SBST Tool Competition 2022

Alessio Gambi
alessio.gambi@uni-passau.de
University of Passau, Germany

Gunel Jahangirova
gunel.jahangirova@usi.ch
Software Institute - USI, Switzerland

Vincenzo Riccio
vincenzo.riccio@usi.ch
Software Institute - USI, Switzerland

Fiorella Zampetti
fiorella.zampetti@unisannio.it
University of Sannio, Italy

ABSTRACT

We report on the organization, challenges, and results of the tenth edition of the Java Unit Testing Competition as well as the second edition of the Cyber-Physical Systems (CPS) Testing Competition. **Java Unit Testing Competition**. Seven tools, i.e., BBC, EvoSuite, Kex, Kex-Reflection, Randoop, UTBot, and UTBot-Mocks, were executed on a benchmark with 65 classes sampled from four open-source Java projects, for two time budgets: 30 and 120 seconds. **CPS Testing Tool Competition**. Six tools, i.e., AdaFrenetic, AmbieGen, FreneticV, GenRL, EvoMBT and WOGAN competed on testing two driving agents by generating simulation-based tests. We considered one configuration for each test subject and evaluated the tools' effectiveness and efficiency as well as the failure diversity.

This paper describes our methodology, the statistical analysis of the results together with the competing tools, and the challenges faced while running the competition experiments.

CCS CONCEPTS

• **Software and its engineering** → *Search-based software engineering*; **Automatic programming**; **Software testing and debugging**.

KEYWORDS

Tool Competition, Software Testing, Test Case Generation, Unit Testing, Java, Cyber-Physical Systems, Autonomous Vehicles, Search Based Software Engineering

ACM Reference Format:
Alessio Gambi, Gunel Jahangirova, Vincenzo Riccio, and Fiorella Zampetti. 2022. SBST Tool Competition 2022. In *The 15th Search-Based Software Testing Workshop (SBST'22), May 9, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3526072.3527538

1 INTRODUCTION

This year we organized the tenth edition of the SBST Tool Competition. The competition has the goal to experiment with testing tools for a diversified set of systems and domains. As for recent years, we invited researchers to participate in the competition with their unit test generation tools for **Java** and system test generation tools for **Cyber-Physical System**s (CPSs). Java testing tools are evaluated against a benchmark with respect to code and mutation coverage, whereas CPS testing tools are evaluated against self-driving cars software in a simulation environment.

Report Structure: Section 2 and Section 3, report the organization, challenges, and results of the JUnit and CPS testing tool competitions.

SBST'22 , May 9, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9318-8/22/05…$15.00
https://doi.org/10.1145/3526072.3527538

2 THE JUNIT TESTING COMPETITION

The tenth edition of the Java Testing Tool Competition received the highest (six) number of submitted tools, namely BBC [5], EvoSuite [17], Kex [3], Kex-Reflection [6], UTBot [7] and UTBot-Mocks. Furthermore, similarly to previous editions, we used Randoop [29] as a baseline for comparison.

Each tool has been executed on 65 classes under test (CUTs) sampled from four out of six projects used also in the previous edition [31]. Starting from the results of the previous edition, we realized that the tools used for coverage and mutation analysis are not able to properly work on projects relying on a recent version of Java. For this reason, we relied on the four projects for which we can obtain both code and mutation coverage.

The competing tools have been compared by using line, branch and mutant coverage metrics, for two different time budgets, i.e., 30 and 120 seconds.

In order to guarantee a fair comparison among the competing tools, the execution of the tools for generating test suites and their evaluation, has been carried out by using a dockerized infrastructure [15] hosted on GitHub at:

https://github.com/JUnitContest/junitcontest.

The remainder of the JUnit testing competition report is structured as follows. Section 2.1 describes the benchmark being adopted once having described the selection criteria. Section 2.2 briefly describes the competing tools, while Section 2.3 presents the methodology for running the competition. Section 2.4, instead, reports and discusses the results. Finally, Section 2.5 concludes the report with remarks and ideas for future improvements.

2.1 The benchmark subjects of the JUnit Testing Competition

Similarly to previous editions, the selection of the projects and classes under test (CUTs) to use as benchmark for test case generation has been done by considering three criteria: (i) projects must belong to different application domains [17]; (ii) projects must be

Table 1: Description of the 10th edition of the benchmark.

Project	# Cand. CUTs	# Sampled CUTs
FastJSON	105	20
Guava	277	15
Seata	33	10
Spoon	169	15
Total	**584**	**65**

open-source for replicability purposes; and (iii) classes must not be trivial, i.e., the whole set of methods belonging to a class must have a McCabe's cyclomatic complexity higher than five, as well as, the class must contain at least one branch [30].

We focused on GitHub projects relying on Maven as build framework, and including JUnit4[1] test suites. Considering that, for the ninth edition [31] there were classes in the benchmark for which it was not possible to compute line, branch and mutant coverage metrics, we choose to identify the root cause behind that behaviour and we fixed it. For this reason, we considered four out of six projects from the ninth edition [31], by using a version for which it was possible to compute the metrics used for comparison. Specifically, we picked:

- *FastJSON (v1.2.50)* (https://github.com/alibaba/fastjson): a Java library to convert Java Objects into their JSON representation, as well as, a JSON string to an equivalent Java object;
- *Guava (v26.0)* (https://github.com/google/guava): a set of core Java libraries from Google, widely used on most Java projects within Google;
- *Seata (v0.5)* (https://github.com/seata/seata): an easy-to-use, highly performing distributed transaction solution;
- *Spoon (v7.0)* (https://github.com/INRIA/spoon): a meta programming library to analyze and transform Java source code.

Based on the time and resources available for running out the competition, as well as, considering the high number of competing tools, we have only sampled a limited number of CUTs using the approach adopted in the ninth edition [31]. Specifically, we relied on javancss[2] to identify for each production class in the system, the number of methods, and for each method its McCabe's cyclomatic complexity. Then, we filtered out all the classes that are non-testable (i.e., classes for which Randoop is not able to generate any test case using the time budget of 10 seconds), and classes for which at least one method has a McCabe's cyclomatic complexity lower than five. As a result, we obtained a set made up of 584 candidate classes, from which we randomly sampled 65 classes to use as our benchmark (as shown in Table 1).

2.2 JUnit Testing Competing Tools

Seven tools are competing in the tenth edition: BBC [5], EvoSuite [17], Kex [3], Kex-Reflection [6], Randoop [29], UTBot [7] and UTBot-Mocks.

BBC (Basic Block Coverage) [5] is a search-based unit test generation technique relying on EvoSuite [17]. Unlike the approach level and branch distance, which considers only information related to the coverage of explicit branches coming from conditional and loop statements, BBC also takes into account implicit branching (e.g., a run-time exception thrown in a branch-less method) denoted by the coverage level of relevant basic blocks in a control flow graph to drive the search process.

EvoSuite [17] uses evolutionary search to automatically generate test suites that aim to maximise code coverage. The test cases are represented in a genetic encoding consisting of variable-length sequences of Java statements. Standard evolutionary search operators such as selection, crossover, mutation are adapted to this representation. The current default evolutionary algorithm of EvoSuite is Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [30]. Furthermore, EvoSuite aims to generate test cases that are readable and free of test smells by applying a post-processing procedure that minimizes the generated test suite.

Kex [3], a tool implemented by JetBrains Research, works as a symbolic execution engine and uses the Satisfability Modulo Theory (STM) solver to perform the constraint solving. Specifically, by analyzing jar files, it constructs the control flow graph for each method and tries to cover each basic block in each method by generating sufficient input data. Finally, by using a novel backward search algorithm, namely Reanimator, Kex generates valid test cases from generated input parameters. **Kex-Reflection** [6], instead, is a Kex variant that uses Java reflection library and Unsafe API to generate test cases (instead of Reanimator approach used in Kex).

Randoop, used as a baseline in the context of the competition, generates unit tests using a feedback-directed random test generation, and collecting information from the execution of the tests as they are generated to reduce the number of redundant and illegal tests [29].

Finally, **UTBot** [7] is a tool implemented by Huawei Research that relies on symbolic execution to extract the information about the execution paths identified inside the method to derive the constraints that need to be met for traversing a desired path. By using the SMT solver, UTBot builds a model (i.e., a set of parameter values for the method under test) satisfying the above constraints with the aim of finding a model satisfying all possible execution paths of the method under test. **UTBot-mocks**, instead, is a variant of UTBot that relies on mocks for test case generation. Specifically, differently from UTBot, it mocks everything that does not belong to the Class Under Test (CUT), and it does not run a concrete execution.

2.3 Methodology of the JUnit Testing Competition

The methodology followed to run the competition is similar to the one adopted in the ninth edition [31]. It is important to remark that, due to time and resource constraints, and the high number of competing tools (six plus Randoop as baseline), we only considered two time budgets: 30 and 120 seconds.

Public contest repository. The complete contest infrastructure is released under a GPL-3.0 license and is available on GitHub [1]. Specifically, the repository contains the set of CUTs contributing to the tenth edition, as well as, the detailed summary of the results obtained by running each tool for each time budget.

Test generation and time budget. For each time budget, each tool has been executed ten times against each CUT to account for the randomness of the test case generation process [9].

[1] https://github.com/junit-team/junit4
[2] https://github.com/nokia/javancss

Execution environment. The infrastructure performed a total of 9,100 executions, i.e., 65 CUTs x 7 tools x 2 time budgets x 10 repetitions, to use for statistical analysis. For all the competing tools, we were able to run the planned number of executions.

To ensure a fair comparison, we ran each tool on the same dedicated machine, i.e., Google Cloud e2-highmem-8 virtual machine instances equipped with 8 vCPUs, 64 GB of RAM and 50 GB of memory. The dockerized version of the infrastructure, as well as the access to Google Cloud virtual machine instances allowed us to distribute the execution of the tools to different machines. We used a dedicated instance for each tool and time budget, employing 14 virtual machines instances overall. When selecting the configuration parameters of these instances we took into account the problems experienced in the previous edition [31], such as insufficient RAM and disk space.

Metrics computation. We compared the performance of the competing tools based on line, branch and mutation coverage metrics. Specifically, to compute both line and branch coverage metrics, we relied on JaCoCo [2], an open-source toolkit for measuring and reporting Java code coverage. For mutation analysis, instead, we relied on PITest [4], considering five minutes as the maximum amount of time available for mutation analysis for each CUT, and a timeout of one minute for each mutant being generated. Among all the mutants being generated by PITest, for CUTs with more than 200 mutants we randomly sampled only 33% of them, while for CUTs with more than 400 mutants we sampled 50% of them for the analysis.

Statistical analysis. Statistical tests are used to support the obtained results. Specifically, we use the Friedman test [38], a non-parametric statistical test, to detect differences in treatments across multiple test attempts, i.e., for assessing whether the scores over the different CUTs and time budgets achieved by the competitors tools are significantly different from each other. On top of this, we also computed the post-hoc Conover's test [35] to determine for which pair of tools the significance actually holds, once having adjusted them with the Holm-Bonferroni procedure [8].

2.4 Results of the JUnit Testing Competition

Table 2 presents for each tool and for each time budget the minimum, mean, median and maximum number of the generated test cases. As expected, the increase in the time budget for the generation process leads to the increase in the number of generated test cases. However, BBC and EvoSuite show a different trend. Specifically, as shown in Table 2, for BBC only the maximum number of generated tests is higher with a budget of 120 seconds, while for EvoSuite all descriptive statistics show a decreasing trend. Furthermore, for 2 CUTs in our benchmark, both Kex and Kex-Reflection were not able to generate any test case, with Kex showing this behaviour also for additional 3 CUTs.

To provide a deeper insight on the obtained results for each competing tool, Figures 1, 2, and 3 report the percentage of lines, branches and mutants being covered by the seven competing tools, for each specific time budget. Note that, the mutation coverage is the ratio between the number of mutants that were killed by at least one test and the total number of mutants being generated.

Table 2: Statistics on number of test cases generation for each tool and each time budget.

Tool	Time budget	Min	Mean	Median	Max
BBC	30	0	29	20	160
	120	0	23	14	192
EvoSuite	30	0	42	27	250
	120	0	29	18	211
Kex	30	0	17	7	159
	120	0	28	13	300
Kex-Reflection	30	0	50	28	312
	120	0	65	38	369
Randoop	30	1	3,705	892	106,143
	120	0	10,654	2,244	303,878
UTBot	30	0	42	31	194
	120	0	58	39	350
UTBot-Mocks	30	2	44	34	170
	120	2	56	35	263

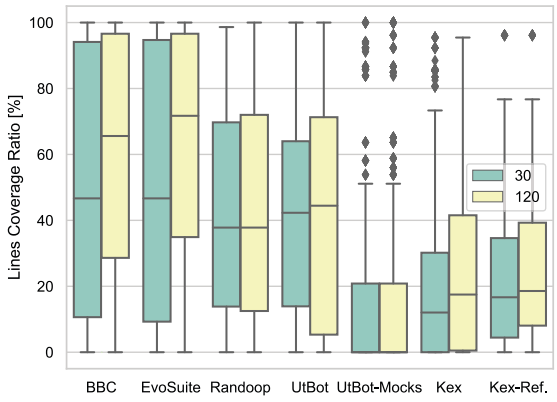

Figure 1: Lines Coverage Ratio for the 7 competing tools for 30 and 120 seconds.

Unsurprisingly, when increasing the time budget for generation purposes, the median line and branch coverage also (slightly) increase. Specifically, as shown in Figure 1, the line coverage ratio for EvoSuite moves from 46.7% (30 seconds) up to 71.7% (120 seconds), while for BBC the increase is from 46.7% up to 65.6%.

When it comes to branch coverage, as shown in Figure 2, only BBC and EvoSuite are able to reach a median branch coverage greater than 40% for each of the two time budgets. EvoSuite performs the best, with its median branch coverage ratio being equal to 60% for the time budget of 120 seconds. UTBot shows a median branch coverage increasing from 22.9% (30 seconds) up to 33.3% with a time budget of 120 seconds. Furthermore, even though for some classes UTBot-Mocks achieves a maximum branch coverage of 100%, its median value is equal to 0. A similar behavior is shown by Kex when using a time budget of 30 seconds. In contrast, Kex-Reflection has a median branch coverage of 5.0% and 9.4% for each time budget respectively.

In terms of mutation coverage ratio (see Figure 3), quite surprisingly, for five out of seven tools, i.e., BBC, EvoSuite, Randoop, UTBot, and Kex there is a decreasing trend when increasing the

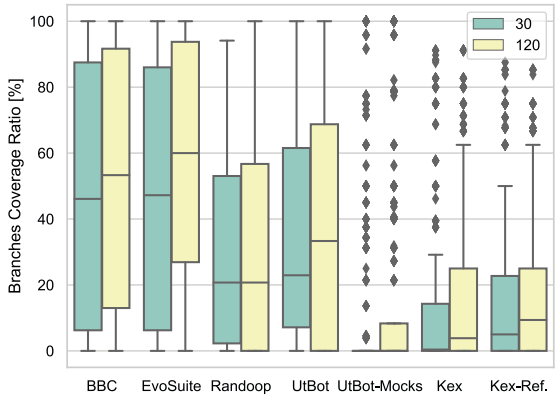

Figure 2: Branches Coverage Ratio for the 7 competing tools for 30 and 120 seconds.

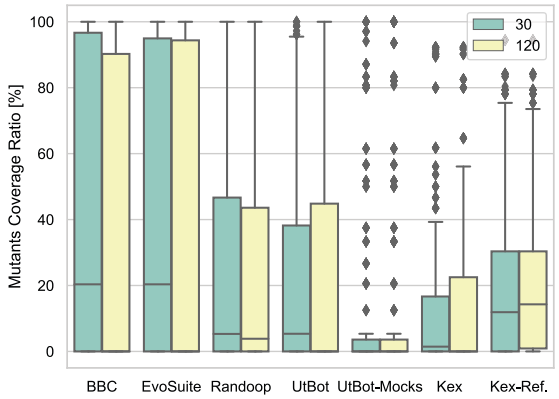

Figure 3: Mutants Coverage Ratio for the 7 competing tools for 30 and 120 seconds.

time budget up to 120 seconds. Furthermore, for a time budget of 30 seconds, BBC and EvoSuite are able to reach a median mutants coverage ratio of $\simeq$ 20%, while, for a time budget of 120 seconds, Kex-Reflection is the only one showing a mutant coverage greater than 10% among the remaining 6 tools.

Moreover, we have looked at the percentage of mutants being killed among the ones being generated. Only for Kex and Kex-Reflection these percentages never reach 100% while stopping at a maximum of 91.7% and 94.4%, respectively. Furthermore, while considering only the classes for which all the competing tools are able to kill at least one mutant, we found that BBC is the one showing the highest median percentage, 72.6% and 96.7% for 30 and 120 seconds, respectively, followed by EvoSuite (i.e., with 67.4% and 94.4%). It is important to remark that BBC and EvoSuite might be showing similar results since BBC is built on top of EvoSuite.

Last but not least, we report the scores and ranking achieved by the tools considering the two different time budgets being used. The final score formula [15] has been created and improved during the previous editions of the tool competition and takes into account the line and branch coverage, the mutation score, and the time budget used by the generator. Moreover, it applies a penalty for flaky and non-compiling tests. We observed a final score of 380.57 for EvoSuite, 371.06 for BBC, 263.43 for UTBot, 246.3 for Randoop, 133.18 for Kex-Reflection, 133.03 for UTBot-Mocks and 80.94 for Kex. In terms of ranking, instead, we have EvoSuite (2.4), followed by BBC (2.55), UTBot (3.58), Randoop (3.65), Kex (5.21), UtBot-Mocks (5.30), and Kex-Reflection (5.31).

2.5 Conclusions and Final Remarks of the JUnit Testing Tool Competition

This year marks the tenth edition of the Java Unit Testing Competition. In comparison to the previous editions, this year we have the highest number of competitors, namely UTBot, UTBot-Mocks, Kex, Kex-Reflection, EvoSuite, BBC, and Randoop as a baseline. As per results of this year, the best performing tool is EvoSuite followed by BBC, while Kex seems to perform the worst on the selected CUTs.

The analysis of collected results by the organisers of the competition and by the participants revealed that for some of the generated test suites it was not possible to perform mutation analysis. The most likely cause of this is that PIT requires some special treatment when it comes to setting up its classpath, as we did not experience this problem when using JaCoCo for both line and branch coverage. We plan to investigate this issue further, to identify the definite root cause of the problem and to perform a fix for the next editions of the competition. In addition, we envision several other possibilities for improvement such as: (i) extending the list of criteria used for the evaluation purposes by adding new ones such as performance-awareness [21] and readability [33]); (ii) widening the scope of the competition to the tools supporting the testing of more complex applications (e.g., cloud-based systems [27]); and (iii) considering to extend the infrastructure to support other programming languages (e.g., Python [26]).

3 THE CYBER-PHYSICAL SYSTEMS TESTING TOOL COMPETITION

Self-driving cars are safety-critical CPSs which are growing in relevance within the SBST research community [36] and industry [12]. Therefore, we organized the CPS Testing Tool Competition to encourage researchers to investigate the problem of testing such CPSs and provide a shared framework for benchmarking test generators.

This second edition received six submissions, namely AdaFrenetic, AmbienGen, FreneticV, GenRL, EvoMBT and WOGAN. Compared to the previous edition, the number of participating teams increased by 50% (from 4 to 6); 2 out of 4 teams from the previous edition joined also this year's competition. The number of submitted tools increased by $\times$1.2 (from 5 to 6).

To ease the test generators' development, we provided the participants with an open-source, extensible test infrastructure [20]. Our infrastructure allows test generators to work on abstract, hence simpler, test case representations and hides the details of implementing and executing the generated test cases as physically accurate simulations using BeamNG.tech [10]. Since the past edition, we integrated in the test infrastructure an additional Deep Learning (DL)-based driving agent and a test feature extraction component [41] which characterizes the generated tests.

As test subjects, we selected BeamNG.AI, the driving agent shipped with the driving simulator, and Dave-2, a DL-based driving agent based on the architecture proposed by Bojarski et al. [11]. Both test subjects have been extensively studied in previous research [18, 25, 37, 41]. In particular, BeamNG.AI is the same test subject used in the first edition of this competition [32].

3.1 Simulation-based Testing of Self-Driving Car Software

For this competition, we considered self-driving cars as representative instance of CPS and tested them in computer simulations, which constitute a viable alternative to expensive, ineffective and dangerous field operational tests. Therefore, we challenged the participants to implement generators of *virtual tests*. To reduce onboarding efforts, we kept the same setup of last year's competition and considered driving scenarios taking place on flat roads surrounded by green grass. For simplicity, we used fixed environmental conditions (i.e., weather and lighting set to sunny day without fog) and road layout (i.e., single roads with two fixed-width lanes).

We focused on testing lane-keeping assist systems (LKAS) by defining the following driving task: driving without going off the lane from a given starting position, i.e., the beginning of a road, to a target position, i.e., the end of that road. Given this driving task, the goal of the test generators was to create challenging, yet valid, virtual roads that cause the test subjects to drive off the right lane.

Our framework represents virtual roads as sequences of *road points* defined on a two-dimensional map. It interpolates the road points using cubic splines and considers the first and last road points as the starting and target position of the driving tasks. Despite the number of possible road point sequences is extremely large, not all of them result in valid roads [19, 32, 37]. In particular, valid roads (i) do not self-intersect; (ii) do not contain overly-sharp turns; and (iii) are fully contained in the map. Our framework ensures that only valid tests are executed as driving simulations. Consequently, invalid tests do not count as failed tests; instead, they are taken into account for assessing the tools' generation effectiveness.

3.2 The Tools of the CPS Testing Competition

Six tools competed in this edition of the CPS Testing Tool Competition: AdaFrenetic [40], AmbieGen [23], FreneticV [14], GenRL [39], EvoMBT [16], and WOGAN [34]. All the six submissions are novel but three of them (i.e., 50%) extend tools submitted in the previous competition, suggesting that our initiative already had an impact on the community. Specifically, AdaFrenetic and FreneticV extend Frenetic [13], whereas AmbieGen is an evolution of SWAT [22]. Noticeably, in this competition, tools implemented a wider range of approaches than last year. These approaches range from the standard feedback-driven search to model-based testing, Reinforcement Learning and Deep Generative Models.

3.3 Methodology of the CPS Testing Competition

3.3.1 Subject Systems of the CPS Testing Competition. We evaluated the competing tools using the BeamNG.tech driving simulator [10]. We chose two test subjects widely used in SBST literature: BeamNG.AI, BeamNG.tech simulator's built-in driving agent, and Dave-2, a DL-based driving agent. We made the driving simulator and the test subjects available to the competitors before the submission but did not disclose the experimental setup.

BeamNG.AI knows the geometry of the whole road and utilizes a complex optimization process to plan trajectories that drive the ego-car as close as possible to the speed limit, while keeping the vehicle as much as possible inside the lane. Dave-2, instead, is an end-to-end approach that uses a DL architecture consisting of three convolutional layers, followed by five fully-connected layers [11] to predict steering angles from images taken by the ego-car's onboard camera. We trained Dave-2 with images captured by BeamNG.tech's camera sensors paired with steering angles of the ego-car collected while BeamNG.AI was driving at the center of the lane. Since the Dave-2 implements imitation learning, we trained it only with positive examples, i.e., we discarded training data in which the ego-car drove out of the lane.

3.3.2 Goal and Metrics. The goal of the competition is to generate the highest number of diverse failure-inducing inputs, i.e., valid virtual roads that cause the ego-car to drive out of the lane. Our infrastructure detects a failure each time the ego-car (partially) drives outside the lane, i.e., if the area of the ego-car outside the lane is above a configurable threshold. For instance, a 0.5 threshold triggers a failure when more than half of the ego-car lies outside the lane. We label those failures as *Out of Bound (OOB)* episodes and refer to the threshold value controlling them as *OOB tolerance* (*OOB Tol.* in Table 3).

We limit the execution of the test generation to be within a given time budget. To provide more flexibility during test generation, we provide the competitors with two time budgets: *generation* and *execution*. The former accounts for the time allotted to the tools for generating test cases, whereas the latter corresponds to the time available for executing simulations. Notably, we improved results reproducibility by measuring the execution budget in *simulated* seconds, i.e., logical time.

Deciding which test generator is objectively the best is difficult and, currently, remains an open challenge. As done in the previous edition, we ranked tools by considering various aspects of test generation, including their ability to generate valid test cases and trigger OOBs. In this edition, we also introduced a new metric to measure the *diversity* of the failure-inducing tests. In the following, we describe the metrics used for ranking the test generators.

Test Generation Effectiveness. Effective test generators wisely use their generation budget and should mostly produce valid tests, i.e., they do not waste the generation budget in generating invalid tests. Therefore, we compute the test generation effectiveness as the ratio of valid tests over all the generated tests.

Test Generation Efficiency. Efficient test generators generate many tests within the test budget. We measure test generation efficiency as the inverse of the average test generation time, i.e., the ratio of the number of generated tests to the used generation budget, and normalized the efficiency by empirically computing the minimum and maximum efficiency values over all the runs of all the tools.

Failure-inducing Test Diversity. Tests are useful when they trigger failures and when those failures are diverse because they give developers information about the (mis-)behavior of the ego-car in various

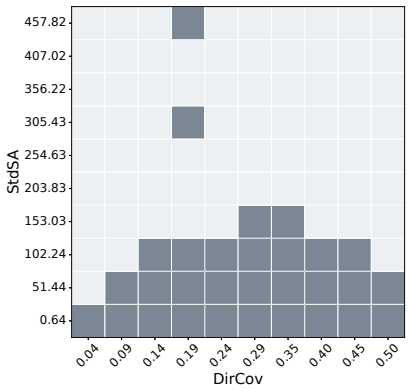

Figure 4: Feature Map showing the OOB coverage achieved by the tools in the BeamNG.AI configuration.

execution conditions. Characterizing failure-inducing inputs is far from trivial, and basic methods to establish input similarity using distance metrics are inadequate to capture driving agents' behavior. Therefore, we consider high-level features that meaningfully characterize the tests. Specifically, we selected two features that have been empirically assessed by Zohdinasab et al. [41] and Jahangirova et al. [24]: Direction Coverage (DirCov) and Standard deviation of the Steering Angle (StdSA).

DirCov is a *structural* feature that indicates how many directions (e.g., N, S, E, W) the virtual road covers. We measure DirCov as the number of different angular sectors covered by the directions of the road segments. In particular, we consider 36 sectors, each spanning 10 degrees. StdSA, instead, is a *behavioral* feature that measures the standard deviation of the sequence of the ego-car's steering angles collected during test execution. StdSA characterizes the agent's driving style.

We use these features to define bi-dimensional *feature maps* where the failure-inducing inputs are positioned based on their feature values, so that similar failure-inducing tests occupy neighboring (or the same) map's cells (see Figure 4). Such a feature map filled with a tool's failure-inducing inputs enables us to measure the OOB coverage achieved by that tool, hence quantifying how many diverse failure-inducing tests it generated. This approach is inspired by recent work on Illumination search [41] and has been already used for test selection [28].

Given a failure-inducing input, we argue that not all of its content determines the OOB. Therefore, before computing DirCov and StdSA, we extract the portion of the road relevant to the failure and compute the features only for that road segment. We define the road segment relevant to a failure as the segment of the road *around* the OOB location, to account for the most recent activity of the ego-car (before OOB location) and the trajectory it might have planned (after OOB location). In this competition, we consider 60m-long relevant segments (30m before and 30m after the OOB). To compute the coverage of the feature space achieved by a tool, we build a 10×10 feature map and detect the cells covered by all the tools (e.g., see Figure 4), then we measure how many of those cells were covered by the considered tool. Given the values of Effectiveness (Effect), Efficiency (Effic) and Diversity (Div), we compute the Score to rank the test generators as follows:

$$\text{Score} = \alpha * \text{Div} + \beta * \text{Effic} + \gamma * \text{Effect} \quad (1)$$

where α, β and γ define the relative contribution of each metric towards the final value. For ranking the test generators, we gave more importance to failure-inducing test diversity ($\alpha = 0.6$) than generation efficiency ($\beta = 0.2$) and effectiveness ($\gamma = 0.2$) because triggering failures is the main goal of testing.

Table 3: Experimental Setups

Name	Map Size	Speed Limit	Gen. Budget	Exec. Budget	OOB Tol.
BeamNG.AI	200m × 200m	70 Km/h	1 h	2 h	0.85
Dave-2	200m × 200m	35 Km/h	1 h	2 h	0.1

3.4 Experimental Procedure

We ran each tool 10 times in the two experimental setups reported in Table 3: BeamNG.AI and Dave-2.

BeamNG.AI features the BeamNG.AI agent driving up to 70Km/h, an OOB tolerance value of 0.85, a generation budget of 1h and execution budget of 2h. This setup is similar to the one adopted in the previous edition of this competition. Dave-2 features the Dave-2 agent driving up to 35Km/h, an OOB tolerance of 0.1, and the same time budgets as BeamNG.AI. In this setup, Dave-2 drives more slowly, which makes it harder to cause OOBs, but we use a more sensible oracle to compensate for the additional difficulty.

To ensure a fair comparison, we ran each tool the same number of times and on dedicated machines. Specifically, we ran the experiments in the BeamNG.AI configuration on a desktop PC running Microsoft Windows 10 Enterprise and featuring a quad-core Intel i7-7700K CPU @ 4.20 GHz, 16 GB of Memory @ 2400Z Mhz, and an NVidia GeForce GTX 1080 GPU. Instead, we ran the experiments in the Dave-2 configuration on Google Cloud `n1-standard-8` virtual machines running Microsoft Windows Server 2016 Datacenter and featuring 8 vCPUs, 30 GB of Memory and an NVIDIA Tesla P100. For all the experiments, we used the version 0.24.0.2 of BeamNG.tech.

3.5 Results of the CPS Testing Competition

Test Generation Effectiveness and Efficiency. Table 4 reports the average count of valid (col. *Val.*) and invalid (col. *Inval.*) test cases produced by each tool (col. *Tool*) in each configuration (col. *Config*). Since the tools adopt different approaches to test generation, we report in the table also the percentage of the generation budget actually used by each tool (col. *Time*), to further characterize their efficiency. From these results, we can observe that all the tools followed similar trends across the two configurations.

WOGAN generated the highest number of test cases using the smallest portion of generation time, thus resulting the most efficient test generator in this competition. However, WOGAN also produced the highest number of invalid roads (more than 50%). Those invalid tests, mostly caused by overly sharp turns, drastically reduce this tool effectiveness. On the contrary, GenRL and EvoMBT never produced invalid test cases, thus resulting the most effective test generators in this competition. However, EvoMBT produced

Table 4: Test Generation Efficiency and Effectiveness. Valid and invalid tests and used generation time.

Config.	Tool	Val.	Inval.	Time (%)
BeamNG.AI	AdaFrenetic	125.6	583.7	100
	AmbieGen	494.8	17.2	95.3
	FreneticV	483.6	0.9	71.9
	GenRL	350	**0**	81.3
	EvoMBT	27	**0**	87.5
	WOGAN	**1146.1**	1453.5	3.2
Dave-2	AdaFrenetic	123.2	585.8	100
	AmbieGen	386.2	13.5	51.1
	FreneticV	339.4	0.4	51
	GenRL	206.7	**0**	33.8
	EvoMBT	22.1	**0**	85.1
	WOGAN	**839.8**	1182.3	2.2

Table 5: Final scores. The results are averaged across 10 repetitions. Bold text indicates the best values.

Config.	Tool	Effic.	Effect.	Div.	Score
BeamNG.AI	AdaFrenetic	.001	.035	.146	.183
	AmbieGen	.001	.193	**.350**	**.544**
	FreneticV	.001	**.200**	.246	.447
	GenRL	.001	**.200**	.036	.237
	EvoMBT	.000	**.200**	.016	.216
	WOGAN	**.148**	.090	.276	.514
Dave-2	AdaFrenetic	.001	.035	.008	.044
	AmbieGen	.001	.193	**.138**	**.333**
	FreneticV	.001	**.200**	.101	.302
	GenRL	.001	**.200**	.010	.211
	EvoMBT	.000	**.200**	.000	.200
	WOGAN	**.148**	.086	.028	.262

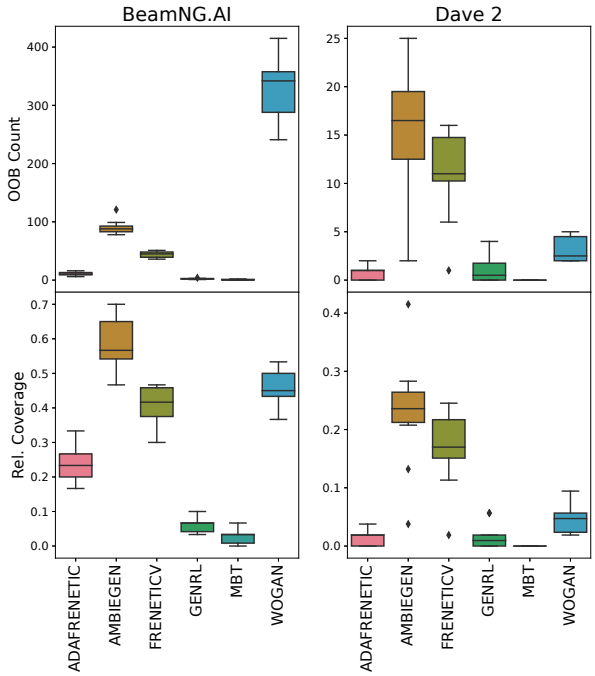

Figure 5: Benchmark Results. The plots report the number of detected failures (top) and the achieved OOB coverage (bottom) in each configuration.

the lowest number of test cases despite using almost the whole generation budget. AmbieGen and Frenetic produced many valid test cases and relatively few invalid ones, whereas AdaFrenetic used the whole generation budget and generated more invalid tests than valid ones.

Failure-inducing Test Diversity. Figure 5 shows the distribution of triggered OOBs (top) and the relative feature maps' coverage achieved by each tool (bottom) in both configurations.

In the BeamNG.AI configuration, WOGAN triggered the largest number of OOBs but did not achieve the highest coverage, which indicates that many of the triggered OOBs had similar features. Instead, AmbieGen obtained the highest coverage by triggering a remarkable number of OOBs in this configuration.

In the Dave-2 configuration, AmbieGen found the highest number of OOBs and achieved the highest coverage. FreneticV exposed many OOBs and achieved remarkable coverage, whereas the other tools found only a few OOBs and achieved low coverage.

Final Score. AmbieGen won this competition since it is the tool that reached the highest score in both configurations (i.e, 0.544 in BeamNG.AI and 0.333 in Dave-2). WOGAN ranked in second place, by achieving the second best score for BeamNG.AI (0.514) and the third best score forDave-2 (0.262), while FreneticV achieved the second best for Dave-2 (0.302) and the third best for BeamNG.AI (0.447), reaching the third place.

3.6 Conclusions and Final Remarks of the CPS Testing Competition

The SBST CPS testing tool competition aims to tackle the challenge of evaluating and comparing test CPSs generators. In this second edition, six tools competed by testing two test subjects (i.e., BeamNG.AI and Dave-2) and generated inputs that triggered failures of both systems. FreneticV, GenRL and EvoMBT always generated valid tests; WOGAN generated the highest number of tests in the shortest time, and Ambiegen triggered the highest number of diverse failures, hence it won the competition. Compared to the previous edition, we improved our testing infrastructure and methodology in various aspects, including integrating a new DL-based driving agent and introducing a novel metric to assess failure diversity in terms of feature space coverage. Nevertheless, we believe that the crucial problems of testing CPS systems and objectively evaluating CPS testing tools are not yet solved and further editions of this competition should investigate them.

ACKNOWLEDGMENTS

We thank the participants in both competitions for their invaluable contribution, BeamNG.GmbH for providing their driving simulator, and the Google Open Source Security Team for providing access to Google Cloud. This work was partially supported by the H2020 project PRECRIME (ERC Grant Agreement n. 787703), the H2020 project COSMOS (Project n. 957254-COSMOS), and the DFG project STUNT (DFG Grant Agreement n. FR 2955/4-1).

REFERENCES

[1] 2021. Contest Infrastructure. https://github.com/JUnitContest/junitcontest. [Online; accessed 23-02-2021].
[2] 2021. JaCoCo. https://www.jacoco.org/jacoco/trunk/doc/. [Online; accessed 23-02-2021].
[3] 2021. Kex. https://github.com/vorpal-research/kex/tree/sbst-21. [Online; accessed 23-02-2021].
[4] 2021. PiTest. http://pitest.org/. [Online; accessed 23-02-2021].
[5] 2022. BBC. https://github.com/pderakhshanfar/evosuite/tree/BBC. [Online; accessed 13-03-2022].
[6] 2022. Kex-Reflection. https://github.com/vorpal-research/kex/tree/sbst2022-reflection. [Online; accessed 28-02-2022].
[7] 2022. UTBot. https://github.com/UnitTestBot.
[8] Hervé Abdi. 2010. Holm's sequential Bonferroni procedure. *Encyclopedia of research design* 1, 8 (2010), 1–8.
[9] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Softw. Test. Verif. Reliab.* 24, 3 (May 2014), 219–250. https://doi.org/10.1002/stvr.1486
[10] BeamNG GmbH. 2021. *BeamNG.tech*. https://www.beamng.gmbh/research
[11] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). arXiv:1604.07316 http://arxiv.org/abs/1604.07316
[12] Markus Borg. 2021. The AIQ Meta-Testbed: Pragmatically Bridging Academic AI Testing and Industrial Q Needs. In *Software Quality: Future Perspectives on Software Engineering Quality*, Dietmar Winkler, Stefan Biffl, Daniel Mendez, Manuel Wimmer, and Johannes Bergsmann (Eds.). Springer International Publishing, Cham, 66–77.
[13] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. 2021. Frenetic at the SBST 2021 Tool Competition. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 36–37. https://doi.org/10.1109/SBST52555.2021.00016
[14] Ezequiel Castellano, Stefan Klikovits, Ahmet Cetinkaya, and Paolo Arcaini. 2022. FreneticV tool. https://github.com/ERATOMMSD/freneticV-sbst22.
[15] Xavier Devroey, Alessio Gambi, Juan Pablo Galeotti, René Just, Fitsum Kifetew, Annibale Panichella, and Sebastiano Panichella. 2021. JUGE: An Infrastructure for Benchmarking Java Unit Test Generators. https://doi.org/10.48550/arXiv.2106.07520
[16] Raihana Ferdous, Chia kang Hung, Fitsum Kifetew, Davide Prandi, and Angelo Susi. 2022. EvoMBT tool. https://github.com/iv4xr-project/iv4xr-mbt.
[17] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2 (dec 2014), 1–42. https://doi.org/10.1145/2685612
[18] Alessio Gambi, Tri Huynh, and Gordon Fraser. 2019. Generating effective test cases for self-driving cars from police reports. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 257–267. https://doi.org/10.1145/3338906.3338942
[19] Alessio Gambi, Marc Müller, and Gordon Fraser. 2019. AsFault: testing self-driving car software using search-based procedural content generation. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 27–30. https://doi.org/10.1109/ICSE-Companion.2019.00030
[20] Alessio Gambi and Vincenzo Riccio. 2021. SBST CPS Tool Competition Infrastructure. https://github.com/se2p/tool-competition-av.
[21] G. Grano, C. Laaber, A. Panichella, and S. Panichella. 2019. Testing with Fewer Resources: An Adaptive Approach to Performance-Aware Test Case Generation. *IEEE Transactions on Software Engineering* (2019), 1–1.
[22] Dmytro Humeniuk, Giuliano Antoniol, and Foutse Khomh. 2021. SWAT tool at the SBST 2021 Tool Competition. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 42–43. https://doi.org/10.1109/SBST52555.2021.00019
[23] Dmytro Humeniuk, Foutse Khomh, and Giuliano Antoniol. 2022. AmbieGen tool. https://github.com/dgumenyuk/tool-competition-av.
[24] Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. 2021. Quality Metrics and Oracles for Autonomous Vehicles Testing. In *Proceedings of 14th IEEE International Conference on Software Testing, Verification and Validation (ICST '21)*. IEEE, 194–204.
[25] Sajad Khatiri, Christian Birchler, Bill Bosshard, Alessio Gambi, and Sebastiano Panichella. 2021. Machine Learning-based Test Selection for Simulation-based Testing of Self-driving Cars Software. *CoRR* abs/2111.04666 (2021). arXiv:2111.04666 https://arxiv.org/abs/2111.04666
[26] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020 (Lecture Notes in Computer Science)*, Aldeida Aleti and Annibale Panichella (Eds.), Vol. 12420. Springer, 9–24. https://doi.org/10.1007/978-3-030-59762-7_2
[27] Diego Martin and Sebastiano Panichella. 2019. The cloudification perspectives of search-based software testing. In *International Workshop on Search-Based Software Testing, SBST@ICSE 2019*, Alessandra Gorla and José Miguel Rojas (Eds.). IEEE / ACM, 5–6. https://doi.org/10.1109/SBST.2019.00009
[28] Vuong Nguyen, Stefan Huber, and Alessio Gambi. 2021. SALVO: Automated Generation of Diversified Tests for Self-driving Cars from Existing Maps. In *2021 IEEE International Conference on Artificial Intelligence Testing, AITest 2021, Oxford, United Kingdom, August 23-26, 2021*. IEEE, 128–135. https://doi.org/10.1109/AITEST52744.2021.00033
[29] Carlos Pacheco and Michael D Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, Vol. 2. ACM Press, 815. https://doi.org/10.1145/1297846.1297902
[30] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435
[31] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. SBST Tool Competition 2021. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 20–27. https://doi.org/10.1109/SBST52555.2021.00011
[32] Sebastiano Panichella, Alessio Gambi, Fiorella Zampetti, and Vincenzo Riccio. 2021. SBST Tool Competition 2021. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 20–27. https://doi.org/10.1109/SBST52555.2021.00011
[33] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The impact of test case summaries on bug fixing performance: an empirical investigation. In *International Conference on Software Engineering, ICSE 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 547–558. https://doi.org/10.1145/2884781.2884847
[34] Jarkko Peltomaki, Frankie Spencer, and Ivan Porres. [n.d.]. Wasserstein generative adversarial networks for online test generation for cyber physical systems. In *15th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2022, Pittsburgh, PA, USA, May 9, 2022*.
[35] Dulce G Pereira, Anabela Afonso, and Fátima Melo Medeiros. 2015. Overview of Friedman's test and post-hoc analysis. *Communications in Statistics-Simulation and Computation* 44, 10 (2015), 2636–2653.
[36] Vincenzo Riccio, Gunel Jahangirova, Andrea Stocco, Nargiz Humbatova, Michael Weiss, and Paolo Tonella. 2020. Testing machine learning based systems: a systematic mapping. *Empir. Softw. Eng.* 25, 6 (2020), 5193–5254. https://doi.org/10.1007/s10664-020-09881-0
[37] Vincenzo Riccio and Paolo Tonella. 2020. Model-based Exploration of the Frontier of Behaviours for Deep Learning System Testing. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20)*. Association for Computing Machinery, 13 pages. https://doi.org/10.1145/3368089.3409730
[38] Michael R Sheldon, Michael J Fillyaw, and W Douglas Thompson. 1996. The use and interpretation of the Friedman test in the analysis of ordinal-scale data in repeated measures designs. *Physiotherapy Research International* 1, 4 (1996), 221–228.
[39] Luigi Libero Lucio Starace, Andrea Romdhana, and Sergio Di Martino. 2022. GenRL tool. https://github.com/luistar/GenRL-testing-tool.
[40] SongYang Tan and Ming Fan. 2022. AdaFrenetic tool. https://github.com/TayYim/adafrenetic-sbst22.
[41] Tahereh Zohdinasab, Vincenzo Riccio, Alessio Gambi, and Paolo Tonella. 2021. DeepHyperion: exploring the feature space of deep learning-based systems through illumination search. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 79–90. https://doi.org/10.1145/3460319.3464811