Java Unit Testing Tool Competition — Fifth Round

Annibale Panichella
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Luxembourg
Email: annibale.panichella@uni.lu

Urko Rueda Molina
Research Center on Software Production Methods (DSIC)
Universitat Politècnica de València
Valencia, Spain
Email: urueda@pros.upv.es

***Abstract*—After four successful JUnit tool competitions, we report on the achievements of a new Java Unit Testing Tool Competition. This 5th contest introduces statistical analyses in the benchmark infrastructure and has been validated with significance against the results of the previous 4th edition. Overall, the competition evaluates four automated JUnit testing tools taking as baseline human written test cases from real projects. The paper details the modifications performed to the methodology and provides full results of the competition.**

***Keywords*-tool competition; benchmark; mutation testing; automated unit testing; Java; statistical analysis**

I. Introduction

The key objective in the past four editions of the Java Unit Testing Tool Competition emerges from the need to advance research in the automated testing field. We have evidence from the results of the past edition [1]: each single automated testing tool performed worse than human written test cases. Using benchmarking [2] provides a means to mature the state of development of testing tools. Additionally, a benchmark can report useful data to interested parties, e.g. the software industry.

Following the benchmarking infrastructure of previous contests [1], [3], [4] we continue evaluating JUnit (Java Unit) testing tools targeting Java classes. Firstly, we define a new set of target classes to guarantee that no tool participant takes advantage of knowing the contest' benchmark Java classes. However, the introduction of DEFECTS4J[1], a database of existing faults to enabling controlled testing studies for Java, forces to prepare the new benchmark Java classes for this framework. It allowed to measure test effectiveness on real faults found in projects. Nevertheless, mutation analysis is applied when no real faults are available and we aim to increase the flexibility of introducing new target classes on demand. Thus, we have reverted the benchmark infrastructure to the original JaCoCo (code coverage) and PITest (mutation analysis) configuration to measure the test effectiveness. Together with a more automated infrastructure to run the benchmarks we aim at providing it the capability to define and run new benchmarks as agile as possible.

In this edition of the competition, we evaluate four tools: EVOSUITE [5], JTEXPERT [6], T3 [7], [8] and RANDOOP [9]. The former two tools are submitted by participating developers while the last two are used as baselines for the competition. All tools use the same versions from past edition, with the only exception of EVOSUITE for which the developer participants submitted a new version for this competition. Additionally, we evaluate the human written test cases for the benchmark subjects. Furthermore, this year's competition differs from previous edition in the following ways:

a) Benchmark subjects: The benchmarks for the last edition of the competition were Java classes extracted from the DEFECTS4J data set [10], which provides information about real faults, and code fixes applied by the original developers of the java libraries. While we could have opted for using DEFECTS4J for this edition as well, we decided to change it in order to have a more flexible infrastructure that can be applied for any Java library. In addition, DEFECTS4J contains a limited number of Java libraries and real faults (per library) and it could have been used by participants to over-tune their tools since the DEFECTS4J dataset was already used in the past edition.

Therefore, for this edition we used the following eight well-known open source Java libraries:

- *Apache commons BCEL*[2]: it is a Byte Code Engineering Library which provides utility classes to analyze, create, and manipulate (binary) Java class files. This library contains 431 classes.
- *Apache commons jxpath*[3]: it contains 180 classes implementing utility routines for manipulating Java Beans using the XPath syntax.
- *Apache commons imaging*[4]: it is a large framework with 427 Java classes that support writing and reading operations for a variety of image formats, as well as image info manipulation (e.g., image size).
- *Freehep*[5]: this open-source repository providing Java utilities for high energy physics applications. For this competition, we focused on the *JMinuit* sub-library that contains 180 Java classes.
- *Gson*[6]: it is a well-known open-source library developed by Google that supports the conversion of Java Objects

[1]https://github.com/rjust/defects4j

[2]https://commons.apache.org/proper/commons-bcel/
[3]https://commons.apache.org/proper/commons-jxpath/
[4]https://commons.apache.org/proper/commons-imaging/
[5]http://java.freehep.org
[6]https://github.com/google/gson/blob/master/UserGuide.md

into their JSON representation and vice versa. It contains 174 Java classes.

- *Re2j*[7]: it is a regular expression engine developed by Google for time-linear regular expression matching. With 47 Java classes, it is the smallest library in our benchmark.
- *LA4J*[8]: it contains 208 Java classes that provide Linear Algebra primitives (matrices and vectors) and algorithms.
- *Okhttp*[9]: it is an HTTP and HTTP/2 client for Android and Java applications containing 193 Java classes.

Similar to the previous editions of the contest, we selected as subjects few Java classes randomly sampled from each library in our benchmark. Section II describes the selection procedure and the characteristics of the selected subjects.

b) Benchmark infrastructure: We modified the benchmark infrastructure from the last edition of the contest to allow the evaluation of the participant tools using the libraries in our benchmark that do not belong to the DEFECTS4J data set. In particular, we developed our own analysis engine that combines JaCoCo and PITest for code coverage and mutation analysis while the previous benchmark infrastructure relied on DEFECTS4J to this aim. Section IV details the benchmark infrastructure and the competition methodology.

c) Flaky tests: As done in the previous edition of the contest, this year's competition penalized the generation of flaky tests (i.e., a test that does not reliably pass when executed multiple times on the same program version) and uncompilable test classes. The current benchmark infrastructure automatically detects flaky tests and ignore them when computing coverage and mutation coverage scores. Further details about the detection of flaky tests are reported in Section IV-C.

d) Time budgets: The high acceptance of the time budgeting [1] pushed us to continue using different time budgets to evaluate the tools. We have included two additional small time budgets (10 seconds and half minute) to measure the performance of the tools when the time resource is critical, and an additional 5min budget. That makes a total of 7 time budgets per benchmark subject (10s, 30s, 1min, 2min, 4min, 5min and 8min). Software industry is potentially interested in the automated tools performance for full software projects. To provide orientative data in this line we roughly estimate the time required to generating automated unit tests for a full project as: the time budget used multiplied by the number of CUTs in each project. Table I displays a rough estimation of the test generation time scale (in hours) for the full projects, between the lowest new 10 seconds budget and the highest 8min budget from past edition.

e) Statistical analyses: This year we are introducing statistical analyses with Friedman's and post-hoc Conover's test for multiple pairwise comparison. This has been integrated into the benchmark infrastructure using R packages for significance validation of the results and has been cross-validated with the results from previous year's results.

[7]https://github.com/google/re2j

[8]http://la4j.org

[9]http://square.github.io/okhttp/

TABLE I
NUMBER OF CUTS PER PROJECT AND TEST GEN. TIMES.

Project	#CUTs	10s	8m
apache commons BCEL	431	1,2h	57.5h
apache commons imaging	427	1.2h	57h
LA4J	208	0.6h	27.7h
Okhttp	193	0.5h	25,7h
apache commons jxpath	180	0.5h	24h
freehep-jminuit	180	0.5h	24h
Gson	174	0.5h	23.2h
Re2j	47	0.1h	6.3h

II. THE BENCHMARK SUBJECTS

For the competition, we randomly selected classes from the eight java open-source libraries considered in our benchmark. However, we took into account the McCabe's cyclomatic complexity during the sampling procedure to avoid the selection of trivial classes. Given a method m, the McCabe's cyclomatic complexity is defined as the number of branches in m plus one, which corresponds to the total number of independent paths in the control flow graph [11]. Methods in a class with a cyclomatic complexity equal to one are trivial since they do not contain branches and, thus, can be fully covered by a simple method call. To increase the challenge for the test case generation tools, we excluded for the competition classes having only methods with a low cyclomatic complexity.

To this aim, we follow the same procedure used in [12]: we first computed the McCabe's cyclomatic complexity for all methods in each java library in our benchmark using the extended `CKJM` library[10]. Then, we pruned the benchmark java libraries by removing all trivial classes, i.e., classes that contains only methods with a McCabe's cyclomatic complexity lower than three. We used this threshold for our filtering because a method with cyclomatic complexity equal to three contains at least one *conditional* statement.

From the pruned open-source libraries, we randomly selected (non-trivial) classes as follows: four classes from *apache commons image*, eight classes from *okhttp* and *re2j*, nine classes from *google gson* and ten classes from *apache commons bcel*, *apache commons jxpath*, *LA4J* and *freehep-jminuit* respectively. This resulted in 69 randomly selected non-trivial classes with a total number of branches[11] ranging between 20 and 872, number of lines ranging between 26 and 1076, and number of mutants generated by PIT raging between 32 and 352.

III. BASELINE AND PARTICIPANTS

This year we are using three baselines: manual written tests (available from real projects), RANDOOP and T3; the last two from past edition versions. The active tool participants are EVOSUITE and JTEXPERT, while only the first did make modifications for this year competition (JTEXPERT participants

[10]http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

[11]The number of branches in a class is equal to the sum of the branches contained in its methods.

TABLE II
SUMMARY OF JUNIT CONTEST TOOLS.

Tool	Technique	Static analysis
EVOSUITE [5]	evolutionary algorithm	yes
JTEXPERT [6]	guided random testing	yes
T3 [7], [8]	random testing	no
RANDOOP (baseline) [9]	random testing	no

checked that the past year contest version was good enough). EVOSUITE had 5 days to test and update the tool. Table II provides a summary of the contest JUnit tools.

However, due to modifications to this year infrastructure we updated the RANDOOP runtool wrapper (check chapter IV) for the tool configuration as it requires to provide all dependency classes to exercise in order to create tests for the target class. This is performed using the option (`classlist`) on a per-CUT basis, but this year we have had to remove the list of fault-related classes for each fault that DEFECTS4J provided past edition. No additional modifications were performed for RANDOOP.

Automated JUnit tools aim to mitigate the industry costs in guaranteeing the quality of (Java) software. Thus, we provide human written test cases as baseline for comparison of the tools. However, we can only compare on the effectiveness of the created tests because human developed tests inside real projects are not limited by a time budget at a scale of the competition (10 seconds to 8 minutes). The developer-written test suites have evolved over years and it is not possible for us to estimate the amount of human effort that was required to produce them.

IV. METHODOLOGY

Current year competition shares a similar structure in the methodology compared to previous editions, illustrated in Figure 1. Each JUnit tool is connected to the benchmark infrastructure by implementing a runtool wrapper, a simple communication protocol described in [1]. A benchmark tool is responsible of performing tools benchmarking for a set of predefined target Java CUTs, which have been selected as described above. A set of time budgets (as depicted above) is used to evaluate tools performance at several time windows, which provide insight on the stopping criteria a potential tool user could apply whenever time is a critical resource.

Past edition did parallelize the whole benchmark process due to the large number of executions, which included 4 tools, 4 time budgets, 68 CUTs and 6 repeated runs; 6528 total executions. The work was distributed over 32 virtual machines, split over two HP Z820 workstations with 20 cores and 256Gb of memory each. Yet, the CPU computation roughly took 1 week, 8 months if a single virtual machine was being used.

This year we include 4 tools, 7 time budgets, 69 CUTs and 3 repeated runs for a total of 5796 executions, %89 executions compared to past edition. However, we have introduced many improvements to the infrastructure this year (e.g. mutation

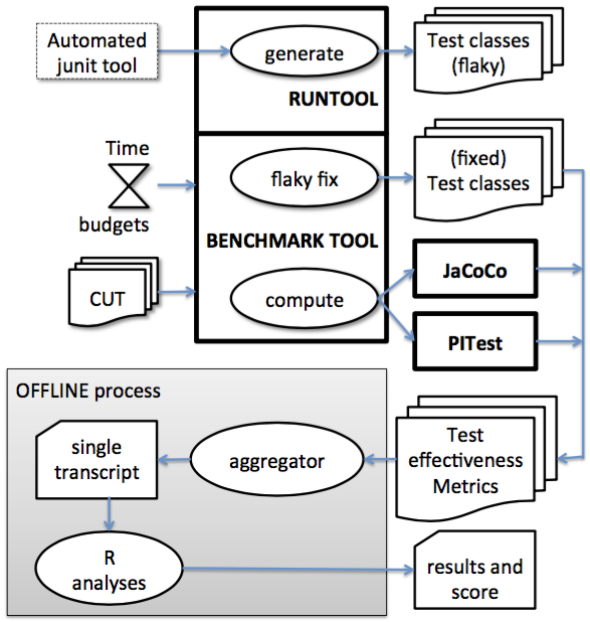

Fig. 1. Overview of the competition methodology.

computation optimizations explained below) so we have used a single HP Z820 workstation. Two virtual machines with 8 CPU cores and 128GB RAM memory each have been provisioned to split the budgets executions. This time the contest did finish computation in 4 complete days with no human intervention between the contest start and the contest end, which is characterized by the availability of metrics. Then, we performed offline analyses consisting of aggregating all the metrics into a single transcript file (5796 data points) and the execution of R scripts. We made the results available to all participants as soon as possible.

We next provide further details of the whole methodology.

A. JUnit tools preparation

Nothing new from the tool participants' perspective compared to previous edition. Sample test subjects from real projects, not overlapping with the contest projects and CUTs, are still available to prepare the JUnit tools: Chart-5, Math-9, Lang-61, Time-6 and Closure-9. The benchmark protocol (runtool wrapper) keeps the same with no further modifications. The participants were able to use these test subjects to test the correct working of their tools. EVOSUITE and JTEXPERT took part on this process, the former updated with a new tool release while the latter decided the past edition version was already prepared for this year. RANDOOP and T3 did not take part of this process, so past edition versions and configuration have been used.

B. Test generation

We have reduced to half the number of repeated executions of the JUnit tools, which was required to account for the randomness in the tools test generation process. We face this reduction with statistical significance of the results.

For each time budget, all the tools were executed 3 repeated times in parallel with 8 dedicated CPU cores and 128GB RAM. As soon as a JUnit tool did finish the test generation phase it continued into the metrics computation phase. The time budgeting implementation stays equal as previous year: a tool was allowed to run at most twice as long as the given time budget and the infrastructure terminates a tool execution if the budget frame is exceeded. Again, the scoring formula (check below) applies a penalty, which is inversely proportional to the excess ratio of the time budget.

Compared to past edition, in which each tool was executed in a dedicated virtual machine with a single CPU core (no multi-threading advantages in tools implementations), we have removed the restriction to allow the tools perform realistically. However, the tools had to compete for the available resources.

C. Flaky tests management

We executed each generated test 5 times for the detection of flaky tests, which are tests that does not reliably pass when executed multiple times on the same test subject. For example, a test that asserts on the system time only passes during generation and fails on every later execution. A JUnit tool might generate flaky or uncompilable tests, even more if a time budget threshold is reached and the infrastructure needs to stop the running tool to preserve the equality of all the contest tools.

While in last past edition flaky/uncompilable tests were directly removed leveraging the DEFECTS4J infrastructure, in this edition our benchmark infrastructure does not remove such tests but ignores them during the metrics computation. We opted for this choice to avoid issues when changing the original tests. Therefore, failing JUnit test methods are simply skipped when computing code coverage and mutation coverage (i.e., ratio of mutants killed).

The scoring formula, as in previous edition, does account for the sanity of tests. Tools are penalized if the generated tests do not compile or are flaky.

D. Metrics computation

The metrics computation phase starts per time budget as soon as a tool finishes generating tests, either because a time budget threshold was reached or because the tool finished its computations. Metrics analyses are performed per budget and tool using 3 simultaneous threads as we were already aware that the computation phase was more time consuming than the test generation phase, from past experience on the previous year competitions.

We are using JaCoCo and PITest similarly to second [4] and third editions [3] for the metrics analyses. However, we have introduced key optimizations this year. For PITest, we did not use its running engine since it gave errors for test cases with ad-hoc/non-standard JUnit runners (e.g. in the EVOSUITE tool). Thus, we only use PITest engine for the generation of mutants, but the execution is done using our own benchmark execution engine. Additionally, to reduce the cost of mutation analyses we combine JaCoCo coverage information with PITest. The final result is that we only execute tests against covered mutants, i.e. mutants that affect/infect lines covered during the execution of the test (according to JaCoCo).

Additionally, we have introduced a strict mutation analysis time window of 5 minutes per tool, budget and CUT. The rationale behind this decision comes from the fact that, during the checks of the benchmark infrastructure, there were some cases in which evaluating each of the PITest generated mutants for a CUT took excessive time. For guaranteeing that the contest did finish in a fair time the mutation analysis skips the remaining mutants after the 5 minutes window. To fit with the comparability of tools results, the order by which the mutants are evaluated is kept the same for all the tools in each budget and CUT.

Next, we further detail in the computed metrics for each tool, time budget and CUT (no real-fault detection metric included as past edition, which restricts the benchmark subjects to projects with real faults):

a) Code coverage: No modifications from past edition. Again, for each test suite the benchmark infrastructure computes two code coverage ratios, statement coverage and condition coverage.

b) Mutation coverage: This year, mutants generation relies on the PITest engine. Then, we compute mutation metrics against these mutants using our own benchmark infrastructure, which resolves issues that have been around in the previous competitions related to evaluating tests not properly set up for the PITest engine.

E. Scoring formula

Only minor adjustments are performed to the scoring formula from past edition [1], which determines the ranking of the JUnit tools considering their performance in the following aspects:

a) Coverage score: Each tool is executed 3 times using 7 different time budgets. Given a tool T, a time budget B and a class under test C, the next set of test effectiveness metrics are gathered for each execution run r:

$$covScore_{\langle T,B,C,r\rangle} := w_i \cdot cov_i + w_b \cdot cov_b + w_m \cdot cov_m$$

It considers the achieved instruction coverage (cov_i), branch coverage (cov_b), and mutation coverage (i.e., the ratio of killed mutants cov_m). w_i, w_b and w_m are the weights, for which we keep the values $w_i = 1$, $w_b = 2$, and $w_m = 4$.

Observe that test methods are ignored for uncompilable and flaky tests so that implicitly reflects negatively into lower coverages.

b) Time score: A tool is benchmarked for each target subject for a limited time L equal to $2 \times B$. A penalty is applied to the score if the tool exceeds the time budget:

$$tScore_{\langle T,B,C,r\rangle} := covScore_{\langle T,B,C,r\rangle} \cdot min\left(1, \frac{L}{genTime}\right)$$

where L is the time limit (twice the budget) and $genTime$ (a value between 0 and $2 \times L$) is the total generation time

spent by the tool T for the execution r with the target subject C. Observe that the coverage score can be reduced to half in the worst case.

c) Test sanity score: The score for a tool T at a given execution r with time budget B and a class under test C is penalized by the number of uncompilable and flaky tests:

$$score_{\langle T,B,C,r\rangle} := tScore_{\langle T,B,C,r\rangle} - penalty_{\langle T,B,C,r\rangle}$$

where:

$$penalty_{\langle T,B,C,r\rangle} := \begin{cases} 2 & \text{if no compilable test classes} \\ \frac{\#uClasses}{\#Classes} + \frac{\#fTests}{\#Tests} & \text{otherwise} \end{cases}$$

where $\#uClasses$ and $\#Classes$ are the number of uncompilable generated test classes and total generated test classes respectively, and $\#fTests$ and $\#Tests$ are the number of flaky test cases and total test cases respectively.

d) Average scores: Given the non-determinism of the four JUnit tools, we executed each tool 3 times. Thus, the score for a given time budget B and class under test C is the average of all the executions for the same tool T, budget B and class under test C:

$$score_{\langle T,B,C\rangle} := avg\big(Score_{\langle T,B,C,r\rangle}\big) \text{ for all } r \text{ executions}$$

e) Final score: The *final* score for a tool T is the sum of all scores for all classes under test and time budgets used in the competition:

$$score_T := \sum_{B,C} Score_{\langle T,B,C\rangle}$$

F. Statistical Analysis

For the statistical analysis, we carefully followed the guidelines by García et al. [13] for comparing different randomized tools over a set of benchmark functions, which are selected 69 Java classes in our case. In particular, we apply the Friedman test by comparing the four participant tools over 69 benchmark subjects and seven different search budgets. Therefore, each tool has (69 $\times$ 7) = 483 data points (or configurations), where each data point is the average (mean) score achieved by the tool under analysis over three different independent runs for a given configuration. The four distributions obtained for the four participant tools, are then compared using the Friedman test [13] with significance level p-value=0.05. The Friedman test is a non-parametric test for multiple-problem analysis and it departs from the traditional tests for significance (e.g., the Wilcoxon test) since it computes the ranking between algorithms over multiple independent problems, i.e., benchmark subjects in our case. A significant p-value indicates that the *null hypothesis* has to be rejected (i.e., no participant tool performs significantly different from others) in favor of the *alternative* one (i.e., participant tools are significantly different from each other). If the null hypothesis is rejected, we use the post-hoc Conover's test for pairwise multiple comparisons. Such a test is used to detect pairs of tool participants that are significantly different. Finally, p-values obtained with the post-hoc test are adjusted with the Holm-Bonferroni procedure to correct the statistical significance level (p-value=0.05) in case of multiple comparisons.

G. Threats to Validity

This section discusses the main threats that could potentially affect the validity of the competition.

a) Conclusion validity: To address the threats affecting the *reliability of treatment implementation*, we have used the same protocol for running and assessing the different tools in the contest. In addition, we gave the same instructions to all developers of the tool participating to the unit testing tool competition. For what concern the *reliability of measures*, all tools in the contest have been executed with the same time budgets, where all timing information was measured using Java native method `System.currentTimeMillis()`. To assess the performance of the tools we used widely applied quality indicators, which are line coverage, branch coverage and ratio of killed mutants. For these performance indicators, we relied on JaCoCo and PITest which have been extensively used in the related literature.

Finally, to address the randomness nature of the tools in the competition, we run the benchmark multiple times and we draw our conclusions on the average performance scores achieve over multiple independent run. Due to time and resource restrictions we could only run each tool a maximum of three times. However, for the statistical analysis we used two non-parametric tests (i.e., the Friedman test and the post-hoc Conover's procedure) that are less sensible to a low number of repetitions (tools re-executions) [13]. Indeed, the sample size for the two tests is not represented by the number of independent runs but it is given by the number of subjects (69 Java classes) multiplied by the number of search budgets (seven budgets in our case) [13].

b) Internal validity: To mitigate the threats to validity related to the selection of the CUTs, we randomly selected 69 Java classes from eight well-known open-source projects. To avoid the selection of trivial classes, we first filtered out classes (see Section II) prior the random sampling of the CUTs. Before running the contest, we have extensively tested our own benchmark infrastructure using the CUTs from both this and the last edition of the competition. To further improve the confidence on the overall contest, the developers of the participant tools could test their tools with the benchmark infrastructure for five days before we ran the competition.

c) Construct validity: As done in past editions, the comparison among testing tools has been performed using a scoring formula that combines different quality indicators into only one single scalar value. The weights in the scoring formula were assigned in accordance with those quality indicators that are more correlated to the fault detection capability [14]. While the final goal of generated tests is to reveal faults, is it impossible to know a priori all faults in a given program. Consequently, we used commonly used surrogate quality indicators such as code coverage ratio or killed mutants. In accordance with the related literature, we gave more importance (larger weight) to mutation coverage than code coverage (both line and branch coverage) since previous work [14] demonstrated a larger positive correlation real fault detection capability and mutation coverage.

TABLE III
AVERAGE LINE, BRANCH AND MUTATION COVERAGE AT DIFFERENT SEARCH BUDGETS

Tool	Budget (in sec)	Line Cov.			Branch Cov.			Mutation Cov.		
		Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
EVOSUITE	10	0	0.35	0.98	0	0.27	0.90	0	0.15	0.74
JTEXPERT	10	0	0.31	0.97	0	0.26	0.93	0	0.19	0.80
RANDOOP	10	0	0.24	0.91	0	0.16	0.84	0	0.09	0.74
T3	10	0	0.22	0.86	0	0.15	0.88	0	0.09	0.64
EVOSUITE	30	0	0.44	0.98	0	0.36	0.93	0	0.20	0.74
JTEXPERT	30	0	0.36	0.98	0	0.31	0.95	0	0.23	0.84
RANDOOP	30	0	0.25	0.94	0	0.18	0.88	0	0.10	0.74
T3	30	0	0.32	0.96	0	0.26	0.92	0	0.08	0.77
EVOSUITE	60	0	0.58	0.99	0	0.51	0.98	0	0.36	0.97
JTEXPERT	60	0	0.38	0.99	0	0.33	0.95	0	0.25	0.92
RANDOOP	60	0	0.25	0.94	0	0.18	0.89	0	0.10	0.74
T3	60	0	0.32	0.96	0	0.26	0.92	0	0.08	0.70
EVOSUITE	120	0	0.67	1.00	0	0.61	0.97	0	0.46	0.97
JTEXPERT	120	0	0.40	0.97	0	0.35	0.96	0	0.26	0.89
RANDOOP	120	0	0.26	0.96	0	0.20	0.91	0	0.09	0.76
T3	120	0	0.32	0.96	0	0.26	0.92	0	0.08	0.55
EVOSUITE	240	0	0.70	1.00	0	0.65	1.00	0	0.50	0.97
JTEXPERT	240	0	0.42	1.00	0	0.37	0.96	0	0.26	0.88
RANDOOP	240	0	0.25	0.96	0	0.19	0.92	0	0.09	0.74
T3	240	0	0.32	0.96	0	0.26	0.92	0	0.08	0.68
EVOSUITE	300	0	0.67	1.00	0	0.62	1.00	0	0.47	0.96
JTEXPERT	300	0	0.40	1.00	0	0.35	0.96	0	0.26	0.87
RANDOOP	300	0	0.25	0.94	0	0.19	0.91	0	0.09	0.76
T3	300	0	0.32	0.96	0	0.26	0.92	0	0.09	0.64
EVOSUITE	480	0	0.72	1.00	0	0.66	1.00	0	0.51	1.00
JTEXPERT	480	0	0.40	1.00	0	0.35	0.96	0	0.26	0.92
RANDOOP	480	0	0.25	0.96	0	0.18	0.92	0	0.09	0.76
T3	480	0	0.34	0.96	0	0.27	0.92	0	0.09	0.57

V. RESULTS

Table III provides the descriptive statistics (min, max, and mean value) for the coverage metrics (i.e., line, branch and mutation coverage) achieved by the different tools over the different time budgets. As we can notice, the minimum line coverage (branch and mutation coverage as well) is zero for all tools in the contest and for all search budgets. This indicates that our benchmark contains CUTs for which none of the participant tools was able to generate test cases even after eight minutes of search. One of this CUT is the class `CCSMatrix` extracted from the library *La4j*. Such a class implements the Compressed Column Storage (CCS) format for sparse matrices in Java. We notice that this class is particularly expensive from a computation point of view when its constructors are called with large (random) input values. Indeed, we observed that all tools could perform only few iterations even within the largest budget of eight minutes. We also obtained similar results for the class `CRSMatrix` from the same library, which implements instead the Compressed Row Storage (CRS) format for sparse matrices. This observation may provide useful hints for the developers of the participant tools when generating tests for classes implementing computational expensive routines.

Table IV shows the scores achieved by the different participant for each time budget. Instead, Table V compares the scores achieved by automated tools when set with the largest time budget (i.e., 8 minutes) with the scores yielded by tests written by the original developers of the Java projects in our contests (DEVELOPER), as well as the optimal score (OPTIMAL). Notice that for this comparison, we only considered the 63 subjects for which we found developers-written tests. Moreover, for DEVELOPER no time budget applies (also for OPTIMAL) and the corresponding tests were executed only once due to their deterministic nature.

TABLE IV
SCORES FOR ALL TIME BUDGETS.

Tool	Budget (in sec.)	Score	Std.dev
EVOSUITE	10	56.20	13.51
T3	10	55.84	18.90
JTEXPERT	10	61.86	12.54
RANDOOP	10	50.95	4.08
EVOSUITE	30	123.03	14.32
T3	30	77.36	11.66
JTEXPERT	30	106.72	14.39
RANDOOP	30	64.40	5.29
EVOSUITE	60	207.46	46.48
T3	60	77.72	14.64
JTEXPERT	60	125.52	16.65
RANDOOP	60	65.56	5.16
EVOSUITE	120	255.18	29.85
T3	120	77.64	11.38
JTEXPERT	120	136.47	10.11
RANDOOP	120	67.50	4.72
EVOSUITE	240	274.01	29.76
T3	240	78.64	
JTEXPERT	240	141.83	15.98
RANDOOP	240	66.74	4.90
EVOSUITE	300	260.47	28.57
T3	300	77.87	11.85
JTEXPERT	300	137.77	14.64
RANDOOP	300	66.53	5.68
EVOSUITE	480	280.90	30.23
T3	480	81.20	14.01
JTEXPERT	480	138.37	17.71
RANDOOP	480	65.93	4.91

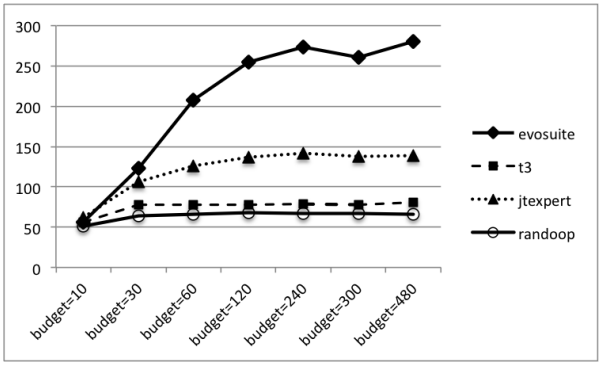

Fig. 2. Tools performance per budget.

Figure 2 shows how the tools perform, based on the achieved scores, with gradual increase of the time budgets. Finally, Table VI gives the overall scores for the four tools. The differences among the achieved scores turn out to be statistically significant according to the Friedman, which returns a significant p-value $< 10^{-16}$. For completeness, the ranking produced by the Friedman tests is depicted in Table VI together with the final scores. To better understand for which pairs of tools the statistical significance holds, Table VII reports the p-values obtained by the post-hoc Conover's procedure for the pairwise comparison. As we can notice,

TABLE V
COMPARISON WITH OPTIMAL (IDEAL) SCORE AND DEVELOPER-WRITTEN TESTS CONSIDERING ONLY THE 63 SUBJECTS WITH AVAILABLE TESTS

Tool	Budget (in sec.)	Score	Std.dev
EVOSUITE	480	250.81	28.71
T3	480	78.15	13.96
JTEXPERT	480	125.11	17.09
RANDOOP	480	60.78	4.75
DEVELOPER	–	268.12	–
OPTIMAL	–	441	–

TABLE VI
OVERALL SCORES FOR ALL TOOLS AND RANKINGS OBTAINED THROUGH FRIEDMAN TEST

Tool	Budget	Score	Std.dev	Ranking
EVOSUITE	*	1457	192.72	1.55
JTEXPERT	*	849	102.03	2.71
T3	*	526	82.43	2.81
RANDOOP	*	448	34.74	2.92

the statistical significance holds for all pairs of tools being compared since the p-values are always $<$0.01. We observe a marginal statistical significant difference only between T3 and RANDOOP for the subjects in our benchmark.

ACKNOWLEDGEMENT

We would like to thank all the JUnit tool participants for their appreciated yearly contributions and feedback. Special thanks go to the benchmark infrastructure contributors since the beginning of the competitions. This work was partly funded by the PERTEST project (TIN2013-46928-C3-1-R) and the National Research Fund, Luxembourg FNR/P10/03.

APPENDIX

Tables VIII, IX, X, XI, XII, XIII and XIV in our online appendix [15] provide the detailed results for each tool on the seven time budgets (10, 30, 60, 120, 240, 300 and 480 seconds). All numbers are averaged across 3 runs for each triple: *CUT* x *tool* x *time-budget*.

TABLE VII
RESULTS OF THE POST-HOC CONOVER'S TEST FOR PAIRWISE ANALYSIS

	EVOSUITE	JTEXPERT	RANDOOP	T3
EVOSUITE	-	-	-	-
JTEXPERT	< 0.01	-	-	-
RANDOOP	< 0.01	< 0.01	-	-
T3	< 0.01	0.01	0.06	-

REFERENCES

[1] U. Rueda, R. Just, J. P. Galeotti, and T. E. J. Vos, "Unit testing tool competition - round four," in *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, May 2016, pp. 19–28.

[2] S. E. Sim, S. Easterbrook, and R. C. Holt, "Using benchmarking to advance research: A challenge to software engineering," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 74–83. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776826

[3] U. Rueda, T. E. J. Vos, and I. S. W. B. Prasetya, "Unit testing tool competition: Round three," in *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, ser. SBST '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 19–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2821339.2821346

[4] S. Bauersfeld, T. E. J. Vos, and K. Lakhotia, *Unit Testing Tool Competitions – Lessons Learned*. Cham: Springer International Publishing, 2014, pp. 75–94. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07785-7_5

[5] 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.

[6] A. Sakti, G. Pesant, and Y.-G. Guéhéneuc, "JTExpert at the fourth unit testing tool competition," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ser. SBST '16. New York, NY, USA: ACM, 2016, pp. 37–40. [Online]. Available: http://doi.acm.org/10.1145/2897010.2897021

[7] I. Prasetya, "T3i: A tool for generating and querying test suites for java," in *10th Joint Meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2015. [Online]. Available: http://dspace.library.uu.nl/bitstream/handle/1874/321619/950.pdf?sequence=1

[8] Prasetya, I.S.W.B., "Budget-aware random testing with T3: benchmarking at the SBST2016 testing tool contest," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*. ACM, 2016, pp. 29–32. [Online]. Available: http://dx.doi.org/10.1145/2897010.2897019

[9] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[10] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 437–440.

[11] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[12] A. Panichella, F. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–37, 2017, to Appear.

[13] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization," *Journal of Heuristics*, vol. 15, no. 6, pp. 617–644, Dec. 2009. [Online]. Available: http://dx.doi.org/10.1007/s10732-008-9080-4

[14] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, November 18–20 2014, pp. 654–665.

[15] A. Panichella and U. Rueda, "Java unit testing tool competition — fifth round," Tech. Rep., 24th February, 2017. [Online]. Available: http://sbstcontest.dsic.upv.es/SBSTcontest2017_detailed_results.pdf