Results for EvoSuite - MOSA at the Third Unit Testing Tool Competition

Panichella, A.; Kifetew, Fitsum Meshesha; Tonella, Paolo

DOI
10.1109/SBST.2015.14

Publication date
2015

Document Version
Final published version

Published in
IEEE/ACM 8th International Workshop on Search-Based Software Testing

Results for EvoSuite-MOSA at the Third Unit Testing Tool Competition

Annibale Panichella*, Fitsum Meshesha Kifetew†‡, Paolo Tonella†
*Delft University of Technology, The Netherlands
†Fondazione Bruno Kessler, Trento, Italy
‡University of Trento, Trento, Italy
a.panichella@tudelft.nl, kifetew@fbk.eu, tonella@fbk.eu

***Abstract*—EvoSuite-MOSA is a unit test data generation tool that employs a novel many-objective optimization algorithm suitably developed for branch coverage. It was implemented by extending the EvoSuite test data generation tool. In this paper we present the results achieved by EvoSuite-MOSA in the third Unit Testing Tool Competition at SBST'15. Among six participants, EvoSuite-MOSA stood third with an overall score of 189.22.**

***Keywords*-search-based testing; many-objective optimization;**

I. Introduction

EvoSuite-MOSA [1] is a research prototype, implemented by extending EvoSuite [2], for automatic generation of unit test suites for Java classes. This paper describes the results obtained by applying EvoSuite-MOSA to the benchmark used in the tool competition at the International Workshop on Search-Based Software Testing (SBST'15). Details on the competition and the benchmark can be found in reference [3].

We also discuss and analyse the limitations encountered when using EvoSuite-MOSA to generate test cases for those classes in the benchmark where the achieved coverage was particularly low. For example, with classes having environmental dependencies, such as files and databases, reaching high coverage is very challenging even though their structure —in terms of number of branches— is not complex. Finally, we report some remarks and lessons learnt from the contest.

II. About MOSA

EvoSuite-MOSA automatically generates JUnit test cases for a given Java class starting from its bytecode. Built upon EvoSuite [2], EvoSuite-MOSA is based on search-based testing and uses a novel many-objective genetic algorithm, namely Many-Objective Sorting Algorithm (MOSA) [1], targeting the maximization of branch coverage.

The key novelty of MOSA compared to other existing search-based techniques is the re-formulation of the branch coverage criterion as a many-objective problem, where different branches are considered as different objectives to be optimized [1]. In this new formulation, a candidate solution is a test case, while its fitness is a vector measuring the closeness from *all* uncovered branches in the program. The potential problem of such reformulation is that the number of branches, hence of objectives considered by the many-objective optimization algorithm, can be huge, making it difficult to rank the candidate solutions just by relative dominance. We have introduced a novel many-objective sorting algorithm which extends the notion of dominance to make it applicable and effective with the huge number of objectives associated with the branch coverage problem. The search population in MOSA is a set of randomly generated test cases, which are evolved using traditional crossover and mutation operators through subsequent generations. At each generation, the selection of the fittest test cases is based on our sorting algorithm, which gives higher probability of survival to those test cases that are closest to *at least one of the uncovered branches*. The closeness of a test case t to cover each uncovered branch b_i is measured according to (i) the normalized *branch distance* of test case t for branch b_i; and, (ii) the corresponding *approach level*, i.e.,

Prerequisites	
Static or dynamic	Dynamic testing at the Java class level.
Software Type	Java classes.
Lifecycle phase	Unit testing for Java programs.
Environment	All Java development environments.
Knowledge required	JUnit unit testing for Java.
Experience required	Basic unit testing knowledge.
Inout and Output of the tool	
Input	Java classes (compiled)
Output	JUnit test cases (source)
Operation	
Interaction	Through the command line.
User guidance	Through the command line (inspection of generated tests to further improve assertions)
Source of information	http://selab.fbk.eu/kifetew/mosa.html, evosuite.org
Maturity	Research prototype
Technology behind the tool	Search-based testing (Many-Objective Optimization)
Obtaining the tool and information	
License	GPL
Cost	Free
Support	None

Does there exist empirical evidence about	
Effectiveness	[1]
Efficiency	[1]
Scalability	[1]

Table I
Description of the tool that is being evaluated

978-1-4673-7079-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SBST.2015.14

the minimum number of control dependencies between the statements in the test case trace and the branch. For test cases having the same *branch distance + approach level* scores, our *preference criterion* selects the shortest test case (i.e., the test case with the lowest number of statements). The final candidate test suite is represented by a second population, named *archive*, that keeps track of test cases as soon as they cover yet uncovered branches of the program under test. The archive is updated at the end of each generation by considering both the covered branches and the length of test cases. For each covered branch b_i, the archive stores the shortest test case covering b_i [1].

III. Configuration for the Competition Entry

Currently, EvoSuite-MOSA uses only *branch coverage* as structural criterion to be optimised. It extends the EvoSuite test data generation framework by implementing the many-objective genetic algorithm briefly outlined in Section II. All other details (e.g. test case encoding schema, genetic operators, etc.) are those implemented in EvoSuite [2].

There are several parameters that control the performance of the tool being evaluated. We adopted the default parameter values used by EvoSuite [2] in the two previous contest competitions, with the exception of the coverage criterion that in our case is *branch coverage*. For the assertions, we configured the tool to include all possible assertions in the test cases, without running the assertion minimiser. For the execution time, we chose three minutes as timeout for the search and we also added the following further empirical stop condition: if none of the objectives scores (related to uncovered branches) is improved for 100 consecutive generations or for 60 seconds, the search is ended even if the global timeout is not reached.

IV. Benchmark results

The results achieved by EvoSuite-MOSA on the benchmark are reported in Table II. On the 63 classes used in the competition, EvoSuite-MOSA produced an average instruction coverage equal to 56.41%, average branch coverage equal to 46.13% and average mutation score equal to 38.53%. Specifically, it generated on average 20 test cases for each class under test with a search time of only 84 seconds on average.

In general, the average total time required by EvoSuite-MOSA for generating test cases is less than 2 minutes (93.97s in particular), where the total time is computed as the sum of (i) preparation time, (ii) generation time, and (iii) execution time of the final test suite. This result is very surprising if compared with the results yielded by other tools in the contest. Indeed, the total time required by EvoSuite-MOSA is about 25% the time required by the other tools with highest ranks in the contest, while its average branch coverage is sometimes even better. This confirms the strong performance of EvoSuite-MOSA for the branch coverage criterion [1].

It is important to notice that the final score used for the contest aggregates time and coverage scores in a unique scalar value, giving a quite marginal weight to the generation time [3]. Hence, our configuration of EvoSuite-MOSA (the empirical stop condition in particular) was penalizing for our tool. Indeed, we found that by giving more time to EvoSuite-MOSA we would have obtained higher coverage scores, with a major impact on the final aggregate score, despite the increased execution time.

Since our tool is built upon EvoSuite, it inherits all the corresponding limitations of EvoSuite for complex classes, classes with non deterministic code and classes with environmental dependencies [4]. Moreover, the version of EvoSuite we extended with MOSA is older than the current version. In fact, some classes of the benchmark crashed our tool due to bugs fixed in later versions of EvoSuite. After a closer analysis, we also found that in those case where the mutation score is particularly low, compared to the other coverage scores (e.g., statement coverage), there was an internal problem of EvoSuite in generating assertions. The common problem in these cases is that the produced assertions do not hold upon test re-execution and tests with failing assertions are excluded from mutation analysis.

In the remainder of this section, we focus our discussion on some interesting cases where EvoSuite-MOSA achieved very low branch coverage and/or mutation scores.

Java Exception At the time of the competition, the version of EvoSuite we used as baseline to implement our many-objective genetic algorithm was not able to handle classes extending the Java class `Exception`. As a consequence, for the classes `SearchException` or `TwitterException` in the benchmark our tool achieved 0% coverage (either statement or branch coverage) after few seconds of generation time, which corresponds to the time required by the tool for the analysis of the classpath.

Environmental dependencies. A widely known limitation of automatic testing tools regards the generation of test cases for classes having environmental dependencies, e.g., classes accessing external files, etc. Most of the classes in the JWPL project (such as `Page`) fall into this category. For example, class `ARFFHandler` consists of only one method that takes an external file as input. Such file must respect the standard format of `ARFF` files. Similarly, class `Page` requires a valid instance of `Wikipedia` and indirectly a valid instance of the class `DatabaseConfiguration`.

Memory consumption Some classes in the benchmark are particularly complex and they can lead to the generation of test cases that may cause an excessive memory consumption. For example, for the class `Predicates` our tool crashed in some runs because of an *out-of-memory exception*. Running

EvoSuite-MOSA with more memory would have likely resulted in avoiding the out-of-memory exception, hence leading to higher coverage.

Other issues We also faced some issues concerning bugs affecting our version of EvoSuite. For example, our tool crashed in some runs for the class `CharMatcher`, thus, terminating its execution without generating any test case. In other runs, for the same subject, the tool didn't crash and generated test cases providing the following scores: 70% of statement coverage, 62% of branch coverage and 44% of mutation score. For `CycleHandler` and `WikipediaInfo` classes, our tool crashed in every run because of some bugs related to the EvoSuite front-end. Similarly, for `ExceptionDiagnosis` out tool was not able to generate test cases because of a bug concerning classes handling the Java class `Throwable`. Finally, for the class `TwitterImpl` our tool had problems in finding all the required dependences at preparation time. After deeper analysis we found that this is also due to another bug affecting our version of EvoSuite.

V. Remarks for the contest

The results achieved by the tools participating in the contest point out some issues associated with classes that have a simple structure (e.g., few branches or few statements), but are particularly challenging for automatic techniques. For example, some classes requiring external environment resources are very difficult to deal with. Similarly, classes probing time information from the system environment pose challenges for the automatic generation of assertions, since assertions that are valid at generation time may not hold when re-executing the test cases later (e.g., when computing the mutation scores, as done for the contest).

We also notice that the analysis of the scores achieved by different tools in the contest is challenging, since multiple criteria, such as coverage scores and time, are involved. For example, the formula used to aggregate time and coverage scores into a unique scalar value gives a particularly marginal relevance (weight) to the execution time of each tool [3]. Thus, tools that were configured with few minutes of execution time have been penalized with respect to other tools that were configured with more time for the generation process. In fact, the tools that yielded the highest global scores in the contest are also those that were configured with a longer timeout for the search. Indeed, we found that, given the formula used to compute the global score, more execution time is quite likely to result in higher coverage score, and then in higher final aggregate score, just by the possibility to randomly cover previously uncovered branches if additional execution time is available.

Another important point to debate is whether to consider or not the size of the generated test cases as a further criterion for evaluating the cost-effectiveness of automated unit test generators. After our own experiment with some classes in the benchmark, we noticed that minimizing the size of test cases, while keeping the same level of structural coverage, might negatively affect the achieved mutation scores. Indeed, at constant level of statement and branch coverage, larger test cases tend to have higher chances to kill mutants (a key factor for the final score [3]). Thus, since our tool maximizes branch coverage while selecting the shortest test cases at equal level of branch coverage, it is penalized since the final aggregate score includes the mutation score.

For example, for the class `FlushLuceneWork` our tool generated a minimized test suite with 26 statements on average (without counting the assertions), which kills 62% of mutants generated by PIT. When disabling the minimization, our tool generates a test suite with the same branch and statement coverage but with 562 statements on average (without counting the assertions) and a corresponding average mutation score equal to 86% (+26%). Thus, the non-minimized test cases are able to kill more mutants, but they are much longer, hence, they are also more difficult to be analyzed manually.

VI. Conclusion

This paper reports the results obtained by EvoSuite-MOSA, an extension of the EvoSuite test data generation framework that employs a novel many-objective optimization algorithm for branch coverage, at the third SBST Unit Testing Tool competition. We believe that the SBST Unit Testing Tool Competition can be improved along several directions: (1) providing a fixed amount of computational resources and execution time to participants; (2) having multiple separate tracks for different adequacy criteria used to rank tools; (3) avoiding subjects under test with non-deterministic behaviours or environmental dependencies; (4) introducing a measurement of test case size, possibly as a secondary goal of test case generation.

Acknowledgment

We would like to thank Gordon Fraser and Andrea Arcuri for sharing with us their implementation of the contest protocol.

References

[1] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *International Conference on Software Testing, Verification and Validation (ICST)*, 2015.

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

[3] U. Rueda, T. Vos, and I. Prasetya, "Unit testing tool competition — round three," in *8th International Workshop on Search-Based Software Testing*, May 2015.

[4] G. F. Andrea and Arcuri, "Evosuite at the SBST 2013 tool competition," in *IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 257–267.

Table II
EVOSUITE-MOSA RESULTS ON THE BENCHMARK CLASSES

Class Name	N. Test Case	Time in min.		Coverage		
		t_{gen}	t_{exec}	Instruction	Branch	Mutation
com.google.gdata.data.AttributeHelper	60.33	0.59	0.20	74.83	72.56	40.22
com.google.gdata.data.DateTime	44.67	0.39	0.17	70.72	57.86	63.59
com.google.gdata.data.Kind	11.17	0.95	0.04	43.22	39.39	40.58
com.google.gdata.data.Link	55.50	1.59	0.21	76.05	73.20	36.11
com.google.gdata.data.OtherContent	23.00	0.67	0.09	41.78	39.77	32.92
com.google.gdata.data.OutOfLineContent	26.17	0.61	0.10	78.35	76.79	50.69
com.google.gdata.data.Source	50.67	1.34	0.20	50.18	45.27	25.07
net.sf.javaml.core.AbstractInstance	21.33	2.13	0.08	87.97	68.45	63.54
net.sf.javaml.core.Complex	8.00	0.08	0.03	88.21	0.00	50.00
net.sf.javaml.core.DefaultDataset	24.83	0.89	0.08	72.68	74.17	35.48
net.sf.javaml.core.DenseInstance	32.83	2.08	0.11	78.53	83.33	67.20
net.sf.javaml.core.Fold	35.83	0.30	0.12	86.68	88.33	77.08
net.sf.javaml.core.SparseInstance	35.83	0.59	0.12	95.47	84.90	66.91
net.sf.javaml.tools.data.ARFFHandler	2.00	3.03	0.01	2.54	0.00	0.00
twitter4j.ExceptionDiagnosis	1.00	3.04	0.01	0.00	0.00	0.00
twitter4j.GeoQuery	45.00	0.42	0.15	97.83	89.58	76.46
twitter4j.OEmbedRequest	36.17	0.63	0.12	94.02	78.40	13.29
twitter4j.Paging	36.67	0.35	0.13	93.90	93.06	64.89
twitter4j.TwitterBaseImpl	3.33	2.01	0.02	7.81	2.94	0.00
twitter4j.TwitterException	0.00	0.06	0.00	0.00	0.00	0.00
twitter4j.TwitterImpl	1.00	1.48	0.01	0.00	0.00	0.00
com.puppycrawl.tools.checkstyle.api.AbstractLoader	5.00	3.04	0.03	77.00	50.00	30.00
com.puppycrawl.tools.checkstyle.api.AnnotationUtility	12.00	0.33	0.05	52.04	45.00	40.91
com.puppycrawl.tools.checkstyle.api.AutomaticBean	5.17	3.04	0.03	54.97	52.38	16.28
com.puppycrawl.tools.checkstyle.api.FileContents	13.33	0.40	0.07	24.51	19.23	23.67
com.puppycrawl.tools.checkstyle.api.FileText	12.33	0.47	0.06	43.20	46.79	56.38
com.puppycrawl.tools.checkstyle.api.ScopeUtils	18.00	1.49	0.07	18.62	8.33	19.13
com.puppycrawl.tools.checkstyle.api.Utils	23.33	3.06	0.11	61.20	83.97	57.94
com.google.common.base.CharMatcher	28.50	12.64	0.08	20.75	16.67	11.78
com.google.common.base.Joiner	39.67	0.38	0.13	83.05	94.20	81.91
com.google.common.base.Objects	27.50	0.28	0.10	98.13	82.41	94.14
com.google.common.base.Predicates	27.67	0.56	0.06	22.12	12.85	15.19
com.google.common.base.SmallCharMatcher	11.67	3.65	0.04	97.76	93.59	60.07
com.google.common.base.Splitter	52.83	0.69	0.19	94.30	85.90	74.18
com.google.common.base.Suppliers	13.17	0.72	0.03	53.84	47.22	42.05
org.hibernate.search.SearchException	0.00	0.10	0.00	0.00	0.00	0.00
org.hibernate.search.Version	1.83	0.16	0.01	100.00	0.00	0.00
org.hibernate.search.backend.BackendFactory	11.67	4.02	0.06	41.48	28.13	50.00
org.hibernate.search.backend.FlushLuceneWork	2.83	0.13	0.01	89.74	100.00	62.50
org.hibernate.search.backend.OptimizeLuceneWork	2.83	0.12	0.01	89.74	100.00	70.83
org.hibernate.search.util.logging.impl.LoggerFactory	2.00	3.16	0.01	4.69	0.00	0.00
org.hibernate.search.util.logging.impl.LoggerHelper	3.00	0.17	0.02	100.00	0.00	11.11
de.tudarmstadt.ukp.wikipedia.api.CategoryDescendantsIterator	1.00	3.05	0.01	8.24	0.00	0.00
de.tudarmstadt.ukp.wikipedia.api.CycleHandler	0.00	0.55	0.00	0.00	0.00	0.00
de.tudarmstadt.ukp.wikipedia.api.Page	7.00	1.63	0.04	1.49	6.00	2.67
de.tudarmstadt.ukp.wikipedia.api.PageIterator	21.00	1.94	0.12	50.77	57.14	38.19
de.tudarmstadt.ukp.wikipedia.api.PageQueryIterable	8.00	1.56	0.05	18.57	9.30	0.00
de.tudarmstadt.ukp.wikipedia.api.Title	9.00	3.04	0.03	82.10	79.17	100.00
de.tudarmstadt.ukp.wikipedia.api.WikipediaInfo	0.33	1.10	0.01	0.00	0.00	0.00
org.asynchttpclient.AsyncHttpClient	28.50	6.62	0.19	54.48	39.58	45.38
org.asynchttpclient.AsyncHttpClientConfig	73.00	0.57	0.27	80.12	53.89	35.91
org.asynchttpclient.FluentCaseInsensitiveStringsMap	57.33	0.65	0.18	78.59	71.70	66.50
org.asynchttpclient.FluentStringsMap	52.00	0.55	0.15	78.63	73.45	69.57
org.asynchttpclient.Realm	68.50	0.79	0.24	88.11	57.63	59.38
org.asynchttpclient.RequestBuilderBase	0.00	0.53	0.00	0.00	0.00	0.00
org.asynchttpclient.SimpleAsyncHttpClient	0.00	0.49	0.00	0.00	0.00	0.00
org.scribe.model.OAuthConfig	6.67	0.09	0.03	89.03	91.67	53.70
org.scribe.model.OAuthRequest	6.50	0.26	0.03	100.00	100.00	80.00
org.scribe.model.ParameterList	16.83	0.12	0.06	98.36	96.30	85.51
org.scribe.model.Request	20.33	0.39	0.07	59.34	43.18	34.11
org.scribe.model.Response	1.00	3.03	0.01	0.00	0.00	0.00
org.scribe.model.Token	14.83	0.11	0.06	98.32	92.71	84.85
org.scribe.model.Verifier	1.00	0.06	0.01	100.00	0.00	50.00
Mean	20	1.41	0.07	56.41	46.13	38.54