SBST Tool Competition 2021

Sebastiano Panichella*, Alessio Gambi†, Fiorella Zampetti‡ and Vincenzo Riccio§
*Zurich University of Applied Science (ZHAW), Zurich, Switzerland
Email: panc@zhaw.ch
†University of Passau, Passau, Germany
Email: alessio.gambi@uni-passau.de
‡University of Sannio, Benevento, Italy
Email: fiorella.zampetti@unisannio.it
§Software Institute - USI, Lugano, Switzerland
Email: vincenzo.riccio@usi.ch

***Abstract*—We report on the organization, challenges, and results of the ninth edition of the Java Unit Testing Competition as well as the first edition of the Cyber-Physical Systems Testing Tool Competition. Java Unit Testing Competition. This year, five tools, Randoop, UtBot, Kex, Evosuite, and EvosuiteDSE, were executed on a benchmark with (i) new classes under test, selected from three open-source software projects, and (ii) the set of classes from three projects considered in the eighth edition. We relied on an improved Docker infrastructure to execute the tools and the subsequent coverage and mutation analysis. Given the high number of participants, we considered only two time budgets for test case generation: thirty seconds and two minutes.**
Cyber-Physical Systems Testing Tool Competition. Five tools, Deeper, Frenetic, GABExplore, GABExploit, and Swat, competed on testing self-driving car software by generating simulation-based tests using our new testing infrastructure. We considered two experimental settings to study test generators' transitory and asymptotic behaviors and evaluated the tools' test generation effectiveness and the exposed failures' diversity.

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

I. Introduction

This year we organized the ninth edition of the SBST tool competition. The competition has the goal to experiment with testing tools for a diversified set of traditional and emerging systems and domains. Specifically, as for recent years, we invited researchers to participate in the competition with their unit test generation tool for **Java**. Tools are evaluated against a benchmark with respect to code coverage and mutation score. In addition to the traditional Java tool competition, we also organized the first **Cyber-Physical System** (CPS) testing competition on self-driving cars simulation environments. Specifically, in collaboration with the BeamNG research team, this competition focuses on the generation of scenarios using BeamNG self-driving cars simulator. As follow, Section II and Section III, report the organization, challenges, and results of the JUnit and CPS testing tool competitions.

II. The JUnit Testing Competition

This year, the ninth edition of the Java Unit Testing Competition, has as participants Randoop [1], UtBot, Kex [2], Evosuite [3] and EvosuiteDSE [4]. Each tool, has been executed with a time budget of thirty seconds and two minutes on 98 classes under test selected from three new open source projects and three open source projects already used in the previous edition [5] of the competition.

We have compared the competitors' tools by using both line and branch coverage metrics, as well as, mutation analysis to evaluate the potential of the generated test suites in revealing fault, for each time budget. Both the execution of the tools for generating test suites, and their evaluation, i.e., computing code coverage metrics together with performing mutation analysis, has been carried out by using a dockerized infrastructure hosted on Github[1].

The remainder of the JUnit testing competition report is structured as follows. Section II-A describes the benchmark being adopted once having described the selection criteria. Section II-B briefly describes the contestants' tools, while Section II-C presents the methodology for running the competition. The results are detailed in Section II-D, and Section II-E concludes the report with remarks and ideas for future improvements.

A. The benchmark subjects of the JUnit Testing Competition

The extraction of the classes under test (CUT) to use as benchmark for unit test case generators has been conducted considering several factors: (i) inclusion of different application domain [3], (ii) replicability, so open source projects may be preferable, and (iii) no trivial classes (e.g., classes without branches) [6]. We focused on GitHub projects (i) relying on Maven or Gradle as build framework, and (ii) including JUnit4 test suites. We used three out of four projects from the eight edition [5], by using a recent version of them, and we added three new ones. Specifically, we picked:

- *Guava (v29.0)* (https://github.com/google/guava), a set of Java libraries widely used within Google;
- *Seata (v1.3.0)* (https://github.com/seata/seata), a distributed transaction solution;
- *Spoon (v7.2.0)* (https://github.com/INRIA/spoon), a meta programming library to analyze and transform Java source code [7];

[1]https://github.com/JUnitContest/junitcontest

TABLE I: Characteristics of the benchmark.

Project	Cand.	Samp.
Guava	274	25
Seata	29	6
Spoon	179	15
FastJSON	120	20
Okio	35	7
Weka	1165	25
Total	1802	98

- *FastJSON (v1.2.66)* (https://github.com/alibaba/fastjson), a JSON parser and generator for Java;
- *Okio (v1.16.0)* (https://github.com/square/okio) a I/O library for Android, Kotlin and Java;
- *Weka (v3.8)* (https://github.com/Waikato/weka-3.8), i.e., a workbench for machine learning.

Based on the time and resources available for running out the competition, we have only sampled a limited number of CUTs using the approach adopted in the last edition [5]. Once having removed the classes in which the whole set of methods have a McCabe's cyclomatic complexity lower than five, we randomly sampled 98 classes from the six projects proportionally to their overall number of testable classes. During the random process, once identified a class, we ran Randoop with a time budget of 10 seconds to exclude classes where Randoop cannot provide any test case.

Table I reports, for each project, the total number of candidate CUTs together with the number of classes used as benchmark.

B. JUnit Testing Competition Tools

Five tools are competing in this edition: Randoop [1], UtBot, Kex [2], Evosuite [3] and EvosuiteDSE [4].

Randoop generates unit tests using a feedback-directed random test generation, collecting information from the execution of the tests as they are generated to reduce the number of redundant and illegal tests [1]. Differently, **UtBot**, a tool implemented by Huawei (Russian Research Institute, Saint Petersburg Research Center, Software Analysis Team), relies on symbolic execution extracting 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 (Satisfiability Modulo Theory) 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. Similarly, **Kex** [2] works as a symbolic execution engine and uses SMT solvers to perform the constraints solving. 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 constructs valid test cases from generated input parameters. **EvosuiteDSE** [4] uses a pure dynamic symbolic execution approach along with a generational search exploration strategy in an attempt to maximize branch coverage. It also uses Evosuite's previously developed test suite post-processing techniques (e.g., test suite minimization). Finally, **Evosuite** [3] uses an evolutionary algorithm to evolve a set of unit tests satisfying a given set of test objectives.

C. Methodology of the JUnit Testing Competition

The methodology followed to run the competition is similar to the one adopted in the eight edition [5]. Due to time constraints and amount of resources required to compare the various tools, we decided to focus on two different time budgets (i.e., 30 and 120 seconds), as well as, on a limited set of classes in the case of more computational demanding tools.

Public contest repository. The complete contest infrastructure is released under a GPL-3.0 license and is available on Github [8]. The repository also contains the benchmark (i.e., CUTs), detailed reports and data for the ninth edition, as well as, for previous ones.

Execution environment. The infrastructure performed a total of 6,250 executions (2,800 in the previous edition). In principle, without considering the resources needed for the execution of each competitors, we planned to have 98 CUTs x 6 tools x 2 time budgets x 10 repetitions, resulting in 11,760 executions in total to use for statistical analysis. However, only for Randoop, Evosuite and EvosuiteDSE we were able to run the planned number of executions. As regards the other three competitors' tools we realized that, (i) Kex was not able to produce any test case for four out of six projects in our benchmark, so we only have results for 30 CUTs belonging to Guava and Seata; finally (ii) Utbot requires too much memory and disk space so we executed it only on 50% of the total number of CUTs in our benchmark. The executions were run in parallel using Docker on four servers with similar characteristics: Linux Flavor with 8 CPU cores, 16 GB of RAM and 240 GB memory.

Test generation and time budget. Once accounted for each tool constraints, we executed each tool ten time against each CUT for each time budget in order to reduce the randomness of the generation processes [9]. Furthermore, we considered two different time budgets, i.e., 30 and 120 seconds.

Metrics computation. As for last year [5], the time budget used for mutation analysis has been set to five minutes for each class under test, while the timeout considered for each mutant has been set to one minute. The mutants has been sampled among the ones generated by PITest [10] using the following procedure: for CUTs with more than 200 mutants we randomly kept only 33% of them, while for CUTs with more than 400 mutants we sampled 50% of them for our analysis. Finally, for coverage metrics, we focused on lines and branches coverage by relying on JaCoCo [11].

Statistical analysis. Similarly to previous editions [5], we used statistical tests to support the results. Specifically, we use the Friedman test for assessing whether the scores over the different CUTs and time budgets achieved by the competitors tools are significantly different from each other; then we use

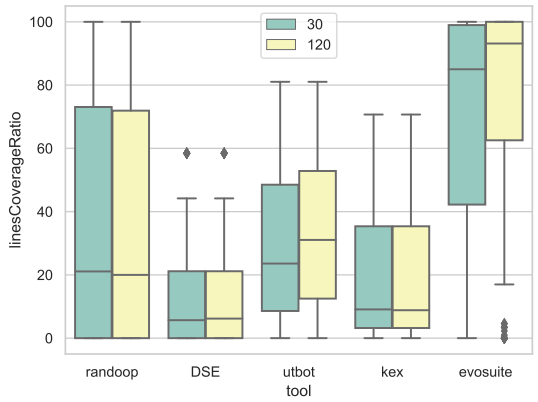

Fig. 1: Line Coverage for Randoop, Evosuite(DSE), Utbot, Kex and Evosuite for 30 and 120 seconds.

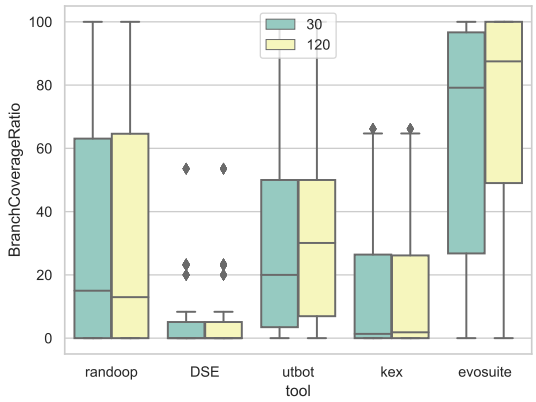

Fig. 2: Branch Coverage for Randoop, Evosuite(DSE), Utbot, Kex and Evosuite for 30 and 120 seconds.

the post-hoc Conover's test to determine for which pair of tools the significance actually holds, once having adjusted them with the Holm-Bonferroni procedure.

D. Results of the JUnit Testing Competition

Table II presents, for each tool and for each time budget the minimum, mean, median and maximum number of test cases being generated. As expected, in almost all the cases, while increasing the time budget given for generation purposes, the number of test cases being produced increase. Furthermore, for 19 CUTs in our benchmark, at least one tool was not able to generate any test case.

TABLE II: Statistics on number of test cases generation for each tool and each time budget.

Tool	Time budget	Min	Mean	Median	Max
Randoop	30	0	1502	754	9428
	120	0	2801	1429	18115
Evosuite	30	0	70	65	229
	120	0	56	49	313
EvosuiteDSE	30	0	5	2	29
	120	0	16	2	356
Utbot	30	0	341	461	595
	120	0	4589	6575	6729
Kex	30	0	28	14	150
	120	0	30	19	150

Going deeper on the evaluation of the test cases being generated by each tool, we observed cases for which our infrastructure was not able to compute metrics in terms of both coverage and mutant analysis. By removing those cases from our analysis, Figures 1, 2 and 3 show the ratio of lines, branches and mutants being covered by Randoop, Evosuite, EvosuiteDSE, Utbot and Kex, for each specific time budget. Note that the mutation coverage is the ratio between the mutants that were killed by at least one test and the total number of mutants. Unsurprisingly, while increasing the time budget, also the median line, branch and mutant coverage

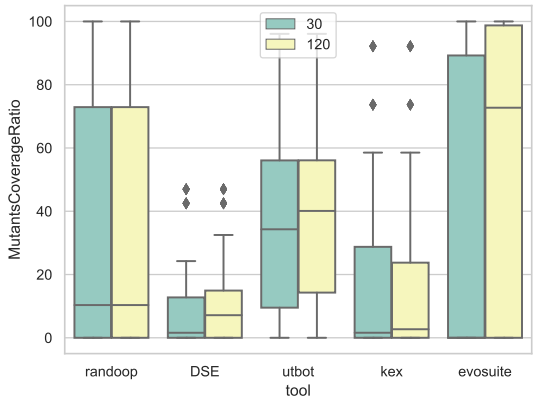

Fig. 3: Mutant Coverage for Randoop, Evosuite(DSE), Utbot, Kex and Evosuite for 30 and 120 seconds.

(slightly) increase. This is more evident for Utbot and Evosuite (see Figure 3). Specifically, for Utbot the number of mutants being killed over the total number of mutants moves from 34.3% (30 seconds) up to 40.1% with a time budget of 120 seconds, while for Evosuite increases from 0% up to 72.7%. Moreover, Evosuite achieves, on average, a higher coverage and mutation score for all the projects followed by Utbot. Specifically, for a time budget of 120 seconds, Evosuite and Utbot achieve (i) a line coverage of 93.2% and 31% compared to 20% of Randoop, 8% of Kex and 6.2% of EvosuiteDSE; (ii) a branch coverage of 87.5% and 30.1% compared to 13%, 1% and 0% of Randoop, Kex and EvosuiteDSE; and (iii) a mutation score of 72.7% and 40.1% compared to 8%, 7.1% and 2.7% obtained while using Randoop, EvosuiteDSE and Kex. Finally, while Evosuite is the tool that performs best, EvosuiteDSE is the worst on the CUTs selected for this ninth edition of the competition. It is important to highlight that these results may be influenced by the specific project versions and classes under tests considered this year, as well as, the

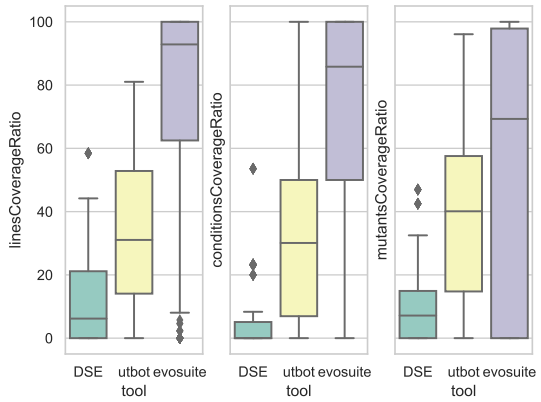

Fig. 4: Coverage for Evosuite(DSE), Utbot and Evosuite on a time budget of 5 minutes.

usage of 30 and 120 seconds as time budgets.

Due to resources and time limitations, we were able to generate test cases by using a time budget of five minutes only for three out of six tools, namely EvosuiteDSE, Utbot and Evosuite. Figure 4 shows the comparison between them while looking at line, branch and mutant coverage ratio. The graph clearly shows a trend between those three tools with EvosuiteDSE performing the worst and Evosuite performing the best. Specifically, the median values for line, branch and mutant coverage are doubled while comparing Evosuite with Utbot. For instance, EvosuiteDSE has a median line coverage ratio of 6.2% compared to 31.1% of Utbot and 92.9% of Evosuite.

For next year we plan to investigate this further, by including changes in the infrastructure that allow parallel executions, thus allowing the execution of different and greater time budgets (e.g., 5 and 10 minutes), for a more complete comparison of the various tools.

Finally, we report the final score and ranking achieved by the tools at different search budgets as well as the ranking produced by the Friedman test. Specifically, we observed a score of 292.05 for Evosuite, 121.04 for Randoop, 87.76 for Utbot, 47.14 for EvosuiteDSE and 44.21 for Kex. In terms of ranking, instead, we have Evosuite (1.43), Randoop (2.56), EvosuiteDSE (2.56), Utbot (3.53), and Kex (4.15). It is important to mention that in doing the comparison we limited the results to the ones available for the six competitors' tools.

E. Conclusions and Final Remarks of the JUnit Testing Tool Competition

This year was the ninth edition of the Java Unit Testing Competition. Compared to previous editions, this year we have four competitors, namely EvosuiteDSE, Kex, Utbot, and Evosuite, and Randoop as a baseline, among which the best performing one is Evosuite followed by Utbot, while EvosuiteDSE seem to perform the worst on the CUTs selected for this year.

The dockerized version of the infrastructure allowed us to distribute the execution on four different servers. However, most of the tool being submitted required too much RAM for generation (around 13GB for Kex) as well as a huge amount of disk space for storing the results. As an example, Kex and Utbot required 140-150GB for storing generated test cases and intermediate results while executing on the whole set of CUTs in our benchmark. The above constraints limited us in properly parallelizing the execution in terms of automatic generation of test cases.

The two-steps procedure used to select the different CUTs proved to be useful again this year. It allowed us to discover configuration issues in the competition infrastructure (e.g., wrong class-paths) and avoid several of the difficulties encountered last year.

As future directions we envision several possibilities: we need to (i) properly verify the reasons why for some CUTs in our benchmark, our infrastructure was not able to produce both coverage and mutation analysis data; (ii) include additional criteria than the coverage and mutation analysis (e.g., performances [12] and readability [13]) for evaluation purposes; (iii) experiment with tools supporting the testing of more complex application (e.g., cloud-based systems [14]); and (iv) consider to extend the infrastructure to support other languages (e.g., Python [15]).

III. The Cyber-Physical Systems Testing Tool Competition

Self-driving cars are a particular family of Cyber-Physical Systems (CPSs), quickly becoming part of our everyday lives. Since those systems are safety-critical, they should be thoroughly tested to avoid deadly consequences [16], [17]. Therefore, in addition to the Java unit testing tool competition, we organized the first CPS testing tool competition this year. This novel competition aims to shed light on the challenges of CPS testing and promote open research.

This first edition received a remarkable number of submissions, despite the short time to prepare them. Namely, this year's participants are Deeper [18], Frenetic [19], Swat [20], and GAB- [21], which was submitted in two orthogonal configurations, i.e., exploratory (GABExplore) and exploitative (GABExploit).

We executed all the tools in the BeamNG.tech driving simulator [22] against the same test subject. We studied their transitory behavior by giving each tool a two-hour generation budget and configuring the test subject to drive up to 70 Km/h. Additionally, we used a five-hour generation budget to study their asymptotic behavior and removed the speed limit.

To carry out the evaluation, we developed a brand new testing infrastructure by exploiting the simulator's official library [23] and a previous research tool [24]. In particular, our infrastructure (i) allows testers to integrate their tools easily; (ii) visualizes and validates the generated tests; (iii) automatically executes the tests using the physically accurate driving simulator BeamNG.tech [22], and (iv) generates concise reports about test generation and execution.

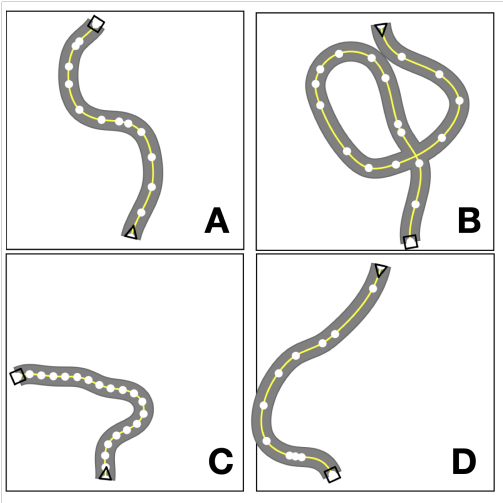

Fig. 5: Examples of valid (A) and invalid (B, C, D) virtual roads. Road B self-intersects, road C contains overly sharp turns, while road D goes outside the map boundaries.

Additionally, we developed the analysis scripts to compare the test generation tools. The infrastructure is open-source and available on GitHub at:

https://github.com/se2p/tool-competition-av.

In the remainder of the CPS testing competition report, we present the benchmark (Section III-A) and the participating tools (Section III-B). Then, we describe the adopted methodology (Section III-C), the experimental procedure (Section III-D) and report the evaluation results (Section III-E). Finally, we conclude with remarks and ideas for future improvements (Section III-F).

A. Simulation-based Testing of Self-Driving Car Software

CPSs range from medical devices to drones to the Internet of Things; hence, they form a broad domain of investigation. To keep our competition focused, we considered only one representative instance of CPSs, i.e., self-driving cars, which are increasingly becoming relevant to both academia [25] and industry [26].

Testing the software that controls self-driving cars is challenging because it requires creating relevant abstract testing scenarios and reifying them into concrete executions [27]. Those executions can take place in the real world or inside computer simulations. Naturalistic field operational tests (N-FOT) leave autonomous vehicles free to roam in the real world; hence, they are dangerous and expensive [28]. In contrast, virtual tests use computer simulations to assess both the hardware and the system's software under test. Virtual tests are faster, cheaper, and less dangerous than N-FOT and enable complete control over the test execution and programmatic test generation. Therefore, for this competition, we executed virtual tests in a driving simulator.

Specifically, we consider driving scenarios on single, flat roads surrounded by plain green grass. We fixed both the environmental conditions (i.e., weather and lighting) and the road layout (i.e., number and width of lanes) for simplicity. Consequently, driving simulations take place on a sunny day and on roads that consist of two fixed-width lanes demarcated by white solid lines and divided by a solid yellow line. This setup follows previous research on testing lane-keeping assist systems (LKAS). It also naturally defines the *driving task* that the ego-car must complete: driving without going off the lane from a *starting* position, i.e., the beginning of a virtual road, to a *target* location, i.e., the end of that virtual road.

The goal of the test generators is to create virtual roads that expose problems in the test subject, such as driving off-road, invading the opposite lane, or stopping in the middle of the road. Our framework represents virtual roads as sequences of points, i.e., *road points*, defined on a two-dimensional map with predefined size and shape. We interpolate the sequence of road points using cubic splines to obtain a smooth *road spine*, i.e., the road's central line, and adopt the convention that the first and last road points define the starting and target locations respectively. We exemplify virtual roads in Figure 5 where white dots represent the road points, triangles and squares represent the starting and target locations, and a yellow line represents the interpolated road spine.

The problem of generating virtual roads is challenging because there is a large number of possible road point sequences, but not all of them result in valid roads [29]. For example, roads might self-intersect or self-overlap (Figure 5-B) or contain turns that are too sharp to be driven without invading the opposite lane (Figure 5-C). Finally, roads must be fully contained in the maps to comply with the underlying driving simulation. Notably, this condition does not automatically hold even if all the road points lie inside the map because we interpolate the road points using cubic splines (Figure 5-D). We validate the roads generated by the tools and report invalid ones without executing them. Therefore, invalid roads do not count as a failed tests. As we discuss in Section III-C, accounting for them, let us make interesting observations on tools' generation effectiveness.

To ensure a uniform representation of inputs, an automatic test execution, and an easy integration of the tools, we designed an extensible testing framework that dynamically loads the test generators as plugin-ins. The framework ensures that only valid virtual tests are executed and procedurally generates driving simulations from them. During the test execution, it monitors the operation of the tools and the driving simulations to enforce the generation budget and collect the data required by the tools to work (e.g., to measure tests' goodness using fitness functions).

B. The Tools of the CPS Testing Competition

Four tools competed in this first edition of the CPS testing tool competition. We summarize below their key features.

Deeper [18] uses a multi-objective search algorithm (i.e., NSGAII [30]) to maximize the distance between the car and the center of the lane while minimizing the road length. It uses Catmull-Rom splines to represent inputs and relies on the

search operators implemented by DeepJanus [31] to evolve a predefined initial population.

Frenetic [19] uses a single-objective genetic approach to maximize the distance between the ego-car and the center of the lane. It uses Frenet frames [32] to represent inputs and, in addition to standard genetic operators, introduces a mechanism to promote the diversity of the generated roads.

GA-Bézier [21] uses a genetic algorithm to evolve roads represented using Bézier curves [32]. To address the trade-off between exploring the test space and exploiting observations collected during the test generation, its authors implemented two versions of the test generator: GABExploit, to maximize the number of failing test cases, and GABExplore, to generate roads with unique features. The authors describe these approaches and their ongoing work in a preliminary paper [33].

Swat [20] uses Markov chains to generate random sequences of road segments, including straight segments and turns. Inspired by AsFault [34], it utilizes affine transformations to generate virtual roads from those segments. Differently from AsFault, Swat does not employ evolutionary search algorithms. According to its authors, optimizing the test cases generated using the Markov chain using evolutionary search algorithms is part of future work [35].

C. The Contest methodology of the CPS Testing Competition

1) Subject System of the CPS Testing Competition: We evaluated the competing tools using the BeamNG.tech driving simulator [22], a freely available research-oriented version of the commercial game BeamNG.drive and chose as test subject its built-in driving agent, BeamNG.AI. This driving agent is *omniscient*, i.e., it 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 inside the lane. We decided to use BeamNG.AI as a test subject for the following reasons: 1) it has been used in previous research to assess test input generators [29], [34] and train vision-based steering predictors [31], 2) it does not require manual training, which reduces the threats to the validity of the evaluation; and, 3) we can alter its driving style by changing parameters such as the maximum speed at which the ego-car can drive.

The competitors had access to the same agent and simulator we used for the final evaluation. Therefore, while preparing the submission, they could assess the performance of their tools under different execution conditions by configuring the test subject.

2) Goal and Metrics: The goal of the competition is to generate the highest number of diverse failure-inducing inputs, i.e., valid roads that cause the ego-car to drive out of the lane within the given time budget. In the following, we report the considerations we made in evaluating the competing tools.

Detected Failures. Effective test generators trigger many failures; therefore, we count for each tool how many generated tests fail. Our infrastructure detects a failure each time the ego-car partially drives outside the lane 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 Episodes (OBEs) following the naming convention defined by Gambi et al. [29] and refer to the threshold value controlling them as *tolerance*.

TABLE III: Experimental Setups

Name	Map Size (m^2)	Max Speed (Km/h)	Budget (h)	Tolerance (%)
DEFAULT	200×200	–	5	0.95
SBST21	200×200	70	2	0.85

Failure Diversity. Tests are useful when they trigger diverse failures; otherwise, they would waste computational resources in exposing the same issues multiple times. To measure failure diversity, we adopt a two-step strategy: first, we extract the road segments relevant to the failures; then, we compute their sparseness.

We define the road segment relevant to a failure as the part of the road *around* the OBE location. We consider the road segment *before* the OBE to account for the most recent activity of the ego-car (e.g., acceleration, steering), and the road segment *after* the OBE to account for the trajectory that the driving agent has planned (e.g., based on the camera field-of-view). Specifically, we consider relevant segments 60m-long (30m before the failure and 30m after it) to keep them as short as possible, hence easy to compare, but informative. We compute failures' sparseness as the average maximum distance of the road sectors relevant to a set of OBEs (S_{OBE}), according to the following formula:

$$sparseness = \frac{\sum_{i \in S_{OBE}} \max_{j \in S_{OBE}} dist(i,j)}{|S_{OBE}|} \quad (1)$$

where $dist(i,j)$ is the weighted Levenshtein distance [36] between road segments suggested by Riccio and Tonella [31].

Generation Efficiency and Effectiveness Efficient test generators generate a large number of tests within the generation budget. In contrast, effective test generators wisely use the generation budget; e.g., they avoid generating invalid tests. Therefore, we count the total number of generated tests and the number of valid and invalid tests to evaluate the performance of the competing tools.

D. Experimental Procedure

We ran the tools on the subject system in two different experimental setups: DEFAULT and SBST21 (see Table III). DEFAULT is the original setup we provided to the competitors. It uses a five-hour generation budget to study the asymptotic behavior of the tools. In this configuration, we set the tolerance to 0.95, making it hard to trigger test failures. We also impose no speed limit to the ego-car, which consequently adopts a more reckless driving style. SBST21 uses a shorter time budget (2 hours) and a lower tolerance value (0.85), which increases the OBE monitor's sensitivity. Additionally, this

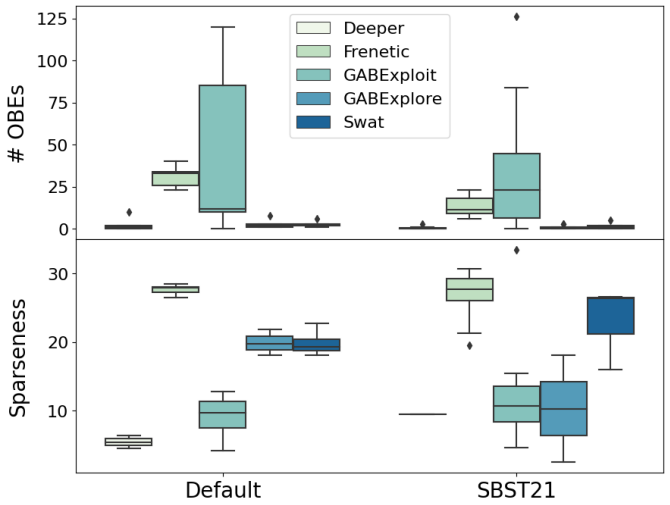

Fig. 6: Benchmark Results. The top plot reports the number of detected failures, while the bottom plot reports their sparseness.

configuration features a more prudent driving agent that can travel up to 70 Km/h.

To ensure a fair comparison, we ran each tool the same number of times and on the same dedicated machine. Specifically, we ran DEFAULT 5 times and SBST21 10 time 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.

E. Results of the CPS Testing Competition

Detected Failures Figure 6 (top) shows the distribution of number of OBEs triggered by each tool. Frenetic and GABExploit triggered at least 10x more OBEs than the other tools for both configurations. In particular, Frenetic has shown consistent performance across the runs, i.e., small standard deviation. In contrast, GABExploit had dramatically different behaviors across the runs returning over 120 OBEs and over 84 OBEs in some runs but failed to trigger any failure in others. Instead, the other tools triggered only a negligible number of OBEs (on avg less than 2 OBEs in the SBST21 configuration and less than 3 OBEs in the DEFAULT configuration). Furthermore, those tools did not expose failures in all the runs.

Failure Diversity Figure 6 (bottom) shows the distribution of failures' sparseness for each tool. Deeper has shown low sparseness in both configurations, probably due to the absence of mechanisms to promote road diversity. Swat achieved high sparseness despite finding a small number of OBEs. Both GABExploit and GABExplore achieved low sparseness within the 2 hour budget of the SBST21 configuration, but only GABExplore improved remarkably (2X) when given the larger generation budget of the DEFAULT configuration. We expected this improvement as GABExplore explicitly promotes road diversity. Frenetic achieved the highest failure diversity for both configurations. Notably, Frenetic found the highest number of OBEs for which the ego-car invades the opposite traffic lane. Finding this kind of OBEs has been difficult for the remaining tools since they did not find any of them in multiple runs.

TABLE IV: Test Generation Efficiency and Effectiveness

Tool	Default			SBST21		
	TCs	Val.	Inval.	TCs	Val.	Inval.
Deeper	433.0	**391.0**	42.0	169.7	**152.9**	16.8
Frenetic	**511.4**	369.4	142.0	**180.6**	134.2	46.4
GABExploit	393.4	357.8	35.6	146.3	135.6	10.7
GABExplore	381.2	339.6	41.6	124.7	115.9	8.8
Swat	405.6	388.8	**16.8**	150.5	144.1	**6.4**

Generation Efficiency and Effectiveness. Table IV reports the average number of test cases produced by each tool across all runs, as well as the average count of valid and invalid test cases. These results suggest that all the tools followed similar trends in both configurations. Frenetic generated the highest number of test cases within the time budget. However, this tool produced the highest number of invalid roads, mainly caused by overly sharp turns. Deeper also generated many test cases. It produced the highest number of valid tests and never generated self-intersecting roads nor roads outside the map. Swat shows the best ratio between valid and generated test cases. For both configurations, it produced over 95% valid test cases. GABExplore and GABExploit generated a remarkable 92% valid test cases; however, these tools also generated the lowest number of test cases.

F. Conclusions and Final Remarks of the CPS Testing Competition

This year we organized the first testing tool competition focusing on CPS and faced several challenges to make it happen. First, we faced the challenge of creating a testing infrastructure that easily integrates and compares the tools. All the research groups which took part in the competition successfully integrated their tools into our infrastructure and provided us with valuable feedback on how to improve it. Another major challenge was the absence of a well-established approach for systematically evaluating and comparing the test generators. In this edition, we used simple metrics already adopted in the literature. However, we believe that the definition of additional metrics to better evaluate CPS testing tools is a crucial problem to tackle for future editions of the competition.

All the five competing tools were able to generate inputs that triggered failures of the subject system. In particular, Frenetic and GABExploit have been very effective and triggered many failures. Nonetheless, both of them presented limitations: Frenetic generated the highest number of invalid tests among the competitors, while GABExploit could not always find OBEs, and the ones it found are not very diverse.

ACKNOWLEDGMENT

We thank all participants of the JUnit-test tool competition of this and previous years, which continuously sustain the evolution and maturity of automated testing strategies. We also thank all participants of the CPS testing tool competition and BeamNG.GmbH for freely providing their driving simulator. This work was partially supported by the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703). Sebastiano Panichella and Fiorella Zampetti gratefully acknowledges the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

REFERENCES

[1] 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 - OOPSLA '07*, vol. 2. ACM Press, 2007, p. 815.

[2] "Kex," https://github.com/vorpal-research/kex/tree/sbst-21, 2021, [Online; accessed 23-02-2021].

[3] G. Fraser and A. Arcuri, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, dec 2014.

[4] "EvoSuite DSE Module," https://github.com/ilebrero/evosuite/releases/tag/DSE.SBSTToolCompetition2021, 2021, [Online; accessed 23-02-2021].

[5] X. Devroey, S. Panichella, and A. Gambi, "Java unit testing tool competition: Eighth round," in *International Conference on Software Engineering Workshops*, 2020, pp. 545–548.

[6] A. Panichella, F. M. 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. 44, no. 2, pp. 122–158, 2018.

[7] F. Kifetew, X. Devroey, and U. Rueda, "Java Unit Testing Tool Competition - Seventh Round," in *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE, may 2019, pp. 15–20.

[8] "Contest Infrastructure," https://github.com/JUnitContest/junitcontest, 2021, [Online; accessed 23-02-2021].

[9] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verif. Reliab.*, vol. 24, no. 3, p. 219–250, May 2014. [Online]. Available: https://doi.org/10.1002/stvr.1486

[10] "PiTest," http://pitest.org/, 2021, [Online; accessed 23-02-2021].

[11] "JaCoCo," https://www.jacoco.org/jacoco/trunk/doc/, 2021, [Online; accessed 23-02-2021].

[12] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[13] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: an empirical investigation," in *International Conference on Software Engineering, ICSE 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 547–558. [Online]. Available: https://doi.org/10.1145/2884781.2884847

[14] D. Martin and S. Panichella, "The cloudification perspectives of search-based software testing," in *International Workshop on Search-Based Software Testing, SBST@ICSE 2019*, A. Gorla and J. M. Rojas, Eds. IEEE / ACM, 2019, pp. 5–6. [Online]. Available: https://doi.org/10.1109/SBST.2019.00009

[15] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020*, ser. Lecture Notes in Computer Science, A. Aleti and A. Panichella, Eds., vol. 12420. Springer, 2020, pp. 9–24. [Online]. Available: https://doi.org/10.1007/978-3-030-59762-7_2

[16] @elonbachman, "Tesla deaths," Feb. 2020, Regularly updated at tesladeaths.com; version hosted on Zenodo will be updated periodically. [Online]. Available: https://doi.org/10.5281/zenodo.3685309

[17] D. Wakabayashi, "Self-driving uber car kills pedestrian in arizona, where robots roam," March 2018. [Online]. Available: https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html

[18] M. H. Moghadam, M. Borg, and J. Mousavirad, "Deeper tool," https://github.com/mahshidhelali/tool-competition-av, 2021.

[19] E. Castellano, A. Cetinkaya, C. H. Thanh, S. Klikovits, X. Zhang, and P. Arcaini, "Frenetic tool," https://github.com/ERATOMMSD/frenetic-sbst21, 2021.

[20] D. Humeniuk, F. Khomh, and G. Antoniol, "Swat tool," https://github.com/dgumenyuk/tool-competition-av.git, 2021.

[21] L. Klampfl, F. Klück, and F. Wotawa, "Ga-bézier tool," The code of the tool is not openly avialable, 2021.

[22] BeamNG GmbH, "BeamNG.tech," 2021. [Online]. Available: https://www.beamng.gmbh/research

[23] ——, "BeamNGpy," 2021. [Online]. Available: https://github.com/BeamNG/BeamNGpy

[24] V. Riccio and P. Tonella, "Deepjanus tool," https://github.com/testingautomated-usi/DeepJanus, 2020.

[25] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5193–5254, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09881-0

[26] M. Borg, "The aiq meta-testbed: Pragmatically bridging academic ai testing and industrial q needs," in *Software Quality: Future Perspectives on Software Engineering Quality*, D. Winkler, S. Biffl, D. Mendez, M. Wimmer, and J. Bergsmann, Eds. Cham: Springer International Publishing, 2021, pp. 66–77.

[27] L. Li, W. Huang, Y. Liu, N. Zheng, and F. Wang, "Intelligence testing for autonomous vehicles: A new approach," *IEEE Trans. Intell. Veh.*, vol. 1, no. 2, pp. 158–166, 2016. [Online]. Available: https://doi.org/10.1109/TIV.2016.2608003

[28] N. Kalra and S. M. Paddock, *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation, 2016. [Online]. Available: http://www.jstor.org/stable/10.7249/j.ctt1btc0xw

[29] A. Gambi, M. Müller, and G. Fraser, "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*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 27–30. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00030

[30] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, April 2002.

[31] V. Riccio and P. Tonella, "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*, ser. ESEC/FSE '20. Association for Computing Machinery, 2020, p. 13 pages.

[32] D. Carroll, E. Hankins, E. Kose, and I. Sterling, "A survey of the differential geometry of discrete curves," *The Mathematical Intelligencer*, vol. 36, no. 4, pp. 28–35, 2014.

[33] F. Klück, L. Klampfl, and F. Wotawa, "Automatic generation of challenging road networks for alks testing based on bezier curves and search," 2021, arXiv:2103.01288 [cs.SE].

[34] A. Gambi, T. Huynh, and G. Fraser, "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*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 257–267. [Online]. Available: https://doi.org/10.1145/3338906.3338942

[35] D. Humeniuk, G. Antoniol, and F. Khomh, "Data driven testing of cyber physical systems," 2021.

[36] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.