EvoSuite at the SBST 2018 Tool Competition

Gordon Fraser
Chair of Software Engineering II, University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

José Miguel Rojas
Department of Informatics, University of Leicester
Leicester, United Kingdom
j.rojas@leicester.ac.uk

Andrea Arcuri
Westerdals Oslo ACT, Norway and University of Luxembourg, Luxembourg
arcand@westerdals.no

ABSTRACT

EvoSuite is a search-based tool that automatically generates executable unit tests for Java code (JUnit tests). This paper summarises the results and experiences of EvoSuite's participation at the sixth unit testing competition at SBST 2018, where EvoSuite achieved the highest overall score (687 points) for the fifth time in six editions of the competition.

ACM Reference Format:
Gordon Fraser, José Miguel Rojas, and Andrea Arcuri. 2018. EvoSuite at the SBST 2018 Tool Competition. In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3194718.3194729

1 INTRODUCTION

The annual unit test generation competition aims to drive and evaluate progress on unit test generation tools. In the 6th instance of the competition at the International Workshop on Search-Based Software Testing (SBST) 2018, several JUnit test generation tools competed on a set of 59 open-source Java classes, and existing open-source test suites were used as baseline. This paper describes the results obtained by the EvoSuite test generation tool [7] in this competition. Details about the procedure of the competition, the technical framework, and the benchmark classes can be found in [22]. In this competition, EvoSuite achieved an overall score of 687, which was the highest among the competing and baseline tools.

2 ABOUT EVOSUITE

EvoSuite [7] is a search-based tool [11] that uses a genetic algorithm to automatically generate test suites for Java classes. Given the name of a target class and the full Java classpath (i.e., where to find the compiled bytecode of the class under test and all its dependencies), EvoSuite automatically produces a set of JUnit test cases aimed at maximising code coverage. EvoSuite can be used on the command line, or through plugins for popular development tools such as IntelliJ, Eclipse, or Maven [2].

The underlying genetic algorithm uses test suites as representation (chromosomes). Each test suite consists of a variable number of test cases, each of which is represented as a variable-length sequence of Java statements (e.g., calls on the class under test). A population of randomly generated individuals is evolved using suitable search operators (e.g., selection, crossover and mutation), such that iteratively better solutions with respect to the optimisation target are produced. The optimisation target is to maximise code coverage. To achieve this, the fitness function uses standard heuristics such as the branch distance; see [11] for more details. EvoSuite can be configured to optimise for multiple coverage criteria at the same time, and the default configuration combines branch coverage with mutation testing [13] and other basic criteria [19]. Once the search is completed, EvoSuite applies various optimisations to improve the readability of the generated tests. For example, tests

Table 1: 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 4 test cases
Operation	
Interaction	Through the command line, and plugins for IntelliJ, Maven and Eclipse
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	Lesser GPL V.3
Cost	Open source
Support	None
Does there exist empirical evidence about	
Effectiveness and Scalability	See [11, 12]

SBST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194729

are minimised, and a minimised set of effective test assertions is selected using mutation analysis [17]. For more details on the tool and its abilities we refer to [7, 8].

The effectiveness of EvoSuite has been evaluated on open source as well as industrial software in terms of code coverage [12, 21], fault finding effectiveness [1, 24], and effects on developer productivity [16, 20] and software maintenance [25].

EvoSuite has a longstanding record of success at the unit testing tool competition, having ranked second in the third edition of the competition [14] and first in all the other editions [9, 10, 15, 18].

3 COMPETITION SETUP

The configuration of EvoSuite for the 2018 competition is largely based on its default values, since these have been tuned extensively [4]. We used the default set of coverage criteria [19] (e.g., line coverage, branch coverage, branch coverage by direct method invocation, weak mutation testing, output coverage, exception coverage). EvoSuite uses an archive of solutions [21] to keep the search focused on uncovered goals, iteratively discarding covered goals and storing the tests that covered them. After a certain percentage of the search budget has passed and with a certain probability, EvoSuite starts using mock objects (relying on Mockito) instead of actual class instances [3] for dependencies; only branches that cannot be covered without mocks lead to tests with mock objects. As in the previous instance of the competition, we continue to use frequency-based weighted constants for seeding [23] and support Java Enterprise Edition features [5]. EvoSuite is actively maintained, therefore several bug fixes and minor improvements have been applied since the last instance of the competition, in particular in relation to non-determinism and flaky tests [6], as well as new types of test assertions (e.g., related to arrays and standard container classes).

Like in previous instances of the competition, we enabled the post-processing step of test minimisation—not for efficiency reasons, but because minimised tests are less likely to break. To reduce the overall time of test generation we included all assertions rather than filtering them with mutation analysis [17], which is computationally expensive. The use of all assertions has a negative impact on readability, but this is not evaluated as part of the SBST contest.

Four time budgets were used to call each tool: 10, 60, 120 and 240 seconds. We used the same strategy used in previous competition (e.g., [15]) to distribute the overall time budget onto the different phases of EvoSuite (e.g., initialisation, search, minimisation, assertion generation, compilation check, removal of flaky tests). That is, 50% of the time was allocated to the search, and the rest was distributed equally to the remaining phases.

4 BENCHMARK RESULTS

Overall Results. Table 2 lists the branch coverage and mutation analysis results achieved by EvoSuite on all benchmark classes in the contest. Coverage and mutation scores are generally in the expected ranges, with clear overall increases for higher time budgets. With the highest time budget of 240*s*, the average branch coverage achieved was 61.6%, lower that last year's 64.6% for the same time budget, but the average mutation score was 53.5%, higher than last year's 49.6% for the same time budget.

Manually Written Tests. This year, the contest also included manually written test suites as baseline (for 49 out of 59 benchmarks). The results indicate that there is no significant difference in branch coverage between EvoSuite-generated test suites (avg. 55.8%) and manually written ones (avg. 46.4%), with Vargha-Delaney's $A_{12} = 0.59$ and $p = 0.120$. In terms of mutation scores, surprisingly, the results show that automatically generated assertions are competitive with manually written ones: the average EvoSuite mutation score for these 49 benchmarks is 47.4% and the average mutation score obtained by developer-written test suites is 34.9% ($A_{12} = 0.62$ and $p = 0.033$).

Flaky Tests. EvoSuite generated 0.8 flaky tests per run on average, much lower than the number of flaky tests produced by the competing and baseline tools (1.5 by T3, 13.5 by jTExpert, and 16.2 by Randoop). We attribute these relatively positive results—also consistent with previous years' results—to the way EvoSuite handles execution environments during test generation (e.g., controlling the static state of the class under test, mocking interactions with the file system, system calls like `System.in` and `System.currentTimeMillis`, etc.) [6]. EvoSuite provides deterministic replacement classes for many classes with known non-determinism, (e.g., those related to date and time), and in particular the FASTJSON benchmarks made heavy use of these. For example, multiple calendar systems are necessary (e.g., classes `HijrahDate`, `MinguoDate`, `JapaneseDate`, `ThaiBuddhistDate`), and EvoSuite instantiated its own replacement versions of these classes 719 times altogether, thus potentially avoiding many flaky tests. Overall, no flaky tests were generated for the vast majority of subjects in the competition.

Amongst those benchmarks for which EvoSuite struggled with flakiness, FASTJSON-4 is a notable example: in the worst case, 90 flaky tests were generated in one single run. An analysis of the execution logs indicates several possible sources for this flakiness: a) a non-final static constant that is not properly reset in between executions; b) a possibly incorrect handling of enum types; and c) a global property not being reset in between executions. Considering that there were still remaining flaky tests, further investigation will be needed to confirm these conjectures and act upon them.

Mocking. As mentioned in the previous section, EvoSuite uses private API access and functional mocking [3]. Overall, in all tests generated in this competition, 1,339 private fields were set and 2,409 private methods were invoked using reflection, and 15,369 Mockito mock instances were created and configured using another 9,553 statements (e.g., using the doReturn method). Considering the 1,804,932 overall test statements, this means that only 1.4% of the tests were devoted to mocking. Amongst the 331 distinct classes for which mocks were generated at least once there are some classes for which EvoSuite can instantiate regular objects (e.g., `List`). This suggests that EvoSuite sometimes struggles to generate and set up complex objects correctly, and there is potential for future improvements.

Problematic benchmarks. EvoSuite failed to produce any test suites for benchmarks DUBBO-2 and WEBMAGIC-4, and generally struggled with most REDISSON benchmarks (highlighted in grey in Table 2). The former two benchmarks had missing dependencies

Table 2: Detailed results of EvoSuite on the SBST benchmark classes.

Benchmark	Java Class	Branch Coverage				Mutation Score			
		10s	60s	120s	240s	10s	60s	120s	240s
DUBBO-10	com.alibaba.dubbo.common.bytecode.Wrapper	24.1%	24.5%	25.3%	28.2%	11.5%	20.9%	21.3%	20.9%
DUBBO-2	com.alibaba.dubbo.common.utils.ReflectUtils	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
DUBBO-3	com.alibaba.dubbo.common.utils.StringUtils	87.1%	94.6%	96.8%	97.1%	93.9%	87.3%	81.8%	85.8%
DUBBO-4	com.alibaba.dubbo.common.utils.ClassHelper	92.4%	94.4%	95.8%	95.1%	66.7%	91.4%	93.8%	92.6%
DUBBO-5	com.alibaba.dubbo.common.io.UnsafeByteArrayOutputStream	94.4%	94.4%	94.4%	94.4%	68.3%	77.8%	79.4%	78.9%
DUBBO-6	com.alibaba.dubbo.common.utils.CompatibleTypeUtils	4.7%	3.9%	4.7%	4.7%	3.1%	2.9%	3.1%	2.3%
DUBBO-7	com.alibaba.dubbo.common.beanutil.JavaBeanDescriptor	80.2%	92.4%	95.1%	97.9%	77.9%	96.2%	97.4%	99.0%
DUBBO-8	com.alibaba.dubbo.common.Parameters	61.1%	66.1%	77.6%	77.6%	49.5%	74.8%	76.9%	78.4%
DUBBO-9	com.alibaba.dubbo.common.io.Bytes	67.8%	71.9%	79.5%	77.1%	38.1%	65.2%	71.8%	70.4%
FASTJSON-1	com.alibaba.fastjson.parser.JSONLexerBase	-	15.9%	15.2%	20.4%	-	3.7%	6.8%	10.2%
FASTJSON-10	com.alibaba.fastjson.serializer.StringCodec	40.0%	44.2%	46.7%	50.0%	21.8%	42.0%	44.3%	45.4%
FASTJSON-2	com.alibaba.fastjson.util.TypeUtils	6.8%	28.7%	31.2%	40.4%	8.6%	30.3%	32.9%	43.5%
FASTJSON-3	com.alibaba.fastjson.parser.DefaultJSONParser	5.3%	9.3%	16.2%	18.5%	2.9%	6.8%	13.4%	16.9%
FASTJSON-4	com.alibaba.fastjson.JSONArray	20.0%	32.2%	21.1%	30.6%	34.2%	52.3%	38.7%	54.0%
FASTJSON-5	com.alibaba.fastjson.util.JavaBeanInfo	0.0%	15.7%	17.6%	21.0%	0.0%	12.2%	15.2%	17.6%
FASTJSON-6	com.alibaba.fastjson.serializer.DateCodec	18.5%	39.2%	48.0%	34.5%	8.8%	20.9%	25.9%	19.1%
FASTJSON-7	com.alibaba.fastjson.util.IOUtils	52.4%	63.6%	66.6%	71.6%	18.9%	40.2%	46.8%	49.8%
FASTJSON-8	com.alibaba.fastjson.parser.JSONReaderScanner	69.6%	70.1%	73.1%	77.3%	74.7%	70.6%	71.0%	73.1%
FASTJSON-9	com.alibaba.fastjson.util.ASMUtils	38.5%	39.1%	39.1%	39.1%	27.6%	38.6%	39.0%	39.5%
JSOUP-1	org.jsoup.parser.TokenQueue	85.8%	90.0%	90.6%	94.9%	59.3%	81.8%	85.6%	87.5%
JSOUP-2	org.jsoup.select.QueryParser	49.8%	24.9%	21.9%	55.8%	36.4%	12.8%	11.3%	40.8%
JSOUP-3	org.jsoup.helper.DataUtil	39.6%	38.7%	44.8%	52.0%	28.3%	28.1%	41.0%	49.8%
JSOUP-4	org.jsoup.parser.Parser	48.3%	95.0%	98.3%	98.0%	31.7%	80.6%	85.6%	87.3%
JSOUP-5	org.jsoup.parser.Tokeniser	35.6%	53.3%	54.8%	56.5%	23.4%	44.0%	47.3%	50.0%
OKIO-1	okio.Buffer	44.6%	42.1%	58.3%	56.1%	9.8%	21.0%	34.1%	23.9%
OKIO-10	okio.Timeout	83.3%	80.0%	85.6%	83.3%	76.8%	78.1%	82.5%	80.7%
OKIO-2	okio.ByteString	81.7%	72.3%	89.8%	95.0%	40.9%	59.2%	84.5%	78.7%
OKIO-3	okio.SegmentedByteString	56.1%	72.5%	88.9%	87.8%	28.0%	61.4%	67.6%	69.2%
OKIO-4	okio.RealBufferedSource	12.7%	27.4%	40.1%	49.9%	12.3%	17.9%	30.5%	51.1%
OKIO-5	okio.RealBufferedSink	61.4%	79.5%	89.1%	93.4%	45.7%	76.9%	86.5%	88.4%
OKIO-6	okio.Okio	73.2%	81.5%	82.1%	85.7%	80.6%	83.3%	83.9%	86.7%
OKIO-7	okio.Segment	85.4%	96.5%	97.2%	97.2%	54.5%	77.6%	78.0%	79.3%
OKIO-8	okio.AsyncTimeout	40.9%	58.0%	55.7%	59.5%	20.6%	46.1%	58.2%	60.5%
OKIO-9	okio.Utf8	42.3%	42.3%	42.3%	42.3%	25.6%	35.9%	35.9%	35.9%
REDISSON-1	org.redisson.RedissonSetMultimapValues	13.3%	0.0%	-	-	1.1%	0.0%	-	-
REDISSON-10	org.redisson.config.Config	63.8%	0.0%	97.8%	-	17.1%	0.0%	20.9%	-
REDISSON-2	org.redisson.RedissonQueue	39.6%	25.0%	-	-	0.0%	0.0%	-	-
REDISSON-3	org.redisson.jcache.JCacheManager	-	0.0%	0.0%	0.0%	-	0.0%	0.0%	0.0%
REDISSON-4	org.redisson.RedissonSetMultimap	0.0%	-	0.0%	-	0.0%	-	0.0%	-
REDISSON-5	org.redisson.RedissonBloomFilter	0.0%	0.0%	-	-	0.0%	0.0%	-	-
REDISSON-6	org.redisson.reactive.RedissonMapReactive	0.0%	0.0%	-	-	0.0%	0.0%	-	-
REDISSON-7	org.redisson.RedissonBaseMapIterator	-	-	-	-	-	-	-	-
REDISSON-8	org.redisson.jcache.JCachingProvider	58.3%	61.1%	-	-	80.6%	79.0%	-	-
REDISSON-9	org.redisson.cluster.ClusterPartition	68.2%	92.4%	99.2%	99.2%	30.1%	69.9%	75.5%	74.5%
WEBMAGIC-1	us.codecraft.webmagic.model.PageModelExtractor	10.3%	17.9%	21.5%	22.4%	24.6%	34.0%	41.0%	42.0%
WEBMAGIC-2	us.codecraft.webmagic.Spider	25.9%	36.1%	40.7%	41.5%	52.1%	60.8%	60.2%	53.6%
WEBMAGIC-3	us.codecraft.webmagic.Site	55.2%	65.4%	79.9%	84.1%	33.9%	90.7%	79.5%	83.2%
WEBMAGIC-4	us.codecraft.webmagic.Page	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
WEBMAGIC-5	us.codecraft.webmagic.utils.DoubleKeyMap	50.0%	26.0%	0.0%	0.0%	68.8%	30.0%	0.0%	0.0%
ZXING-1	com.google.zxing.qrcode.detector.FinderPatternFinder	29.8%	32.8%	36.3%	39.3%	4.4%	16.2%	22.8%	16.7%
ZXING-10	com.google.zxing.qrcode.encoder.Encoder	68.4%	79.6%	80.4%	81.7%	54.1%	73.7%	76.3%	73.2%
ZXING-2	com.google.zxing.pdf417.decoder.PDF417ScanningDecoder	24.3%	25.6%	27.5%	30.3%	6.8%	17.9%	27.1%	32.9%
ZXING-3	com.google.zxing.common.StringUtils	87.6%	90.8%	91.5%	91.7%	3.9%	65.5%	65.9%	68.6%
ZXING-4	com.google.zxing.client.result.ResultParser	86.2%	81.9%	89.2%	95.3%	82.0%	75.7%	83.2%	84.3%
ZXING-5	com.google.zxing.qrcode.encoder.MatrixUtil	46.0%	77.9%	80.0%	89.0%	53.4%	53.5%	56.8%	52.6%
ZXING-6	com.google.zxing.datamatrix.decoder.BitMatrixParser	36.3%	56.5%	60.5%	61.5%	45.3%	33.1%	34.4%	37.7%
ZXING-7	com.google.zxing.pdf417.decoder.ec.ModulusPoly	96.1%	97.4%	96.5%	97.4%	72.7%	87.0%	92.4%	92.4%
ZXING-8	com.google.zxing.maxicode.decoder.DecodedBitStreamParser	54.4%	67.2%	77.8%	82.2%	31.1%	36.9%	42.3%	46.8%
ZXING-9	com.google.zxing.oned.CodaBarWriter	89.8%	91.5%	92.5%	96.9%	52.5%	57.1%	61.0%	63.5%
Average		47.8%	53.3%	58.9%	61.6%	34.8%	46.6%	51.4%	53.5%

in the competition setup [22], which explains why EvoSuite was incapable of generating any test: When a dependency of the class under test is missing, then EvoSuite intentionally aborts with an error to inform the user of the configuration error. While this is desirable behavior during regular usage, arguably in the scope of the competition EvoSuite could try to ignore such errors and try to generate tests nevertheless, as it might still be possible to cover code even with some missing dependencies.

For the REDISSON benchmarks, most executions ended with a timeout; unfortunately, we have not been able to reproduce this behaviour locally, hence we can only conjecture that the timeouts (reported during the initialisation phase) are due to the large number of dependency classes (>7000) that need to be loaded for these benchmarks to start off the test generation.

5 CONCLUSIONS

This paper reports on the participation of the EvoSuite test generation tool in the 6th SBST Java Unit Testing Tool Contest. With an overall score of 687 points, EvoSuite achieved the highest score of all tools in the competition.

To learn more about EvoSuite, visit our Web site:

`http://www.evosuite.org`

Acknowledgments: Many thanks to all the contributors to EvoSuite. This project has been funded by the EPSRC project "GREATEST" (EP/N023978/1), and by the National Research Fund, Luxembourg (FNR/P10/03).

REFERENCES

[1] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *ACM/IEEE Int. Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 263–272.

[2] A. Arcuri, J. Campos, and G. Fraser, "Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2016, pp. 401–408.

[3] A. Arcuri, G. Fraser, and R. Just, "Private api access and functional mocking in automated unit test generation," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 126–137.

[4] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Software Engineering (EMSE)*, pp. 1–30, 2013.

[5] ——, "Java enterprise edition support in search-based junit test generation," in *Int. Symposium on Search Based Software Engineering*. Springer, 2016, pp. 3–17.

[6] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 79–90.

[7] 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.

[8] ——, "EvoSuite: On the challenges of test case generation in the real world (tool paper)," in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2013, pp. 362–369.

[9] ——, "EvoSuite at the SBST 2013 tool competition," in *Int. Workshop on Search-Based Software Testing (SBST)*, 2013, pp. 406–409.

[10] ——, "EvoSuite at the second unit testing tool competition." in *Fittest Workshop*. Springer, 2013, pp. 95–100.

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

[12] ——, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 8:1–8:42, 2014.

[13] ——, "Achieving scalable mutation-based generation of whole test suites." *Empirical Software Engineering (EMSE)*, vol. 20, no. 3, pp. 783–812, 2015.

[14] ——, "EvoSuite at the SBST 2015 tool competition," in *Int. Workshop on Search-Based Software Testing (SBST)*. IEEE Press, 2015, pp. 25–27.

[15] ——, "EvoSuite at the SBST 2016 tool competition," in *Int. Workshop on Search-Based Software Testing (SBST)*. ACM, 2016, pp. 33–36.

[16] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[17] 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.

[18] J. C. Gordon Fraser, José Miguel Rojas and A. Arcuri, "EvoSuite at the SBST 2017 tool competition," in *Int. Workshop on Search-Based Software Testing (SBST)*. IEEE Press, 2016, pp. 39–41.

[19] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*. Springer, 2015, pp. 93–108.

[20] J. M. Rojas, G. Fraser, and A. Arcuri, "Automated unit test generation during software development: A controlled experiment and think-aloud observations," in *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 338–349.

[21] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A Detailed Investigation of the Effectiveness of Whole Test Suite Generation," *Empirical Software Engineering (EMSE)*, vol. 22, no. 2, pp. 852–893, 2016.

[22] U. Rueda, F. Kifetew, and A. Panichella, "Java unit testing tool competition – sixth round," in *Int. Workshop on Search-Based Software Testing (SBST)*, 2018.

[23] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "Instance generator and problem representation to improve object oriented code coverage," *IEEE Transactions on Software Engineering*, vol. 41, no. 3, pp. 294–313, 2015.

[24] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[25] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?" in *IEEE Int. Conference on Software Testing, Verification and Validation (ICST)*, 2018, to appear.