EvoSuite at the SBST 2013 Tool Competition

Gordon Fraser
University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

Andrea Arcuri
Certus Software V&V Center at Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

***Abstract*—EvoSuite is a mature research prototype that automatically generates unit tests for Java code. This paper summarizes the results and experiences in participating at the unit testing competition held at SBST 2013, where EvoSuite ranked first with a score of 156.95.**

***Keywords*-test case generation; search-based testing; testing classes; search-based software engineering**

I. Introduction

This paper describes the results of applying the EvoSuite test generation tool [4] to the benchmark used in the tool competition at the International Workshop on Search-Based Software Testing (SBST) 2013. Details on the competition and the benchmark can be found in [2]. In this competition, EvoSuite ranked first with a score of 156.95.

II. About EvoSuite

A. Whole Test Suite Generation

EvoSuite automatically generates JUnit test cases for a given class. It requires only the bytecode of the class under test and its dependencies as input. EvoSuite is based on search-based testing, and uses a Genetic Algorithm (GA) to evolve a population of candidate test suites with respect to a choice of code coverage criteria.

This *whole test suite generation* approach [8] is a key novelty of EvoSuite compared to other tools, and represents an effective counter-measure to the problem of infeasible coverage goals. When targeting individual goals one at a time, any resources spent on an infeasible goal are per definition wasted, whereas the search in EvoSuite is not adversely affected by the number of infeasible goals [8]. Our past experiments have shown that this approach leads to significantly higher coverage than targeting individual goals.

The search population in EvoSuite's GA is initialized with small random test suites, which are successively evolved using crossover and mutation. The number of tests and their length is variable, such that the evolution will automatically lead to a suitable size of test suite for the criterion at hand. This variability in length requires bloat-control techniques to counter the problem of population bloat [5]. EvoSuite incorporates seeding strategies [6] that boost coverage, even in the case of string dependencies.

Table I
Classification of the EvoSuite unit test generation tool.

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
Does there exist empirical evidence about	
Effectiveness and Scalability	See [7], [8].

The default coverage criterion used by EvoSuite is branch coverage, but there is also rudimentary support for dataflow and mutation testing, and other coverage criteria could be integrated by encoding them as fitness functions.

Before presenting test cases to the user, EvoSuite applies a range of post-processing steps. Test cases are minimized, constants are inlined, individual values can be minimized, and assertions can be added to the test cases.

B. Efficient Assertion Generation

A key challenge in automated white-box testing is given by the human oracle problem: Unless a test case reveals a generic fault such as an undeclared exception, a tester manually needs to assess the test outcome to decide whether

a fault has been found. In unit testing of object-oriented code, the oracle problem amounts to adding test assertions to the unit tests. Because any given JUnit test case offers a potentially large choice of assertions, EVOSUITE determines which of all the possible assertions for a given test case are good at detecting faults. This is based on mutation analysis [9]: EVOSUITE first determines which assertions can reveal mutations of the bytecode, and then uses a heuristic to calculate a minimal set of assertions to detect all mutants that the test can reveal. However, these assertions reflect the currently implemented behaviour. This means that they can immediately be used for regression testing, but to determine whether there is a fault in the current version of the CUT the developer needs to inspect and verify each of these assertions.

C. Safe Test Execution

To evaluate the fitness of a test suite, the GA in EVOSUITE executes all tests using instrumentation that collects the necessary data. The test execution may have undesired side-effects, for example if the class under test or the sequence of calls EVOSUITE generated to satisfy the dependencies access the filesystem. For example, when running experiments on the 100 randomly selected projects of the SF100 corpus of classes [7] we observed creation of files with random filenames and even deletion of entire directories. To prevent such undesired actions, EVOSUITE uses a custom security manager to restrict test execution to a sandbox environment. Furthermore, to restrict execution of GUI related code that may cause windows and other GUI elements showing up during the search, EVOSUITE is run in *headless* mode, such that no GUI elements will be shown.

III. CONFIGURATION FOR THE COMPETITION ENTRY

EVOSUITE supports several coverage criteria, and many other configuration options. Most of these configuration options are set to reasonable defaults based on our studies on parameter tuning [1], and we argue that in most cases a user should not be required to change low-level parameters that would require an understanding of the underlying techniques. However, it is reasonable to assume that a user will know how long he or she is prepared to wait for the results, and which test criterion the generated test cases should satisfy. As branch coverage may be a weak criterion, in particular if classes consist of many small methods with trivial control flow, we chose *weak mutation testing* as target criterion. EVOSUITE uses bytecode instrumentation to create a meta-mutant for the class under test, and can then activate individual mutants using a parameter. A mutant is weakly killed if it leads to an immediate state change. Furthermore, we arbitrarily chose three minutes as timeout for the search. This is based on our past experience, where 10 minutes for a class in all but very complex examples is more than enough time, whereas two minutes for non-trivial classes with many mutants may easily be insufficient time. Given more than three minutes would likely have resulted in higher coverage.

Considering that the score calculation in the SBST competition does not directly include the test suite size or length (only in terms of execution time), we deactivated the test minimization in EVOSUITE, as it will take significantly more time to minimize a test suite than can be gained during execution. Furthermore, for the same reason we deactivated assertion filtering, such that the resulting test cases include all possible assertions.

IV. BENCHMARK RESULTS

The results of EVOSUITE on the benchmark classes are listed in Table II. On average, EVOSUITE achieved 61.4% line coverage, 57.6% branch coverage[1], and 13.3% mutation score. On average, EVOSUITE produced 9 tests per class, and it took an average of 186 per class to do so (with EVOSUITE configured to 3 minutes per class).

While the results on code coverage are in line with our expectations from past experiments (e.g., [7]), the mutation scores are surprisingly low. A closer inspection of this unexpected result revealed several issues in assertion generation, which are well known in principle. The common problem is these cases is that the assertions produced by EVOSUITE do not hold upon test re-execution, and tests with failing assertions are excluded from mutation analysis.

A. Low Mutation Scores

The first issue becomes apparent when considering the overall low mutation scores on Joda Time classes, which are contrary to previous results (e.g., [9]). One reason for this is that assertions in Joda Time tend to reflect the time during test generation. For example, an instance of a time object based on the current system time will include this time in its `toString` representation, and any successive test runs will fail, unless they happen to be executed at the same time. EVOSUITE in theory overcomes this issue by instrumenting the bytecode such that all calls to `System.currentTimeMillis` and related methods are replaced with custom calls that allow for deterministic test execution. However, to enable this during JUnit test execution requires bytecode instrumentation also at runtime. As this was not supported by the SBST contest infrastructure, we deactivated this feature in EVOSUITE.

The second issue we observed is related to static initializers, and is an issue that is long known in test generation [3]. To make sure that test cases are independent, all static initializers would need to be reset before every single test execution. As this poses a significant overhead, EVOSUITE creates executable copies of the static initializer for each class, removing assignments of `final` fields. However, we deactivated this feature in EVOSUITE for the competition for

[1]Using Cobertura's definition of branch coverage, which only counts conditional statements, not edges in the CFG.

performance reasons, which possibly led to higher coverage, but apparently to lower mutation scores.

B. Classes with Low Coverage

Besides the generally low mutation scores, we see nine classes on which EVOSUITE achieved 0% coverage in Table II. The classes *XlsSheetIterator* and *XlsxSheetIterator* both take a *URL* as input, which EVOSUITE produced using calls like *ClassLoader.getSystemResource("")*. However, the resulting URL encodes the current directory during test generation and results in assertions like

```
assertEquals("/home/evosuite/", uRL0.getPath());
```

These assertions fail when executed during analysis in a different directory, and consequently all tests for these classes fail even before an instance of the target class has been produced, thus leading to 0% coverage. This problem would not have occurred if we had not deactivated assertion minimization – the problematic assertions are unrelated to the class under test, and would have immediately been removed in normal operation.

The classes *net.sourceforge.barbecue.Barcode* and *LinearBarcode* are both GUI components extending *java.awt.Component* and cannot be initialised in EVOSUITE in headless mode and with activated sandbox.

org.apache.commons.lang3.BooleanUtils revealed a bug in EVOSUITE related to multi-dimensional arrays, which unfortunately led to EVOSUITE crashing on this class in all runs, even though the class itself would be easily covered by EVOSUITE. Similarly, the problem on *org.joda.time.DurationField* is related to a bug in EVOSUITE in how it handles test generation for abstract classes.

org.joda.time.field.MillisDurationField is a tricky case: The class has only a private constructor and thus cannot be constructed by EVOSUITE directly. There is one public static instance of the class, but it is declared as its supertype:

```
public static final DurationField INSTANCE = new
    MillisDurationField();
```

If EVOSUITE would be left running long enough, then eventually it would also try and assign this *INSTANCE* object, discovering that it actually is a *MillisDurationField* instance. However, this did not happen in the three minutes given for the competition.

Finally, the classes *BuddhistChronology*, *GregorianChronology*, and *DateTimeFormatterBuilder* are working fine in our own experiments, so the reason for the 0% coverage in the competition is currently not clear to us; possibly this is related to compile errors or failures in the produced JUnit tests. A further possible contributing factor is that tests requiring the EVOSUITE security manager (e.g., when trying to access a file), then the resulting JUnit test case spawns a new thread to execute the code using the EVOSUITE security manager. As this construct is not supported by Javalanche, we automatically removed all such tests, thus potentially reducing the coverage.

We also note that *org.joda.time.Chronology* achieves low coverage (only 10.5% line coverage). This is an interesting case, as much of this class is contained in methods tagged as deprecated. By default, EVOSUITE does not attempt to cover deprecated code, although deprecated code seems to be considered for coverage and mutation analysis. However, EVOSUITE can be configured to also cover deprecated code.

V. CONCLUSIONS

The road to practically usable unit test generators is long, and we are by far not there yet. The SBST competition has provided an invaluable incentive to work on the robustness of EVOSUITE, which in writing research papers is usually not rewarded. The participation in this competition has brought EVOSUITE a big step closer to being useful in practice, and it has helped us to identify areas where future work is necessary to improve EVOSUITE further.

To learn more about EVOSUITE, visit our Web site:

`http://www.evosuite.org`

Acknowledgments. This project has been funded a Google Focused Research Award on "Test Amplification". Andrea Arcuri is funded by the Norwegian Research Council.

REFERENCES

[1] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering (SSBSE)*, 2011, pp. 33–47.

[2] S. Bauersfeld, T. Vos, K. Lakhotiay, S. Poulding, and N. Condori, "Unit testing tool competition," in *International Workshop on Search-Based Software Testing (SBST)*, 2013.

[3] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, pp. 1025–1050, 2004.

[4] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software." in *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011, pp. 416–419.

[5] ——, "It is not the length that matters, it is how you control it," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 150 – 159.

[6] ——, "The seed is strong: Seeding strategies in search-based software testing," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012, pp. 121–130.

[7] ——, "Sound empirical evidence in software testing," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012, pp. 178–188.

[8] ——, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[9] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering (TSE)*, vol. 28, no. 2, pp. 278–292, 2012.

Table II
DETAILED RESULTS OF EVOSUITE ON THE SBST BENCHMARK CLASSES.

Class	Generation Time	Execution Time	Tests	Line Coverage	Branch Coverage	Mutation Score
com.googlecode.sqlsheet.stream.XlsSheetIterator	217.65	0.00	1.00	0.0000	0.0000	0.0000
com.googlecode.sqlsheet.stream.XlsxSheetIterator	216.29	0.00	1.00	0.0000	0.0000	0.0000
net.sourceforge.barbecue.Barcode	195.05	0.00	1.00	0.0000	0.0000	0.0000
net.sourceforge.barbecue.BlankModule	193.12	0.26	1.00	1.0000	1.0000	0.1825
net.sourceforge.barbecue.CompositeModule	194.58	0.08	2.17	1.0000	1.0000	0.3527
net.sourceforge.barbecue.Module	192.90	0.03	3.67	0.8762	0.8611	0.2885
net.sourceforge.barbecue.Modulo10	186.83	0.00	1.33	0.8667	1.0000	0.7971
net.sourceforge.barbecue.SeparatorModule	194.15	0.05	1.33	1.0000	1.0000	0.2012
net.sourceforge.barbecue.env.DefaultEnvironment	188.56	0.14	1.00	1.0000	1.0000	1.0000
net.sourceforge.barbecue.env.EnvironmentFactory	189.45	0.01	1.83	0.7593	0.6667	0.2436
net.sourceforge.barbecue.env.HeadlessEnvironment	186.99	0.00	1.00	1.0000	1.0000	1.0000
net.sourceforge.barbecue.linear.LinearBarcode	194.73	0.00	1.00	0.0000	0.0000	0.0000
net.sourceforge.barbecue.linear.codabar.CodabarBarcode	194.15	0.01	1.00	0.2738	0.1319	0.0000
net.sourceforge.barbecue.linear.code128.Code128Barcode	194.80	0.01	2.00	0.2548	0.0308	0.0000
net.sourceforge.barbecue.linear.code128.ModuleFactory	196.60	0.01	1.83	0.9944	0.9000	0.0008
net.sourceforge.barbecue.linear.code39.Code39Barcode	193.88	0.02	3.00	0.5109	0.5625	0.0000
net.sourceforge.barbecue.linear.ean.UCCEAN128Barcode	194.93	0.03	5.00	0.3591	0.1404	0.0000
net.sourceforge.barbecue.linear.twoOfFive.Int2of5Barcode	193.77	0.01	2.00	0.2667	0.5000	0.0000
net.sourceforge.barbecue.linear.twoOfFive.Std2of5Barcode	193.61	0.01	2.00	0.4136	0.4833	0.0139
net.sourceforge.barbecue.output.GraphicsOutput	192.90	0.15	3.33	0.8444	0.6333	0.1411
org.apache.commons.lang3.ArrayUtils	200.56	0.01	12.33	0.1097	0.0805	0.0000
org.apache.commons.lang3.BooleanUtils	12.55	0.00	0.00	0.0000	0.0000	0.0000
org.apache.commons.lang3.CharRange	187.10	0.00	13.67	0.9778	0.9400	0.4729
org.apache.commons.lang3.math.Fraction	195.11	0.01	35.67	0.9497	0.8861	0.1494
org.apache.commons.lang3.math.NumberUtils	190.04	0.01	8.83	0.1422	0.1183	0.0000
org.apache.lucene.util.FixedBitSet	192.46	0.03	41.50	0.9158	0.5173	0.0000
org.apache.lucene.util.WeakIdentityMap	186.98	0.01	4.83	0.9032	0.5000	0.0000
org.joda.time.Chronology	190.47	0.07	1.00	0.1053	1.0000	0.0000
org.joda.time.DateTimeComparator	191.00	0.06	9.33	0.9107	0.8225	0.0040
org.joda.time.DateTimeFieldType	189.80	0.07	10.83	1.0000	1.0000	0.0075
org.joda.time.DateTimeUtils	191.04	0.08	9.33	0.5653	0.4912	0.0040
org.joda.time.DateTimeZone	221.39	0.08	21.83	0.5161	0.4589	0.0052
org.joda.time.Days	191.94	0.17	18.67	0.8680	0.8632	0.0045
org.joda.time.DurationField	186.85	0.00	1.00	0.0000	0.0000	0.0028
org.joda.time.DurationFieldType	189.61	0.04	4.83	0.9944	1.0000	0.0021
org.joda.time.Hours	190.23	0.09	18.17	0.7764	0.7500	0.0040
org.joda.time.IllegalFieldValueException	187.27	0.01	11.50	0.4474	0.5625	0.0035
org.joda.time.Minutes	190.67	0.09	16.50	0.8213	0.8238	0.0037
org.joda.time.Months	190.17	0.08	20.00	0.8586	0.8485	0.0049
org.joda.time.MutableDateTime	195.96	0.33	56.50	0.3057	0.2027	0.0000
org.joda.time.PeriodType	192.25	0.05	13.00	0.4886	0.3494	0.0050
org.joda.time.Seconds	190.79	0.09	17.33	0.8478	0.8333	0.0042
org.joda.time.Years	190.04	0.08	15.17	0.8497	0.8571	0.0055
org.joda.time.chrono.BuddhistChronology	6.07	0.00	0.00	0.0000	0.0000	0.0000
org.joda.time.chrono.GJChronology	216.55	0.07	21.33	0.7836	0.5909	0.0205
org.joda.time.chrono.GregorianChronology	6.14	0.00	0.00	0.0000	0.0000	0.0000
org.joda.time.chrono.ISOChronology	189.47	0.04	5.33	0.8444	0.5833	0.0186
org.joda.time.chrono.LenientChronology	190.50	0.04	2.00	0.4967	0.1146	0.0165
org.joda.time.convert.CalendarConverter	191.64	0.05	3.33	0.5667	0.4500	0.4086
org.joda.time.convert.ConverterManager	189.47	0.02	12.33	0.6492	0.4624	0.1233
org.joda.time.convert.ConverterSet	190.54	0.05	9.00	0.6709	0.5979	0.4240
org.joda.time.convert.DateConverter	189.46	0.01	1.00	0.6667	1.0000	0.2353
org.joda.time.convert.LongConverter	189.43	0.02	1.33	0.9167	1.0000	0.4815
org.joda.time.convert.NullConverter	189.47	0.05	1.00	0.9091	1.0000	0.6439
org.joda.time.convert.ReadableDurationConverter	189.92	0.04	1.33	0.8889	0.8333	0.3125
org.joda.time.convert.ReadableInstantConverter	190.91	0.05	3.50	0.4537	0.2292	0.2262
org.joda.time.convert.ReadableIntervalConverter	190.57	0.06	4.33	0.6429	0.5278	0.5808
org.joda.time.convert.ReadablePartialConverter	190.73	0.05	1.83	0.5104	0.3750	0.3214
org.joda.time.convert.ReadablePeriodConverter	189.74	0.05	1.33	1.0000	1.0000	0.5000
org.joda.time.convert.StringConverter	191.44	0.07	16.17	0.5056	0.4571	0.0883
org.joda.time.field.BaseDateTimeField	194.84	0.07	17.50	0.8363	0.7986	0.0753
org.joda.time.field.FieldUtils	189.50	0.06	25.83	0.9262	0.9195	0.2500
org.joda.time.field.MillisDurationField	186.89	0.00	1.00	0.0000	0.0000	0.0394
org.joda.time.field.OffsetDateTimeField	188.86	0.05	3.17	0.4402	0.6250	0.0681
org.joda.time.field.PreciseDateTimeField	189.55	0.04	1.00	0.4683	0.3889	0.0124
org.joda.time.field.PreciseDurationDateTimeField	191.30	0.05	8.17	0.9383	0.8500	0.0771
org.joda.time.field.PreciseDurationField	186.85	0.00	1.00	0.0417	0.0000	0.0672
org.joda.time.field.ScaledDurationField	187.39	0.00	5.67	0.6579	0.5000	0.0717
org.joda.time.field.UnsupportedDateTimeField	193.01	0.07	38.83	0.7295	0.9306	0.0627
org.joda.time.format.DateTimeFormat	196.14	0.06	25.83	0.7200	0.5641	0.0142
org.joda.time.format.DateTimeFormatter	200.64	0.07	21.50	0.7480	0.6348	0.0205
org.joda.time.format.DateTimeFormatterBuilder	229.80	0.00	0.00	0.0000	0.0000	0.0000
org.joda.time.format.ISODateTimeFormat	191.23	0.04	18.33	0.7710	0.5076	0.0213
org.joda.time.format.ISOPeriodFormat	190.74	0.09	3.67	1.0000	1.0000	0.0341
org.joda.time.format.PeriodFormat	189.58	0.03	1.33	1.0000	1.0000	0.0293
org.joda.time.format.PeriodFormatter	191.23	0.05	8.33	0.9722	0.9318	0.0335
org.joda.time.format.PeriodFormatterBuilder	207.49	0.07	31.67	0.7805	0.6519	0.0393
Average	186.3	0.05	9.06	0.61	0.58	0.13