-Based Testing of Java Classes, which is based on the States Reached by the Object under Test

Ina Papadhopulli[1] and Elinda Meçe[2]

[1, 2] Department of Computer Engineering, Polytechnic University of Tirana, Tirana, Albania

[1]*ipapadhopulli@fti.edu.al*, [2]*ekajo@fti.edu.al*

ABSTRACT

Genetic Algorithms are among the most efficient search-based techniques to automatically generate unit test cases today. The search is guided by a fitness function which evaluates how close an individual is to satisfy a given coverage goal. There exists several coverage criteria but the default criterion today is branch coverage. Nevertheless achieving high or full branch coverage does not imply that the generated test suite has good quality. In object oriented programs the state of the object affects its behavior. Thereupon, test cases that put the object under test, in new states are of interest in the testing context. In this article we propose a new fitness function which takes into consideration three factors for evaluation: the approach level, the branch distance and the new states reached by a test case. The coverage targets are still the branches, but during the search, the state of the object under test evolves with the scope to produce individuals that discover interesting features of the class and as a consequence can discover errors. We implemented this fitness function in the eToc tool. In our experiments the usage of the proposed fitness function towards the original fitness function results in a relative increase of 15.6% in the achieved average mutation score with the cost of a relative increase of 12.6% in the average test suite size.

Keywords: *Structural Testing, Test Case Generation, Search Based Software Testing, Fitness Function, Object State, Coverage Criteria, Mutation Score.*

1. INTRODUCTION

Due to the fact that the influence of software in all areas has grown rapidly in the past 40 years, the software has become very complex and also its reliability is fundamental. All the software development phases have been adapted to produce these complex software systems, but especially the testing phase is of critical importance and testing thoroughly today's software systems is still a challenge. According to a study [1] conducted by the National Institute of Standard & Technology, approximately 80% of the development cost is spent on identifying and correcting defects. It is a well-known fact that it is a lot more expensive to correct defects that are detected during later system operation. Considering past experiences, inadequate and ineffective testing can result in social problems and human/financial losses. In order to improve the testing infrastructure, several efforts have been made to automate this process.

In the unit testing level, there are three approaches towards automation: random testing, static analysis (Symbolic Execution [3]) and metaheuristic search. A considerable number of tools have been developed based on these approaches; eg. RANDOOP [4], EvoSuite [5], AgitarOne [6]. Nevertheless, the effectiveness of these tools is still not completely proved, because the results obtained from the experiments depend on the subjects under test. Usually, a coverage criteria is used to evaluate these tools, but achieving a high degree of code coverage does not imply that a test is actually effective at detecting faults [7]. According to [8], today there is no tool to find more than 40.6% of faults.

This article is focused on structural testing at the unit level of Java programs using Search-Based Software Testing (SBST) [9]. According to [10], SBST has been used to automate the testing process in several areas including the coverage of specific program structures, as part of a structural, or white-box testing strategy. Every unit (class) of the software must be tested before proceeding to the other stages of the development cycle. SBST is a branch of Search Based Software Engineering (SBSE). SBSE is an engineering approach in which optimal or near optimal solutions are sought in a search space of candidate solutions. The search is guided by a fitness function that distinguishes between better and worse solutions. SBSE is an optimization approach and it is suitable for software testing since test case generation is often seen as an optimization or search problem. Since SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective they are at reaching their test

objectives and whether they scale up to realistic development artifacts. However, approaches to empirically study SBST techniques have shown wide variation in the literature. There exist several search-based optimization methods used for test automation; e.g. genetic algorithms, hill climbing, ant colony optimization and simulated annealing, etc, but Genetic algorithms (GAs) are among the most frequently applied in test data generation.

GAs have several components which need to be defined in order for the GA to be implemented. According to [10], the component that affects mostly the results obtained from the search is the fitness function. The fitness function is a mathematical representation of the coverage goal the search should achieve. There are different coverage goals each of them aims at covering certain parts of the unit under test. These different coverage criteria verify the quality of a test suite. The gold criterion is strong mutation, but today this criterion it is mainly used by the research community for evaluation of proposed techniques. The most used criterion is branch coverage [11]. However achieving high branch coverage (even 100%), for some classes is not sufficient.

In object oriented programs the state of the object is a factor that affects the execution of a method. This is why the state of the object of the Class Under Test (CUT), should evolve during the search in order to discover hidden features of the class [12]. A test case that puts the object in one or several new states is of interest in the testing context. The scope of this paper is to propose and evaluate a new fitness function, which rewards the test cases according to branch coverage and also according to the new states the object has taken during the execution of the test.

The rest of this paper is organized as follows: In the second section we explain in what unit testing of java programs consists and in the third section we present an overview of GAs. The fourth section is focused on branch coverage and the fifth section presents the proposed fitness function. The implementation of the proposed fitness function is described in section six. The seventh section gives details of the experimental setup and in the eighth section the results achieved are presented and discussed. We conclude finally with the conclusions we have come preparing and accomplishing this study.

2. UNIT TESTING FOR OBJECT ORIENTED SOFTWARE

Software testing at the unit level (Java classes) consists of three steps:

1. The design of test cases
2. The execution of these test cases
3. The determination of whether the output produced is correct or not.

The second step is performed fully automatically using frameworks like JUnit [2]. Automatically generating the test oracle is still a challenge and there exists few research publications regarding this topic [13], therefore the third step is almost completely performed manually by the testers. Regarding the first step, there exist a lot of research effort for the generation of test cases automatically. Due to the complexity and the diversity of the programs under test this is still an open research topic. Moreover test cases in object oriented unit testing are not just a sequence of input values like in procedural languages. According to [14], a unit test of a Java class must accomplish the following four tasks:

1. Create an object of the class under test using one of the available constructors.
2. Invoke a sequence of zero or more methods on the created object.
3. Execute the method which is currently under test.
4. Examine the final state of the object to produce the pass/fail result

Some parameters in method calls are objects themselves, thus requiring further object constructions and as a consequence task 1 and 2 must be repeated for each parameter of object type.

The statements for Java unit test cases are:

1. Primitive statements: declaration of variables e.g. int a = 15;
2. Constructor statements: construction objects of any given class e.g. String s = new String(“Test”);
3. Method statements: calling the methods of any given class e.g. char b = s.charAt(2);
4. Field statements: accessing the fields of any given class e.g. int c = ob.size;
5. Assignments statements: assign values to the fields of any given class e.g. ob.size = 17;

Since objects have a state, the results are affected by the state of the object under test and of the object parameters.

3. GENETIC ALGORITHMS

Genetic Algorithms (GAs) are inspired by natural evolution. They were first introduced by Holland in 1975. Today GAs are used for optimization in testing real life applications. The most important components in GA are:

- representation of individuals: genotype (the encoded representation of variables) to phenotype (the set of variables themselves) mapping
- fitness function: a function that evaluates how close an individual is to satisfy a given coverage goal
- population: the set of all the individuals (chromosomes) at a given time during the search
- parent selection mechanism: selecting the best individuals to recombine in order to produce a better generation
- crossover and mutation: the two types of recombination used to produce new individuals
- replacement mechanism: a mechanism which replace the individuals with the lowest fitness function in order to produce a better population.

How does the GA work?

The space of potential solutions is searched in order to find the best possible solution. This process is started with a set of individuals (genotypes) which are generated randomly from the whole population space (phenotype space). New solutions are created by using the crossover and mutation operators. The replacement mechanism selects the individuals which will be removed so that the population size does not exceed a prescribed limit. The basis of selection is the fitness function which assigns a quality measure to each individual. According to the fitness function, the parent selection mechanism evaluates the best candidates to be parents in order to produce better individuals in the next generation. It is the fitness function which affects the search towards satisfying a given coverage criteria. Usually the fitness function provides guidance which leads to the satisfaction of the coverage criterion. For each individual the fitness is computed according to the mathematical formula which represents how close is a candidate to satisfy a coverage goal, e.g. covering a given branch in the unit under test. GAs are stochastic search methods that could in principle run for ever. The termination criterion is usually a search budget parameter which is defined at the beginning of the search and represents the maximum amount of time available for that particular search.

4. COVERAGE CRITERIA

A. Types of Coverage Criteria

Automatic unit testing is guided by a structural coverage criterion. There exist many coverage criteria in literature, each of them aims at covering different components of a CUT. Nevertheless, not all the criteria have the same strength and can be fulfilled practically. Furthermore some criteria are subsumed by other criteria. Below is a list of coverage criteria for structural testing of Java programs.

1. Line Coverage
2. Branch Coverage
3. Modified Condition Decision Coverage [21]
4. Mutation
5. Weak Mutation
6. Method coverage
7. Top-level Method Coverage
8. No-Exception Top Level Method Coverage
9. Direct Branch Coverage
10. Output Coverage
11. Exception Coverage
12. Path Coverage
13. Condition Coverage
14. Multiple Condition Coverage
15. Condition/Decision Coverage

Mutation criterion is considered the gold criterion in research literature [15]. This criterion is difficult to apply and computationally expensive and it is practically only used for predicting suite quality by researchers. Another option to achieve high quality test cases with search based technique is to use a combination of multiple criteria. [16] performed an experiment to evaluate the effects of using multiple criteria and concluded that:

- Given enough time the combination of all criteria achieves higher mutation score than each criterion separately (except Weak Mutation).
- Using all the criteria increases the test suite size by more than 50% that the average test suite size of each constituent criterion used separately.
- The next best criterion (after Weak Mutation) to achieve high mutation scores is branch coverage.

The usage of multiple criteria increases the overall coverage and mutation score with the cost of a considerable increase in test suite length, so the usage of the combination in practice will be not feasible, because managing large test suites is difficult. A balance between mutation score and average test suite size is achieved with branch coverage criterion.

B. Branch Coverage

The most used criterion is branch coverage, but even though it is an established default criterion in the literature, it may produce weak test sets (mutation score less than 30% [17]). For example consider the Stack implementation in Figure 1.

```
public class Stack {
        private int size = 0;
        private int st [] = new int [4];
        void push (int x){
                if (size < st.length)
                        st[size++] = x;
        }
        int pop (){
                return st[size--];
        }
}
```

Figure1: Example Stack implementation

The class Stack is very simple (8 LOC, 2 attributes, 2 methods). Suppose the test suite generated is the test suite given in Figure 2.

```
1. @Test
2. public void test0()  {
3.   Stack s0 =new Stack();
4.   s0.push(1);
5.   s0.push(0);
6.   int int0 = s.pop();
7.   assertEquals(0, int0);
8.   s.push(0);
9.   s.push(0);
10. s.push((-1916));
11. s.push((-1916));
12. }
```

Fig. 2. Test suite for class Stack

We used EclEmma [35] tool as a plugin in Eclipse to measure branch coverage. The branch coverage obtained by executing this test suite was 100%. There are 4 coverage goals in class Stack (2 methods and 2 branches from the predicate in line 5).
Even though class Stack is very simple, and the branch coverage obtained is 100%, the mutation score is relatively low (29%). We added an assertion in the test (line 7) and used the JUnit framework to run it in Eclipse. The test passed. The tester may assume the class is correct with 100% branch coverage and a passing test.
Is branch coverage sufficient for this class?
Analyzing class Stack we notice the following errors:

- If method pop is called first and then is called method push, an uncaught exception is thrown (field size before calling push is -1).
- If method pop is called two times consequently an uncaught exception is thrown (field size before calling pop is -1).
- If method push is called four times consequently and then is called method pop an uncaught exception is thrown (field size before calling pop is 4).

It is obvious that branch coverage is not sufficient for class Stack!
Is there any possibility to improve the fitness function for branch coverage in order to obtain a test suite with higher quality?
Both of the methods are covered by the test generated, but it is evident that the state of the object (the value of field size) before calling them affects the results of the tests. The same method called on different states of the object behaves differently. This is why, a possibility to improve the suite's ability to detect errors, is to evolve the state of the object during the search in order to put the object in new states that probably can discover interesting behaviors of the CUT. Since the search is guided by the fitness function, then this function should also consider the states reached by a test before evaluating it.

5. THE PROPOSED FITNESS FUNCTION

Fitness functions are a fundamental part of any search algorithm. They provide the means to evaluate individuals, thus allowing a search to move towards better individuals in the hope of finding a solution [18]. The approach considered here is to minimize the fitness function during the search. The fitness function proposed in this paper rewards the individuals based on how close they are at covering a target (branch) and the states they put the object under test. This function is a mathematical equation depending on the:

- Approach level
- Branch Distance
- New states achieved

A. Approach Level

For each target, the approach level show how many of the branch's control dependent nodes were not executed by a particular input [20]. The fewer control dependent nodes executed, the "further away" an input is from executing the branch in control flow terms. The approach level is the most used factor in the fitness function for structural criteria, but the fitness landscape contain plateaus because the search is unaware of how

close a test case was to traversing the desired edge of a critical branching node.

B. Branch Distance

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current "target" branch. For every operator the branch distance is calculated using the formulas introduced by Tracey [19].
The approach level is more important that the branch distance and as a consequence the branch distance should be normalized at the fitness function formula. This distance will be normalized at a value between 0.0 and 1.0. Value 0.0 means "true"; the desired branch has been reached. Values close to 1.0 means that the condition is far from being fulfilled. Intermediate values guide slightly the search towards the accomplishment of the condition (in order to remove plateaus in the fitness landscape). The formula for branch distance in our proposed fitness function is the formula introduced by Arcuri [21].

$$BD(normalized) = \frac{BD}{BD + \beta}$$

BD is the branch distance before normalization and β is 1.

New States Achieved (NSA)

With the term state in this paper we refer to:

Definition 1. State: The set of the values of all the fields in the CUT before calling a method + the method called.

For example, for the class Stack the two states:

field size = 0 and filed st = !null, before calling method push

field size = 0 and filed st = !null, before calling method pop are considered two different states and both of them are interesting in the testing context.
The total number of states in the CUT is computed as a product of all the possible combinations of the class fields (declared non final) after abstraction (explained in the next section), with the number of public methods.
The approach level is more important that the number of new states achieved and as a consequence this factor should be normalized at the fitness function formula. The normalization formula is:

$$NSA = \frac{states\_total - states\_new}{states\_total}$$

The greater the number of the new states achieved by a test case the smaller this factor in the overall fitness.
The fitness function proposed considers the three factors described above and is computed with the formula:

$$Fitness = approach\_level + \frac{BD}{BD + 1} + \frac{states\_total - states\_new}{states\_total}$$

C. Abstract States

If we use the real values of the fields, the number of states will be infinite. Moreover, not all the states are of equal relevance during testing. For example, from the testing prospective, calling method pop() of the class Stack with field size = 1, is the same as calling method pop with filed size = 2. On the other hand calling method pop() with filed size = 0 in not the same, since this state reveals an interesting behavior of the object under test. Therefore, we use abstractions over the values of the fields rather than the concrete values themselves. We use a state abstraction function provided by Dallmeier at al. [34]. The abstraction is performed based on the three rules below:

- If the type of the field is concrete (int, double, long etc), the value will be translated in three abstract values: $x_i < 0$, $x_i = 0$ and $x_i > 0$.
- If the type of the field is an object, the value will be translated in two abstract values: $x_i = null$ dhe $x_i \neq null$
- If the type of the field is Boolean, there is no need to do translation, since there are only two values.

For example the combinations of the field values of class Stack, after abstraction are those listed in Table 1.

Table 1: Combination of field values for class stack

	size	st
state1	= 0	null
state2	> 0	≠null
state3	< 0	≠null

6. IMPLEMENTATION OF THE PROPOSED FITNESS FUNCTION

The proposed fitness function was implemented in the eToc [22] tool. eToc is a simple search based tool for unit testing of Java programs. Is uses GA and branch coverage criterion. This tool has been mentioned in many research works and has been used as the basis for the design of other tools. eToc is appropriate for the scope used in this work. In the high level architecture of this tool [22], the Branch Instrumentor module and the Test Case Generator module need to be differently

implemented for the search to be guided by the proposed fitness function. The new implementation of these modules is described below.

6.1 The Intrumentor

The function of the instrumentor module is to transform the source code of the CUT in order to provide information about the executed branches, the branch distance and the states achieved during execution. The new statements added during instrumentation must not change the behavior of the CUT. In order to obtain information for the states reached by the object under test, for each of the attributes (except those declared final) of the CUT, a ***get*** method will be added. A static analysis can be used to provide information about the mutators and inspectors methods of a class [23][24], but in this case a static whole-program analysis is required, which is very expensive for this context used. Since it is not the purpose here to obtain a behavioral model of the CUT, the get methods are appropriate to be used as inspectors for obtaining the state of the object because these methods:

- Return the value of an attribute
- Do not take parameters
- Do not have any side effects in the execution of the program.

Based on the state definition given in section 5.C, the get methods should be called before the execution of each method of the CUT, so during instrumentation the statements calling the get methods are added before the existing statements of each method. The concrete values are translated in abstract values as described in section 5.D. Then the states reached by a test case are saved in a LinkedList and consequently during fitness evaluation the new states achieved by a test case can obtained.

```
public class Stack {
     private int size = 0;
     private int st [] = new int [4];
     void push (int x){
             returnState();
             if (size < st.length)
                     st[size++] = x;
     }
     int pop (){
             returnState();
             return st[size --];
     }
     public int getsize1(){
             return size;
     }
     public Object getst1(){
             return st;
     }
     public void returnState (){
      reachedStates.add(String.valueOf(getsiz
     e1()+" "+getst1());
     }
     static java.util.List reachedStates;
     public static void newReachedStates()
     {
       reachedStates = new
java.util.LinkedList();
     }
   }
```

Fig. 3. Class Stack after instrumentation for the new ststes achieved

6.2 The Test Case Generator

The instrumented version of the CUT is executed repeatedly with the scope to cover a specified target (branch of the CUT). The state lists resulting after each execution are compared with the state lists of the test cases that make up the population. The new states reached by an individual are used to compute part of its fitness.

This module is also responsible for the minimization of the generated test suite. Normally during minimization the tests that do not cover any target that is not covered by any other test are omitted from the test suite. Taking into consideration that a test case that reaches one or more new states is important in the testing context, before removing a test case because it does not cover any new target, it will be reconsidered regarding the states it puts the object under test in. The test cases which contain unreached states in their state lists, will be part of the final test suite. The proposed minimization has the advantage that it probably increases the number of tests in the generated test suite and as a consequence it also increases the length of the test suite. On the other side the usage of the proposed fitness function is expected to increase the capability of the test suite to detect errors. An experimental evaluation of the new fitness function is presented in the next section.

Table 2: Characteristics of the Classes Selected for the Experiments: Name of the Project, LOC, Number of Public Methods, Number of Branches, Number of Mutants, Number of Non-Final Fields, Cyclomatic Complexity, Project URL

Project	Class	LOC	Branches	Mutants	Non-final Fields	Public Methods	Cyclomatic Complexity	URL e projektit
	Staku	**12**	4	22	2	**2**	**1.5**	
Commons CLI	**Option**	**155**	131	140	9	**42**	**1.52**	https://commons.apache.org
	TypeHandler	**124**	28	28	0	**9**	**2.66**	
	AlreadySelectedExcept	**26**	4	1	2	**2**	**1**	
	OptionGroup	**86**	21	19	2	**8**	**1.875**	
Math4J	**Rational**	**61**	36	161	2	**19**	**1.526**	https://sourceforge.net/projects/math4j
	ExponentialFunction	**40**	11	31	1	**9**	**1**	
	ArrayUtil	**320**	167	1769	0	**36**	**3.48**	
	PolyFunction	**245**	100	827	2	**12**	**3.63**	
	Complex	**102**	24	682	2	**20**	**1.091**	
jdk (java.util	**StringTokenizer**	**313**	78	434	7	**6**	3.12	
Genetic Algorithm in Java	**GAAlgorithm**	**65**	14	6	6	**8**	**2**	https://sourceforge.net/projects/gaj
	Genome	**14**	9	21	3	**4**	**1.4**	
	Population	62	13	44	4	11	1.08	
ObjectExplorer4J	**ExplorerFrame**	**158**	26	74	8	**9**	**1.44**	https://sourceforge.net/projects/objectexplorer
	ObjectViewManager	**114**	41	41	8	**17**	**1.571**	
NewzGrabber	**DirectoryDialog**	**177**	47	155	16	**13**	**2.235**	https://sourceforge.net/projects/newsgrabber
	NewsFactory	**121**	45	88	4	**7**	**4**	
	SongInfo	**55**	12	59	3	**4**	**2**	
	BatchJob	**28**	11	29	8	**10**	**1.27**	
	StringSorter	**63**	12	47	1	**4**	2.2	
	OptionsPanel	**363**	75	214	15	**4**	9.8	
Jipa	**Label**	**18**	11	42	3	**4**	**1.8**	https://sourceforge.net/projects/jipa
	Variable	**40**	23	87	3	**4**	**2.1**	
Total		***276***	***943***	***5021***	***111***	***264***		

7. EXPERIMENTAL EVALUATION

In this work we aim to answer the following research questions:

- *RQ1: How does the usage of the proposed fitness function affect the branch coverage?*
- *RQ2: How does the usage of the proposed fitness function affect the mutation score of the suite?*
- *RQ3: How does the usage of the proposed fitness function affect the number of suite's test cases and their size?*

7.1 System Characteristics

For the experiments we used a desktop computer running Linux 32 bit Operating System, 1 GB of main memory and a Intel Core 2 Duo CPU E7400 2.8GHz x 2 Processor.

7.2 Subject Selection

Selecting the classes under test is very important since this selection affects the results of the experiments. We chose 7 open source projects and selected randomly 23 classes from them. Also, the class Stack discussed throughout this paper was used as a subject for the experiments. To obtain comprehensive results, the evaluation must be done to real and not simple subjects. Also these subjects should not have any common characteristics which affect the obtained results. The characteristics of the 24 classes are listed in Table 2. The information about LOC (without comments and empty lines) and cyclomatic complexity is obtained using Metrics 1.3.6 [25], as a plugin in Eclipse. As can be noted from Table 2, the classes have very different characteristics and complexity.

Five of the projects were downloaded from SourceForge [26] which is today the greatest open source repository (more than 300,000 projects and two million of users). One project was downloaded from the Apache Software Foundation [27] which exists from 1999 and has more than 350 projects (including Apache HTTP Server). Class StringTokenizer was taken from the java.util package which is part of jdk 1.8.0. This package has been used by several studies for evaluation of automatic test case generation techniques.

7.3 Parameters of GA

Defining the parameters of GAs to obtain the optimal results is difficult and a lot of research effort is dedicated to this topic [28][29]. Therefore we let the parameters of the GA to their default values [22]. The values of three of the most relevant parameters are listed in Table 3. Regarding the search budget, it was determined depending on the experiment and will be shown next for each experiment.

Table 3: Parameters of GA

Parameter	Value
Population Size	10
Search Budget	600s
Maximal number of generations/target	10

7.4 Experiment

For each of the classes we run eToc with the following configurations:

1. Original Fitness (OF) function with search budget of 2 min
2. Proposed Fitness (PF) function with search budget of 2 min
3. Original Fitness (OF) function with search budget of 10 min
4. Proposed Fitness (PF) function with search budget of 2 min

To overcome the randomness of the genetic algorithms each experiment was repeated 5 times.
The results of the experiments (average of all runs) are presented in Table 4.

Table 4: Branch Coverage, Mutation Score, Number of Tests, Length of Test Suite for Each Configuration, Average of All Runs for Each Cut

Class	BC with OF (2 min)	BC with PF (2 min)	BC with OF (10 min)	BC with PF (10 min)	MS with OF (2 min)	MS with PF (2 min)	MS with OF (10 min)	MS with PF (2 min)	No. test with OF	Test length with OF	No. test with PF	Test length with PF
Staku	100	100	100	100	29	**72**	**29**	**72**	2	8	4	15
Option	69	69	69	69	41	**49**	**41**	**49**	62	147	71	166
TypeHandler	75	75	75	75	**46**	**46**	**46**	**46**	12	24	12	24
AlreadySelectedException	100	100	100	100	100	**100**	**100**	**100**	3	5	3	5
OptionGroup	100	100	100	100	84	**89**	**84**	**89**	8	27	7	35
Rational	94	94	94	94	75	**79**	**75**	**79**	12	24	12	31
ExponentialFunction	100	100	100	100	60	**55**	**60**	**60**	8	16	7	15
ArrayUtil	100	99	100	100	9	**9**	**9**	**9**	64	141	64	141
PolyFunction	-	-	85	87	-	-	**31**	**38**	27	89	30	98
Complex	100	100	100	100	34	**37**	**34**	**37**	13	27	12	31
StringTokenizer	65	65	69	69	**15**	**21**	**19**	**23**	8	18	16	33
GAAlgorithm	93	93	93	93	33	**33**	**33**	**50**	10	21	8	19
Genome	44	44	55	55	0	**4**	**0**	**4**	3	6	4	10
Population	92	92	100	100	32	**32**	**32**	**32**	11	29	11	29
ExplorerFrame	8	15	8	15	0	**3**	**0**	**3**	2	2	2	3
ObjectViewManager	54	54	54	54	17	**24**	**17**	**24**	2	3	2	3
DirectoryDialog	6	6	6	6	0	**0**	**0**	**0**	5	11	5	11
NewsFactory	-	-	-	-	-	-	-	-	-	-	-	-
SongInfo	50	50	50	50	22	**27**	**24**	**27**	5	12	8	19

BatchJob	100	100	100	100	62	**69**	**62**	**69**	10	20	9	22
StringSorter	100	100	100	100	17	**17**	**17**	**17**	6	17	6	17
OptionPanel	-	-	37	37	-	-	**3**	**9**	7	21	8	19
Label	100	100	100	100	55	**55**	**55**	**55**	4	16	4	16
Variable	100	100	100	100	55	**56**	**56**	**59**	6	9	9	19
Average	**60.5**	**69**	**74.8**	**75.2**	**37.9**	**42.7**	**35.9**	**41.5**	**-**	**-**	**-**	**-**
Total	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**290**	**693**	**314**	**781**

8. RESULTS AND DISCUSSION

- *RQ1: How does the usage of the proposed fitness function affect the branch coverage?*

The branch coverage was measured with EclEmma. For both functions the average branch coverage is greater when the search budget is 10 min. This result was expected since the individuals improve during the search and more time results in better solutions.
In order to do the best comparison of the approaches we focus on the case with search budget of 10 min in this section, since for the scope of the experiment, it is not appropriate to compare results affected by the limited search time.
The difference between the average branch coverage is inconsiderable (0.4%) when a search budget of 10 min is used. This difference may be due to the randomness of the results achieved by the search. Since the approach presented in this work does not change the targets to cover, the almost equal coverage was expected. For the class ExplorerFrame, there is an increase of 7% in the coverage achieved by the proposed approach. Even though the targets are identical, the proposed function rewards the individuals that reach more new states and therefore the test cases after minimization may be different and more complex. So, this increase probably is the effect of indirect coverage.
Only in the case of class ArrayUtil there was a decrease of 1% in the coverage achieved, with budget 2 min, but more likely it is due to the randomness of the search. For the class NewsFactory the search failed to produce results for both approaches. We changed the parameters of the GA, but even for a population of 20, or 30 individuals, no results were generated. It is not the scope of this work to investigate the reasons why this happened.

RQ1: In our experiments, there is no difference in the average branch coverage achieved between the usage of the original fitness function and the proposed fitness function.

- *RQ2: How does the usage of the proposed fitness function affect the mutation score of the suite?*

Since mutation score is the measure used in the strongest criterion (Mutation Coverage), here we have used it to measure the quality of the generated test suite. Computing the mutation score for a test suite requires determining, for every mutant, whether the test suite succeeds or fails when run on the mutant. In the worst case each test must be run on each mutant. For each of the classes the mutants were generated using as a plugin in Eclipse the tool MuClipse v1.3 [30]. MuClipse generates mutants using the traditional operators and the operators in the class level [31]. The number of generated mutants for each class is given in Table 2. Even classes with a small number of LOC can have many mutants (e.g. class Stack has 22 mutants).
Assertions were inserted manually to the tests generated, so that these cases can be used by MuClipse. Then, the generated tests were executed with JUnit against all the mutants and the presence of failures shows that the tests were able to kill the mutants.
The results of the mutation scores of each class for all the configurations are given in Table 4.
The mutation scores achieved by both of the fitness functions are far from the optimal value (100%). Almost this range of mutation scores is also obtained from other studies [32]. The main reasons of these low scores are:

- the targets to cover are the branches and not the mutants
- the presence of equivalent mutants (behave the same as the original program) which cannot be killed.

Nevertheless, despite the relatively low mutation scores, our interest is focused on the difference between the scores achieved by the original function against the proposed function.
For 6 classes (6/23 = 26%) there is an improvement in the mutation score achieved when using a search budget of 10 min against a search budget of 2 min.

For the same reasons mentioned in the discussion of RQ1, to answer RQ2 we are focusing mainly at the results achieved with a search budget of 10 min. The average mutation score reached by the original function is 35.9%, whereas the mutation score reached by the proposed function is 41.5%, thus a difference of 5.6%. The improvement is 5.6/35.9 = 15.6%. For 15 classes out of 23 (15/23 = 65%), there is an improvement in the mutation score achieved by the proposed function; for the remaining 8 classes (8/23 = 35%), the scores achieved are identical. There is no class where using the proposed function results in a lower mutation score. Even though we are aware that the results depend on the CUT (despite the fact that CUT chosen have different characteristics), the results obtained are very promising.

RQ2: In our experiments, the usage of the proposed fitness function results in a relative increase of 15.6% in the average mutation score achieved against the original fitness function.

- *RQ3: How does the usage of the proposed fitness function affect the number of suite's test cases and their size?*

Automatically generated JUnit tests need to be manually checked in order to detect faults because automatic oracle generation is not possible today. This is the reason why not only the achieved coverage of the generated test suite is important, but the size of the test suite is of the same importance [33].

Here we refer to the *size of a test suite* as the number of statements after the minimization phase (without assertions).

Only the results achieved with a search budget of 10 min, are shown in Table 4, because in answering RQ3 we are interested in the number of tests generated and their size in the "worst case". The minimization phase does not depend on the search budget, so the results with search budget of 10 min, subsume the scenario with a search budget of 2 min. The LOC of the generated suite was obtained with the tool Metrics 1.3.6.

There is an increase of 314 – 290 = 24 tests in the total number of test generated, or a relative increase of 24/290 = 8.2%. This increase is acceptable, although the number of tests in the test suite is not relevant in respect to the size of the test suite, because having many short size tests is not a problem for the tester who is detecting faults.

Regarding the size of the test suite, we can see from the results in Table 4, that using the proposed fitness function results in an average test suite size of 33.9 (781/23) statements. The relative increase is (33.9 – 30.1) / 30.1 = 12.6%. For 8 of the classes (34%), there is no change in the average test suite size. Regarding classes ExponentialFunction and GAAlgorithm (8.7% of the classes), there is a decrease in the average test suite size, although there is no decrease either in branch coverage or mutation score. These results are explained with the appearance of indirect coverage [36].

ArrayUtil is the class with the greatest test suite size because of the large number of branches (167).

The average increase in test suite size with the usage of the proposed function is the consequence of two reasons:

- During the minimization phase the test cases that do not cover any target, but put the object under test in new states, are added in the minimized test suite (as explained in Section 6)
- Two different fitness functions probably will generate different test suites with different number of statements (not necessarily a larger number).

RQ3: In our experiments, the usage of the proposed fitness function results in a relative increase of 8.2% in the average number of test cases and 12.6% in the average test suite size achieved against the original fitness function.

9. CONCLUSIONS

This paper concerns the fitness function used to guide the search during automatic unit test generation of Java classes. The branch coverage criterion is easy to implement but can produce weak test sets. Test cases that put the object under test in new states discover hidden behaviors and consequently are relevant in the testing context. Targeting all the states during the search is impossible due to the fact that some of them are infeasible. In this article we presented a new fitness function that takes into consideration the states reached during the execution of a test case. The implementation of this fitness function is very simple since the targets to cover remain the branches, but the state evolve during the search and the minimization phase the tests that reach one or more new states are not removed even though these tests does not reach any uncovered branches. The usage of the proposed fitness function does not decrease the branch coverage and results in a relative increase of 15.6% in the achieved average mutation score with the cost of a relative increase of 12.6% in the average test suite size. The results are promising but since the subjects under test are very different further evaluation of the proposed approach needs to be performed.

REFERENCES

[1] NIST (National Institute of Standards and Technology): The Economic Impacts of Inadequate Infrastructure for Software Testing, Report 7007.011,
[2] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01-12, Department of Computer Science, Iowa State University, Nov. 2001.
[3] C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later". Communications of ACM, pages 82-90, 2013
[4] C. Pacheco, S. Lahiri, M. Ernst, "Feedback-directed Random Test Generation". In Proceedings of International Conference in Software Engineering (ICSE) 2007
[5] G. Fraser, A. Arcuri, "EvoSuite at the SBST 2015 Tool Competition". In Proceedings of International Conference in Software Engineering (ICSE) 2015
[6] T. Tsuji, A. Akinyele, "Evaluation of AgitarOne". Analysis of Software Artifacts Final Project Report April 24, 2007
[7] G. Fraser, P. McMinn, A. Arcuri, M. Staats, "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study". ACM Transactions on Software Engineering and Methodology, 2015
[8] S. Shamshiri, R. Just, J. Rojas, G. Fraser, P. McMinn, A. Arcuri, "Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges" In Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015
[9] F. Gross, G. Fraser, A. Zeller, "Search-based system testing: high coverage, no false alarms". In Proceedings of International Symposium on Software Testing and Analysis (ISSTA), 2012.
[10] P. McMinn, "Search-based Software Test Data Generation: A Survey", Software Testing, Verification and Reliability, pp. 105-156, June 2004.
[11] K. Lakhotia, P. McMinnb, M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN". Journal of Systems and Software, 2010
[12] M. Mirazz, "Evolutionary Testing of Stateful Systems: a Holistic Approach". PhD thesis, University of Torino, 2010
[13] G. Fraser, A. Zeller, "Mutation-Driven Generation of Unit Tests and Oracles," IEEE Transactions on Software Engineering, 2012.
[14] P. Tonella, "Search-Based Test Case Generation", TAROT Testing School Presenetation, 2013
[15] G. Fraser, A. Arcuri, "Achieving Scalable Mutation-based Generation of Whole Test Suites". Empirical Software Engineering 2014.
[16] I. Papadhopulli, E. Meçe "Coverage Criteria for Search Based Automatic Unit Testing of Java Programs", International Journal of Computer Science and Software Engineering (IJCSSE), Volume 4, Issue 10, October 2015
[17] J. Miguel Rojas, J. Campos1, M. Vivanti, G. Fraser, A. Arcuri, "Combining Multiple Coverage Criteria in Search-Based Unit Test Generation" in Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 436-439, 2011
[18] K. Lakhotia, M. Harman, H. Gross, "AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation on Deployed Automotive Systems". International Symposium on SBSE, 2010
[19] N. Tracey. "A Search-Based Automated Test-Data Generation Framework For Safety-Critical Software". PhD thesis, University of York, 2000
[20] J. Wegener, A. Baresel, H. Sthamer. "Evolutionary Test Environment for Automatic Structural Testing". Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms, 43(14):841{854, December 2001.
[21] A. Arcuri, "It Does Matter How You Normalise the Branch Distance in Search Based Software Testing". Third International Conference on Software Testing, Verification and Validation, 2010"
[22] P. Tonella, "Evolutionary Testing of Classes". In Proceedings of International Symposium on Software Testing and Analysis (ISSTA) 2004
[23] A. Rountev, "Precise identification of side-effect-free methods in java". 20th IEEE International Conference on Software Maintenance (ICSM '04), pages 82–91, 2004.
[24] A. Salcianu, M. Rinard, "Purity and side effect analysis for java programs". In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, pages 199–215, January 2005.
[25] http://metrics.sourceforge.net/
[26] http:/.sourceforge.net/
[27] http://www.apache.org/
[28] A. Aleti, L. Grunske, "Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm". Journal of Systems and Software, 2014.
[29] A. E. Eiben, S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms". Journal: Swarm and Evolutionary Cmputation, pages 19-31, 2011.
[30] http://muclipse.sourceforge.net/
[31] Y. Ma, J. Ouffut, "Description of Class Mutation Mutation Operators for Java", August 2014
[32] D. Le, M. Alipour, R. Gopinath, and A. Groce, "MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs". In Proc. of the International Symposium on Software Testing and Analysis, 2014.
[33] G. Fraser, A. Arcuri, "Handling test length bloat". In Proceedings of ICST, 2013.
[34] V. Dallmeier, C. Lindig, A. Vasilowski, "Mining Object Behaviour with ADABU". In Proceedings of the International Workshop on Dynamic Systems Analysis, 2006
[35] http://www.eclemma.org/
[36] I. Papadhopulli, N. Frasheri, "Today's Challenges of Symbolic Execution and Search-Based for Automated Structural Testing", In Proceedings of ICTIC, 2015.