Java unit testing tool competition - Fifth round

Panichella, Annibale; Molina, Urko Rueda

DOI
10.1109/SBST.2017.7

Publication date
2017

Document Version
Peer reviewed version

Published in

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
Java Unit Testing Tool Competition - Sixth Round

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

Fitsum Kifetew  
Fondazione Bruno Kessler  
Trento, Italy  
kifetew@fbk.eu

Annibale Panichella  
SnT – University of Luxembourg, Luxembourg  
Delft University of Technology, The Netherlands  
annibale.panichella@uni.lu

ABSTRACT
We report on the advances in this sixth edition of the JUnit tool competitions. This year the contest introduces new benchmarks to assess the performance of JUnit testing tools on different types of real-world software projects. Following on the statistical analyses from the past contest work, we have extended it with the combined tools performance aiming to beat the human made tests. Overall, the 6th competition evaluates four automated JUnit testing tools taking as baseline human written test cases for the selected benchmark projects. The paper details the modifications performed to the methodology and provides full results of the competition.

CCS CONCEPTS
• Software and its engineering → Software defect analysis; Software testing and debugging; Empirical software validation; Search-based software engineering;

KEYWORDS
tool competition, benchmark, mutation testing, automation, unit testing, Java, statistical analysis, combined performance

ACM Reference Format:

1 INTRODUCTION
Continuing the tradition of the past five editions [8] of the Java unit testing tool competition, this year as well we have carried out the competition on a fresh set of test classes under test (CUT) selected from various projects. In the current edition, as in the previous [8], there are a total of four tools considered for the competition, namely: EvoSuite [1], JTestExpt [12, 13], T3 [10, 11], and Randoop [6]. All the tools were executed against the same set of subjects, set of time budgets, and execution environment.

In this year’s edition, we do not have new tools entering into the competition, however the developers of EvoSuite and T3 have actively participated with improved versions of their tools. Furthermore, we have introduced a combined analysis in which we construct test suites by putting together all the tests generated, for a particular CUT, by all four tools. Such an analysis could shed some light on the overall strengths and weaknesses of the tools with respect to the CUTs under consideration. We also compare and contrast the results achieved by the various tools, either individually or combined together, against the manually-written test suites included in the original projects from which our test subjects were extracted.

For comparing the tools, we used well-stablished structural coverage metrics, namely statement and branch coverage, which we computed by using JaCoCo. Additionally, we apply mutation analysis to assess the fault revealing potentials of the test suites generated by the tools. To this aim, we use the PITest mutation analysis tool to compute the mutation scores of the various test suites (either automatically generated or manually-written).

Following lessons learned from previous editions, we have considered different time budgets, i.e., 10, 60, 120, and 240 seconds. Such a range of search budgets allows us to assess the capabilities of the tools in different usage scenarios. Furthermore, we augment the comparative analysis by considering the combined test suites composed of all the tests generated by all tools in the competition.

The report is organized as follows. Section 2 describes the set of benchmarks used this year, which were not used in previous competitions, and section 3 describes the objects under study (the tools) and the baseline (developer tests). Next, section 4 collects the changes introduced since the last competition [8]. The obtained results of running the competition are later described in section 5. Finally, our concluding remarks are available in section 6.

2 THE BENCHMARK SUBJECTS
Building a benchmark for assessing testing tools is always challenging as it requires considering different factors. For example, the benchmark should be a representative sample of real-world software projects [2]; the projects should be open-source and cover different application domains [2]; the classes should not be too trivial [7] (e.g., classes with only branchless methods) and should have different types of input parameters. With the aim of taking into account these factors, for this edition we focused on the top 500 GitHub repositories that satisfy the following criteria: (i) having more than 4K stars on 01/01/2018, (2) can be built using Maven, and (3) contain JUnit 4 test suites. From this large pool of possible candidates, we randomly sampled (through a script that filtered the projects based on the criteria) the following seven projects:
---

**Table 1: Characteristics of the projects in our benchmark**

<table>
<thead>
<tr>
<th>Project</th>
<th>#Stars</th>
<th>#CUTs</th>
<th>10s</th>
<th>4m</th>
<th># Sampled CUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dubbo</td>
<td>16.3K</td>
<td>235</td>
<td>39m</td>
<td>15.7h</td>
<td>9</td>
</tr>
<tr>
<td>FastJson</td>
<td>12.6K</td>
<td>217</td>
<td>36m</td>
<td>14.5h</td>
<td>10</td>
</tr>
<tr>
<td>JSoup</td>
<td>5.6K</td>
<td>248</td>
<td>41.3m</td>
<td>16.5h</td>
<td>5</td>
</tr>
<tr>
<td>Okio</td>
<td>4.6K</td>
<td>44</td>
<td>7.3m</td>
<td>2.9h</td>
<td>10</td>
</tr>
<tr>
<td>Redisson</td>
<td>4.4K</td>
<td>1,392</td>
<td>3.9h</td>
<td>92.8h</td>
<td>10</td>
</tr>
<tr>
<td>Webmagic</td>
<td>6.1K</td>
<td>162</td>
<td>27m</td>
<td>10.8h</td>
<td>5</td>
</tr>
<tr>
<td>Zxing</td>
<td>17.4K</td>
<td>268</td>
<td>44.7m</td>
<td>17.9h</td>
<td>10</td>
</tr>
</tbody>
</table>

- **Dubbo**\(^1\): is a large remote procedure call (RPC) and microservice framework written in Java. For the competition, we focused on the maven sub-module **dubbo-common**.
- **FastJson**\(^2\): is a Java library providing utilities to convert JSON string to an equivalent Java object and vice versa.
- **JSoup**\(^3\): is a Java library containing classes and methods for extracting and manipulating data stored in HTML documents using Document Object Model (DOM) traversal methods and CSS and jQuery-like selectors.
- **Okio**\(^4\): is a small Java library providing utilities to access, store and process binary and character data using fast I/O and resizable buffers.
- **Redisson**\(^5\): implements a Java client for *redis* and provides distributed Java objects and services, such as *List*, *Queue*, *Cache*.
- **Webmagic**\(^6\): is a multi-thread web crawler framework supporting all typical crawler activities, such as url management and web page content extraction. For the competition, we focused on two maven sub-modules, namely webmagic-core and webmagic-extension.
- **Zxing**\(^7\): is an open-source library that supports the decoding and the generation of barcodes (e.g., QR Code).

Table 1 summaries the main characteristics of the selected projects. The total number of CUTs in each project ranges between 44 (Okio) and 1,392 (Redisson) classes. Computing test cases for the full projects would take between 7 minutes (10 seconds budget per CUT) and nearly 93 hours (4 minutes budget).

Comparing the tool participants on the entire projects was clearly unfeasible due to very large amount of time the competition would require: (i) running each tool on each CUT multiple times, (ii) with different time budgets, (iii) collecting the corresponding performance metrics (among which, mutation score is very resource and time demanding) from each independent run. For these reasons, we randomly sampled few CUTs from each project as reported in Table 1.

For sampling the CUTs, we used the same procedure used in the previous edition of the contest \([8]\) and leveraging the McCabe’s cyclomatic complexity. First, we computed the cyclomatic complexity for all methods and classes in each project using the extended CKJM library\(^8\). Then, we filtered the benchmark projects by removing classes that contain only methods with a McCabe’s cyclomatic complexity lower than three. The McCabe’s cyclomatic complexity of a method \(m\) corresponds to the number of branches in \(m\) plus one (or equivalently, the total number of independent paths in the control flow graph of \(m\)). Therefore, our filtered benchmark contains only classes with at least one method with at least two condition points (i.e., with a complexity \(\geq 3\)). This filter reduces the chances of sampling very trivial classes with either no branches or that can be fully covered with few randomly generated tests \([7]\).

From the filtered benchmark, we randomly sampled few classes from each project as follows: five classes from JSoup and Webmagic, nine classes from Dubbo, plus 10 classes from each of the remaining four projects. This resulted\(^9\) in 59 Java classes, whose number of branches ranges between 4 and 2197, while number of lines ranges between 26 and 3091, and number of mutants produced by PIT ranges between 16 and 1023.

3 THE BENCHMARK OBJECTS

In this edition of the competition, a total of four tools are considered. The tools are the same as in the previous edition, with the exception that EvoSuite has been updated to a new version. T3 has introduced some changes by way of bug fixes and improved competition interface implementation for better integration with the evaluation framework. The other tools, i.e., Randoop and JTexpert, remain the same as in the previous edition.

3.1 Baseline human made JUnit tests

As baseline, we use test suites generated by Randoop, as well as the manually written test suites of the CUTs available from their respective projects. Even though we use human test suites as baseline with the aim of giving an idea of how the automatically generated test suites fair with respect to human written tests, it is difficult to draw direct parallels between the two. Human written test suites are typically evolved and improved overtime, and it is usually hard to determine exactly how much (human) effort has been spent in producing each test suite.

Additionally, we use the JTexpert tool as baseline, because it is not updated since the last competition and the authors are not actively participating in the current competition.

3.2 Competition Tools

This year, the active competitors are EvoSuite and T3. As shown in Table 2, EvoSuite uses an evolutionary algorithm for evolving test suites, while T3 employs a random testing strategy. The table also summarizes the main characteristics of the four tools considered in this edition. Moreover, similar to what was done in previous editions, participants were able to test their implementation using a set of sample CUTs. Concretely, the full set of CUTs from the previous competition \([8]\). Note that the CUTs used in this edition are all newly selected and were not revealed to the authors before running the competition.

---

\(^1\)https://github.com/alibaba/dubbo
\(^2\)https://github.com/alibaba/fastjson
\(^3\)https://github.com/jhy/jsoup
\(^4\)https://github.com/square/okio
\(^5\)https://github.com/redisson/redisson
\(^6\)https://github.com/code4craft/webmagic
\(^7\)https://github.com/zhenglongxing/zxing
\(^8\)http://gromit.imar.pwr.wroc.pl/p_inf/ckjm/
\(^9\)https://github.com/PROSRESEARCHCENTER/junitcontest/tree/master/bin/benchmarks_6th
4 CONTEST METHODOLOGY
This 6th contest shares most of the methodology from the previous edition [8]. We will focus on describing the modifications made to run this year competition.

→ **Public contest repository**\(^\text{10}\). The full contest infrastructure was published to GitHub four months before the competition. The initial objective was to attract new participants raising the awareness, with no success. However, the long run aim was to share the competition experiences to allow future competitors to collaborate, better prepare their tools for automation, report bugs, request new features or improvements, etc. Therefore, the global goal was to advance the maturity of the infrastructure built for the competition, and so the efficiency and effectiveness of the tools by offering a public benchmark to compare with.

→ **JUnit tools set up.** Participants were able to test their correct operation for the contest using the latest version of the infrastructure. We provided them the full set of benchmarks from past 5th contest [8], which did not contain any of the benchmarks from this edition.

→ **CUTs.** We selected 59 new benchmark classes as described in Section 2 and that constitute the subjects of the competition.

→ **Execution frame.** This year a total of 5664 executions have been scheduled (5796 executions in the previous edition): 59 CUTs × 4 tools × 4 time budgets × 6 repetitions for statistical analyses. In an attempt to foster the replicability of the contest executions we have transferred the know-how of the past 5 years to a new environment operated by new people. We have switched the infrastructure from an HP Z820 workstation with two virtual machines, each with 8 CPU cores and 128GB RAM, to a cluster environment running Sun Grid Engine (SGE). We have used three physical nodes each with 24 CPU cores and 256GB RAM. On each node, we executed two repetitions of the tools on all four budgets, for a total of six repetitions in total. Similar to the previous edition, all tools were executed in parallel. For each tool, and each search budget, the contest infrastructure first invokes the tool on each CUT to generate the test cases. Once test generation is completed, the infrastructure continues to the second phase, which is the computation of the metrics, i.e., coverage and mutation scores.

→ **Test generation.** The execution frame used this year granted enough power to repeat the generation of tests by tools a total of 6 times, to account for the inherent randomness of the generation processes. The tools had to compete for the available resources as they were run in parallel. The benchmark subjects used this year made the contest execution to sporadically hang during tests generation. In that case, we had to force kill some of the executions as neither the tools nor the contest infrastructure did succeed to stop the related processes. The impact for these executions is a 0 score as the provided budget is exceeded. Additionally, and continuing the automation procedure of past competitions, the CUTs were specified with the paths for: i) source java files, ii) compiled class files, and iii) the classpath with the required dependencies. However, this specification missed some critical dependencies on some CUTS (e.g. DUBBO-2, WEBMAGIC-4) and tools could have generated crashing test cases.

→ **Metrics computation.** We kept the strict mutation analysis time window of 5 minutes per CUT, and a timeout of 1 minute for each mutant. Moreover, mutants sampling\(^\text{11}\) is applied on the set of mutants generated by PTest. The rationale behind it is to reduce the computation time and provide results in a feasible amount of time. Moreover, recent studies [4] showed that random sampling is particularly effective despite its very low computational complexity compared to other mutant reduction strategies. Note that we applied the same set of sampled mutants to evaluate the test suites generated by different tools on the same CUT.

→ **Combined analyses.** To explore whether the combined tools’ tests would outperform the developer designed tests, we have introduced the combined analyses to evaluate the cooperative test performance. The process consists of building new test suites that contain all test cases generated by all tools on a given CUT and with a given time budget. Then, the metrics computation is performed on the combined test suite in the exact same way as for the individual tools. Yet, the computation costs increase to the sum of the costs required to evaluate the test suites generated by each individual tool. We approached it in a separate analysis to measure the achieved instruction and branch coverages, and the test effectiveness. Furthermore, due the high computation costs for the full combined analyses we were only able to obtain data on the budget of 10 seconds as we run out of time to compute the rest of the budgets.

→ **Time budgets.** In the former edition of the competition [8], we did not observe any significant improvement (i.e., coverage and mutation score) after four minutes of search budget. Therefore, for this edition of the competition we have decided to consider only four search budgets, i.e., 10, 60, 120 and 240 seconds. This allowed us to use the saved computation resources for the combined analyses introduced in this edition.

→ **Statistical Analysis.** Similar to the previous edition of the competition [8], we used some statistical tests to support the results collected in this edition of the competition. First, we use the Friedman test to compare the scores achieved by the different tools over the different CUTs and different time budgets. In total, each tool produced (59 CUTs × 4 budgets) × 236 data points, corresponding to the average scores achieved across six independent repetitions. Second, we applied the post-hoc Conover’s test for pairwise multiple comparisons. While the former test allows us to assess whether the scores achieved by alternative tools differ statistically significantly from each other, the latter test is used to determine for which pair of tools the significance actually holds.

In this edition, we augmented the statistical analysis by using the permutation test [3] to assess whether there exists a significant interaction among the scores produced by the tools, the cyclostatic

---

\(^{10}\)https://github.com/PROSRESEARCHCENTER/junitcontest

\(^{11}\)We applied a random sampling of 33% for CUTs with more than 200 mutants, and a sampling of 50% for CUTs with more than 400 mutants.
complexity of the CUTs and the allocated search budgets. The
permutation test is a non-parametric equivalent of the ANOVA
(Assay of Variance) test and it is performed by randomly per-
muting data points across different groups in a given distribution.
For this test, we set the number of iterations to a very large number
(i.e., $10^8$) to obtain robust results [7].

Note that for all aforementioned statistical tests, we used the
confidence level $\alpha=0.05$; $p$-values obtained with the Conover’s test
were further adjusted with the Holm-Bonferroni procedure, which
is required in case of multiple comparisons.

4.1 Threats to Validity

Conclusion validity. As in previous editions, we perform statisti-
cal analyses for significance. In addition, we applied the permuta-
tion test to analyze possible co-factors that could potentially
influence the performance of the tools.

Internal validity. We have a trajectory of six competitions
where the contest infrastructure has been constantly stressed
and improved. Furthermore, this year the infrastructure was made
public four months before the competition, which allowed for the
identification of potential threats of implementation. Only the bench-
marks to be used in the competition were hidden to the participants,
while they were able to test the environment with the full bench-
marks from past competition.

Construct validity. The scoring formula used to rank the tools
—which is identical to the past edition— assigns a higher weight to
while they were able to test the environment with the full bench-
marks’ CUTs) of varied complexity to account for their repre-
sentativeness on the real world, and include developers’ tests and a
special case since its configuration missed critical dependencies
(e.g., javassist), and all tools failed to generate compilable tests.

Experimenter validity. To mitigate the effects on the slow sam-
pling of subjects (the benchmarks) and objects (the competition tools)
we continuously introduce new software projects (which compose the
benchmarks’ CUTs) of varied complexity to account for their repre-
sentativeness on the real world, and include developers’ tests and a
random test generator Randoop as baseline to compare the perform-
ance of the competing tools. Nonetheless, the representativeness of
the testing field is weak as most of the tools have a random na-
ture and only Evosuite implements Search-Based Software Testing
techniques. We expect so that the public contest repository could
help to attract more tools from the field.

5 CONTEST RESULTS

Following the detailed results [9] for the last contest, we provide the
average results for each tool across six independent runs for budgets
of 10 seconds (Table 8), 60 seconds (available online [5]), 120 seconds

<p>| Table 3: Average (mean) coverage metrics and overall (sum) scores obtained across all CUTs. |
|---------------------------------|-----|-----|-----|-----|-----|-----|-----|-----|</p>
<table>
<thead>
<tr>
<th>Tool</th>
<th>Budget</th>
<th>Instruction Coverage</th>
<th>Branch Coverage</th>
<th>Mutation Coverage</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>------</td>
<td>--------</td>
<td>----------------------</td>
<td>-----------------</td>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>evosuite</td>
<td>0.00</td>
<td>0.47</td>
<td>0.97</td>
<td>0.40</td>
<td>0.96</td>
</tr>
<tr>
<td>t3</td>
<td>10</td>
<td>0.00</td>
<td>0.41</td>
<td>1.00</td>
<td>0.35</td>
</tr>
<tr>
<td>jtpexpert</td>
<td>0.00</td>
<td>0.36</td>
<td>0.99</td>
<td>0.35</td>
<td>0.96</td>
</tr>
<tr>
<td>randoop</td>
<td>0.00</td>
<td>0.37</td>
<td>0.98</td>
<td>0.28</td>
<td>0.94</td>
</tr>
<tr>
<td>evosuite</td>
<td>0.00</td>
<td>0.54</td>
<td>0.99</td>
<td>0.48</td>
<td>0.97</td>
</tr>
<tr>
<td>t3</td>
<td>60</td>
<td>0.00</td>
<td>0.47</td>
<td>1.00</td>
<td>0.42</td>
</tr>
<tr>
<td>jtpexpert</td>
<td>0.00</td>
<td>0.39</td>
<td>1.00</td>
<td>0.34</td>
<td>0.98</td>
</tr>
<tr>
<td>randoop</td>
<td>0.00</td>
<td>0.38</td>
<td>0.99</td>
<td>0.31</td>
<td>0.97</td>
</tr>
<tr>
<td>evosuite</td>
<td>0.00</td>
<td>0.56</td>
<td>1.00</td>
<td>0.56</td>
<td>0.98</td>
</tr>
<tr>
<td>t3</td>
<td>120</td>
<td>0.00</td>
<td>0.49</td>
<td>1.00</td>
<td>0.44</td>
</tr>
<tr>
<td>jtpexpert</td>
<td>0.00</td>
<td>0.39</td>
<td>1.00</td>
<td>0.34</td>
<td>0.98</td>
</tr>
<tr>
<td>randoop</td>
<td>0.00</td>
<td>0.39</td>
<td>1.00</td>
<td>0.31</td>
<td>0.99</td>
</tr>
<tr>
<td>evosuite</td>
<td>0.00</td>
<td>0.57</td>
<td>1.00</td>
<td>0.52</td>
<td>0.99</td>
</tr>
<tr>
<td>t3</td>
<td>240</td>
<td>0.00</td>
<td>0.49</td>
<td>1.00</td>
<td>0.44</td>
</tr>
<tr>
<td>jtpexpert</td>
<td>0.00</td>
<td>0.41</td>
<td>1.00</td>
<td>0.36</td>
<td>0.96</td>
</tr>
<tr>
<td>randoop</td>
<td>0.00</td>
<td>0.39</td>
<td>1.00</td>
<td>0.32</td>
<td>0.98</td>
</tr>
</tbody>
</table>

(available online [5]) and 4 minutes (Table 9). Table 3 summarizes
the average (mean) instruction coverage ($\text{cov}_i$), branch coverage
($\text{cov}_b$), and strong mutation coverage ($\text{cov}_m$) achieved across all
CUTs over different search budgets. This table also shows the overall
scores, which are computed as the sum of the average scores reached
across all CUTs and for each search budget, separately. As expected,
the coverage metrics and the scores increases when larger time is
given for test generation. This is true for all tools except: i) Randoop,
for which the coverage metrics remains mostly unchanged and
ii) T3 for budget 10s, which better manages the provided search
budget (Table 8) while other tools exceed the budget at some extent,
suffering from the scoring formula penalty (half coverage score in
the worst case according to [8]).

Comparison with manual and combined suites. Table 7 com-
paries the performance of manual tests written by the original de-
velopers and the performance of combined test suites—i.e., the test
suites obtained as the union of the test generated by all tools on the
same CUTs— using a 10s of search budget. For this analysis, we con-
sider only the 49 CUTs for which we could find manually-written
tests (i.e., excluding the project redisson that misses working manual
tests for the selected CUTs). It is worth reminding that DUBBO-2 is
a special case since its configuration missed critical dependencies
(e.g., javassist), and all tools failed to generate compilable tests.

To better understand our results, Figure 113 and Table 6 show the
performance on 49 CUTs of (1) each individual tool , (2) the
performance of the combined suites with 10 seconds budget, and (3)
manually-developed test cases from the projects’ developers. The
performance of the combined suites has been computed by building
one test suite per CUT and run consisting of all the test cases
generated by the four tools in each corresponding CUT and run
(only tests generated with a budget of 10 seconds). The performance
is analyzed by: (1) the achieved instruction coverage ($\text{cov}_i$), (2) the
branch coverage ($\text{cov}_b$) and (3) the mutation coverage ($\text{cov}_m$).

The main finding of this analysis is that combining the test cases
generated by individual tools can outperform the human developed
tests, for nearly all software projects selected as benchmarks in this
competition. This is particularly interesting if we consider that the
combined suites are built by combining the tests generated in only

12Instruction, branch and mutation coverages
13Vaxx = budgets, Vaxx = coverage percents, 5th_manual = 5th contest dev. tests
14REDISSON had no working manual tests available from the project
10 seconds of search budget. Therefore, larger budgets would likely result in better performance for the combined test suites over the individual tool results and the human-developed tests.

**Final scores and statistical results**. Table 4 shows the overall scores achieved by the four tools at different search budgets together with the ranking produced by the Friedman test. According to this test, the four tools statistically differ in terms of scores across all 59 CUTs ($p$-value $< 10^{-12}$). To better understand which pairs of tools statistically differ, Table 5 reports the $p$-values produced by the post-hoc Conover’s procedure. Note the $p$-values are adjusted with the Holm-Bonferroni correction procedure as required in case of multiple pairwise comparison. We note that evosuite achieves significantly higher scores than jtestexpert and randooop while there is no (or only marginal significance) with t3. On the other hand, t3 has marginally significantly higher scores than jtestexpert and randooop. Finally, the remaining two tools (i.e., jtestexpert and randooop) turn out to be statistically equivalent.

The permutation test reveals that there is a significant interaction between the achieved scores and the tools being used to generate the tests ($p$-value $< 10^{-12}$), further confirming the results of the Friedman test. There is a significant interaction between the performance scores, the McCabe’s cyclomatic complexity of the target CUTs and the testing tools ($p$-value $< 10^{-12}$). In other words, the scores of the alternative testing tools significantly differ for very complex CUTs (i.e., with large cyclomatic complexity). Moreover, the interaction between the testing tools and the adopted search budgets statistically interact with the achieved performance score ($p$-value $< 10^{-12}$). This means that the scores of the tools significantly increase when using larger search budgets.

### 6 CONCLUDING REMARKS

The combined tools performance analysis introduced in this edition reveals the power of a “super-tool” built over individual testing tools, which can potentially outperform developer tests, even with a small search budget like 10 seconds. This scenario brings an interesting field to explore in future competitions. Instead of running the tools isolated from each other, they could cooperate by trying to generate better test cases (more effective) in less time (more efficient).

### 7 ACKNOWLEDGEMENTS

Special thanks to Wishnu Prasetya whose interest on the combined tools performance pushed us to make it a reality. This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under the project DataME (TIN2016-80811-P); and by the H2020 EU project SUPERSEDE under agreement number 644018.
REFERENCES


Figure 1: Test performance on 49 CUTs (without REDISSON)
Table 8: Averaged results for 6 runs on 10 seconds time budget

<table>
<thead>
<tr>
<th>Model</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUBBO-3</td>
<td>61.8</td>
<td>80.0</td>
<td>29.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>80.0</td>
</tr>
<tr>
<td>DUBBO-6</td>
<td>81.2</td>
<td>95.1</td>
<td>7.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>81.2</td>
</tr>
<tr>
<td>DUBBO-7</td>
<td>56.0</td>
<td>82.3</td>
<td>16.7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>56.0</td>
</tr>
<tr>
<td>FASTJSON-3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100.0</td>
<td>4.4</td>
<td>0</td>
<td>4.4</td>
</tr>
<tr>
<td>FASTJSON-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>86.1</td>
<td>28.3</td>
<td>11.6</td>
</tr>
<tr>
<td>OKIO-4</td>
<td>0</td>
<td>15.5</td>
<td>15.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15.5</td>
</tr>
<tr>
<td>OKIO-9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>14.1</td>
<td>0</td>
<td>0</td>
<td>14.1</td>
</tr>
<tr>
<td>REDISSON-4</td>
<td>0</td>
<td>0</td>
<td>17.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17.0</td>
</tr>
<tr>
<td>REDISSON-6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>89.1</td>
<td>12.8</td>
<td>0</td>
</tr>
<tr>
<td>WEBMAGIC-2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WEBMAGIC-4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ZXING-8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 5: Averaged results for 6 runs on 240 seconds time budget

<table>
<thead>
<tr>
<th>Method</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>FASTJSON-1</td>
<td>18.0</td>
<td>0</td>
<td>0</td>
<td>161.6</td>
<td>75.0</td>
<td>0</td>
</tr>
</tbody>
</table>