EvoSuite at the Second Unit Testing Tool Competition

Gordon Fraser[1] and Andrea Arcuri[2]

[1] University of Sheffield
Dep. of Computer Science, Sheffield, UK
gordon.fraser@sheffield.ac.uk,
[2] Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

Abstract. EvoSuite is a mature research prototype implementing a search-based approach to unit test generation for Java classes. It has been successfully run on a variety of different Java projects in the past, and after winning the first instance of the unit testing tool competition at SBST'13, it has also taken part in the second run. This paper reports on the obstacles and challenges encountered during the latter competition.

Keywords: automated unit testing, search-based testing, competition

1 Introduction

The EvoSuite test generation tool is a mature research prototype that automatically generates unit test suites for given Java classes. The first experiments with EvoSuite were reported in [3], and it has since been applied to a range of different projects and domains [4, 9], leading to various improvements over time. In the first unit test competition organised at the SBST'13 workshop [2], EvoSuite obtained the highest score among the participating tools [8].

Besides the obvious research challenge of achieving high coverage, the challenges in building a tool like EvoSuite often lie in practical issues imposed by the Java language, and the nature of real code. For example, often the structure of code is trivial in terms of the complexity of the branching conditions, yet difficult for a testing tool as the code has complex environmental dependencies, such as files and databases [5]. These findings are once more confirmed by the results of the second unit testing tool competition. In this paper, we analyze the major obstacles EvoSuite encountered in this second competition by focusing on classes where the coverage is particularly low.

2 About EvoSuite

EvoSuite is a tool that automatically produces unit test suites for Java classes with the aim to maximize code coverage. As input it requires only the bytecode

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
Input and Output of the tool	
Input	Bytecode of the target class and dependencies
Output	JUnit test cases (version 3 or 4)
Operation	
Interaction	Through the command line
User guidance	manual verification of assertions for functional faults
Source of information	http://www.evosuite.org
Maturity	Mature research prototype, under development
Technology behind the tool	Search-based testing / whole test suite generation
Obtaining the tool and information	
License	GNU General Public License V3
Cost	Open source
Support	None
Empirical evidence about the tool	
Effectiveness and Scalability	See [4, 9].

Table 1. Description of EvoSuite

of the class under test as well as its dependencies. This is provided automatically when using the Eclipse-frontend to interact with EvoSuite, and on the command-line it amounts to setting a classpath and specifying the target class name as a command-line parameter.

EvoSuite applies a search-based approach, where a genetic algorithm optimizes whole test suites towards a chosen target criterion (e.g., branch coverage by default). The advantage of this approach is that it is not negatively affected by infeasible goals and its performance does not depend on the order in which testing goals are considered, as has been demonstrated in the past [9]. To increase the performance further, EvoSuite integrates dynamic symbolic execution (DSE) in a hybrid approach [11], where primitive values (e.g., numbers or strings) are optimized using a constraint solver. However, due to the experimental nature of this feature it was not activated during the competition.

The search-based optimization in EvoSuite results in a set of sequences of method calls that maximizes coverage, yet these sequences are not yet usable as unit test cases — at least not for human consumption. In order to make it easier to understand and use the test cases produced, EvoSuite applies a range of post-processing steps to produce sets of small and concise unit tests. The final step of this post-processing consists of adding test assertions, i.e., statements that check the outcome of the test. As EvoSuite only requires the bytecode as input and such bytecode often does not include formal specifications, the assertions produced will reflect observed behavior, rather than the intended behavior. Consequently, the test cases are intended either as regression tests, or serve as starting point for a human tester, who can manually revise the assertions in order to determine failures [10]. However, EvoSuite is also able to automatically detect certain classes of bugs which are independent of a specification; for example, undeclared exceptions or violations of assertions in the code [6].

Table 1 describes EvoSuite in terms of the template used in the unit testing tool competition.

3 Configuration for the Competition Entry

Even though several new features have been developed for EvoSuite (e.g., DSE integration [11]) since the last competition, we did not include any for the competition entry, as the risk of reducing the score due to immature code would be too high. One notable new feature we did include because it is now enabled by default is support for Java Generics [7]. Besides this, the version of EvoSuite used in the competition is largely the same as in the first round in terms of features, yet has seen many revisions to fix individual problems or bugs.

We used the same configuration as for the first unit testing competition without any changes. This means that EvoSuite was configured to optimize for weak mutation testing, with three minutes time for the search per class. Minimization was deactivated to reduce the test generation time, and EvoSuite was configured to include all possible assertions in the tests (rather than the default of minimizing the assertions using mutation analysis [10]). For all other parameters, EvoSuite was configured to its default parameter settings [1].

4 Results and Problems

Coverage results achieved by EvoSuite are listed in Table 2. On the 63 classes of the benchmark used in the competition, EvoSuite produced an average instruction coverage of 59.9%, average branch coverage of 48.63%, and average mutation score of 43.8%. These results are in line with our expectations based on recent experimentation [4].

For each class it produced on average 12.3 test cases. For the entire benchmark of 63 classes, EvoSuite took on average 4 hours for test generation, which means

No.	Class	Project	LOC	NBD	Time (min)		Coverage		
					Gen	Exec	Instr.	Branch	Mutation
1	SearchException	Hibernate	15	1	3.23	0.00	0.00	0.00	0.00
2	Version	Hibernate	12	1	3.74	0.01	60.00	0.00	0.00
3	BackendFactory	Hibernate	72	2	4.53	0.02	27.71	6.25	40.95
4	FlushLuceneWork	Hibernate	26	2	3.22	0.03	89.74	100.00	75.00
5	OptimizeLuceneWork	Hibernate	26	2	3.22	0.03	89.74	100.00	75.00
6	LoggerFactory	Hibernate	14	1	7.75	0.01	17.41	0.00	0.00
7	LoggerHelper	Hibernate	17	1	3.73	0.01	87.50	0.00	33.33
8	OAuthConfig	Scribe	63	3	3.21	0.05	91.14	100.00	100.00
9	OAuthRequest	Scribe	36	2	3.15	0.03	100.00	100.00	80.00
10	ParameterList	Scribe	95	3	3.19	0.06	99.60	96.83	81.99
11	Request	Scribe	213	2	3.19	0.11	64.45	40.91	37.21
12	Response	Scribe	72	2	3.14	0.01	1.77	0.00	0.00
13	Token	Scribe	64	2	3.17	0.08	100.00	94.64	90.91
14	Verifier	Scribe	15	1	3.13	0.01	100.00	0.00	50.00
15	ExceptionDiagnosis	Twitter4j	73	4	3.27	0.07	98.30	91.27	60.00
16	GeoQuery	Twitter4j	126	2	3.30	0.18	100.00	95.83	81.86
17	OEmbedRequest	Twitter4j	147	2	3.30	0.17	95.47	88.36	57.14
18	Paging	Twitter4j	171	3	3.24	0.14	93.91	91.27	68.76
19	TwitterBaseImpl	Twitter4j	328	5	6.73	0.04	9.16	6.16	3.54
20	TwitterException	Twitter4j	237	4	3.40	0.24	78.15	70.12	17.56
21	TwitterImpl	Twitter4j	1187	3	2.75	0.06	1.11	3.90	0.36
22	AsyncHttpClient	Async Http Client	296	5	9.02	0.00	0.00	0.00	0.00
23	AsyncHttpClientConfig	Async Http Client	621	2	8.71	0.11	39.76	27.62	32.34
24	FluentCaseInsensitiveStringsMap	Async Http Client	292	4	3.35	0.24	74.62	67.25	68.69
25	FluentStringsMap	Async Http Client	240	4	3.31	0.24	72.77	67.11	65.95
26	Realm	Async Http Client	481	3	3.43	0.28	92.98	64.65	70.09
27	RequestBuilderBase	Async Http Client	544	6	8.59	0.09	23.76	21.31	16.83
28	SimpleAsyncHttpClient	Async Http Client	610	3	2.70	0.00	0.00	0.00	0.00
29	AttributeHelper	GData Java Client	344	4	3.45	0.22	85.90	80.49	59.07
30	DateTime	GData Java Client	264	5	3.30	0.19	78.97	67.55	56.59
31	Kind	GData Java Client	189	5	3.36	0.06	53.31	43.83	46.27
32	Link	GData Java Client	190	4	3.79	0.17	62.98	59.46	36.43
33	OtherContent	GData Java Client	179	4	4.37	0.09	46.85	37.01	36.07
34	OutOfLineContent	GData Java Client	102	4	4.13	0.15	83.55	81.12	67.26
35	Source	GData Java Client	324	4	3.47	0.23	42.59	30.84	33.04
36	CharMatcher	Guava	824	4	6.96	0.07	70.86	65.00	47.73
37	Joiner	Guava	222	3	3.30	0.09	84.40	90.06	77.20
38	Objects	Guava	130	4	3.24	0.12	96.20	79.37	96.14
39	Predicates	Guava	379	3	3.31	0.13	36.99	19.35	22.63
40	SmallCharMatcher	Guava	92	4	3.22	0.02	55.77	30.77	26.79
41	Splitter	Guava	266	4	3.24	0.11	93.74	87.91	82.89
42	Suppliers	Guava	172	4	3.18	0.04	30.03	21.43	27.60
43	CategoryDescendantsIterator	JWPL	103	4	3.18	0.01	8.24	0.00	0.00
44	CycleHandler	JWPL	71	4	3.09	0.01	21.41	20.71	27.73
45	Page	JWPL	351	4	2.81	0.01	0.64	2.00	1.90
46	PageIterator	JWPL	182	7	3.14	0.05	46.55	40.82	34.23
47	PageQueryIterable	JWPL	131	3	2.62	0.03	14.93	7.31	0.00
48	Title	JWPL	79	2	3.16	0.03	76.01	76.79	92.86
49	WikipediaInfo	JWPL	222	6	3.19	0.07	10.78	10.29	12.58
50	AbstractLoader	eclipse-cs	77	3	3.30	0.01	77.00	50.00	18.57
51	AnnotationUtility	eclipse-cs	78	4	3.24	0.10	58.11	54.29	46.75
52	AutomaticBean	eclipse-cs	178	4	4.45	0.02	49.45	29.59	13.29
53	FileContents	eclipse-cs	161	4	3.35	0.10	59.57	53.30	50.29
54	FileText	eclipse-cs	184	4	3.43	0.09	48.53	45.60	55.93
55	ScopeUtils	eclipse-cs	189	4	6.15	0.04	48.73	33.29	10.07
56	Utils	eclipse-cs	160	4	3.49	0.13	55.75	78.02	65.31
57	AbstractInstance	JMLL	136	2	3.69	0.08	87.06	67.86	68.08
58	Complex	JMLL	53	1	3.18	0.08	100.00	0.00	47.93
59	DefaultDataset	JMLL	152	3	3.30	0.10	95.04	97.14	60.60
60	DenseInstance	JMLL	155	3	4.07	0.15	98.84	98.66	76.73
61	Fold	JMLL	201	2	3.26	0.16	87.56	82.86	80.36
62	SparseInstance	JMLL	191	3	3.32	0.16	98.17	87.50	72.90
63	ARFFHandler	JMLL	51	6	3.15	0.01	0.00	0.00	0.00

Table 2. EvoSuite results on the benchmark classes

that EvoSuite spent an additional 51 seconds per class on the up-front analysis of the classpath as well as the post-processing after the search.

In the following discussion, we focus on some interesting cases where EvoSuite achieved no or very little ($< 2\%$) coverage or mutation score.

Trivial classes: Class 1 (*SearchException*) is a trivial class with only 15 lines of code, and it seems surprising that EvoSuite achieved 0% coverage here. The reason for this bad result is that the class consists of four constructors that do nothing but calling the constructor of the superclass, and as a result there are no statements for which EvoSuite produced any mutants, and there are no branching instructions. While the standard branch coverage criterion in EvoSuite also enforces that each method is executed at least once, the weak mutation testing criterion we used in the competition did not do so at the time of the competition, and so EvoSuite produced no tests.

Classpath dependent behavior: Class 2 (*Version*) is another trivial class, yet the reason for the bad result do not lie in EvoSuite, but rather the frontend of EvoSuite we built to interface with the competition infrastructure. When setting up the classpath for EvoSuite, our competition frontend unnecessarily included the source directory of the unit under test in the classpath. In the competition setup there are compiled versions of the classes in the source directory as well as a dedicated target path; unfortunately, they differ: Method *getVersionString* returns "[WORKING]" in the class in the source directory, whereas the deployed version of the class is changed to return "[0.4.4-SNAPSHOT]". As EvoSuite's assertions were expecting "[WORKING]", resulting tests failed and were not considered for mutation analysis.

Nondeterministic code: Class 30 (*DateTime*) makes use of the current system time, such that automatically generated assertions may refer to the time of test execution. If this happens, then the tests will fail, and will not be considered for mutation analysis. This is a known issue, and EvoSuite overcomes it by using bytecode instrumentation that replaces nondeterministic calls. However, as the PIT mutation testing tool used in the competition is not able to handle JUnit tests with this kind of instrumentation, we had to deactivate it. EvoSuite does compile and execute tests as a last step to comment out any failing assertions; yet in this case there were assertions dependent on the seconds of the current time, and so this verification step was too quick to notice the failing assertions.

Environmental dependencies: One of the largest problems in unit testing remains the handling of environmental dependencies. For example, most classes in the JWPL project (43–49) depend on a valid instance of a *Wikipedia* class, which in turn depends on a valid *DatabaseConfiguration*. As another example, class 63 (ARFFHandler) consists of only one method (and one wrapper for that method) that receives a file as parameter, and then tries to parse this file.

Complex classes: The benchmark included large classes, most notably class 21 (*TwitterImpl*): This class has 1,187 lines of code, and EvoSuite produced 2,453 mutants for it. That on its own would not be a challenge for EvoSuite; however, test execution on this class turns out to be very slow, and can lead to excessive memory consumption. To prevent out of memory exceptions from occurring and crashing EvoSuite, EvoSuite cancels test generation when the Java garbage collector fails to free sufficient memory. This happened not only in this class, but also several others (19, 23, 43, 44, 45, 47, 49). Running EvoSuite with more memory would have likely resulted in higher coverage on these classes.

5 Conclusions

With an overall score of 205.26, EvoSuite achieved the highest score of all tools in the competition. The score calculated for manual testing is 210.45 — a very close call. This means that EvoSuite is achieving almost human-competitive results in terms of the effectiveness of the resulting test suites. Potentially, just by fixing some of the errors in EvoSuite or its competition frontend the resulting score could be higher than 210, even without adding new features. We are confident that some of the experimental features will further increase this once the required level of robustness has been achieved.

Acknowledgements

This project has been funded by a Google Focused Research Award on "Test Amplification" and the Norwegian Research Council.

References

1. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering (EMSE) pp. 1–30 (2013)
2. Bauersfeld, S., Vos, T., Lakhotia, K., Poulding, S., Condori, N.: Unit testing tool competition. In: International Workshop on Search-Based Software Testing (SBST). pp. 414–420 (2013)
3. Fraser, G., Arcuri, A.: Evolutionary generation of whole test suites. In: International Conference On Quality Software (QSIC). pp. 31–40. IEEE Computer Society (2011)
4. Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE). pp. 178–188 (2012)
5. Fraser, G., Arcuri, A.: EvoSuite: On the challenges of test case generation in the real world (tool paper). In: IEEE International Conference on Software Testing, Verification and Validation (ICST) (2013)
6. Fraser, G., Arcuri, A.: 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empirical Software Engineering (2013), to appear
7. Fraser, G., Arcuri., A.: Automated test generation for java generics. In: Software Quality Days (SWQD) (2013)
8. Fraser, G., Arcuri, A.: Evosuite at the SBST 2013 tool competition. In: International Workshop on Search-Based Software Testing (SBST). pp. 406–409 (2013)
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276–291 (2013)
10. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering (TSE) 28(2), 278–292 (2012)
11. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: IEEE International Symposium on Software Reliability Engineering (ISSRE) (2013)