Cost-effective Simulation-based Test Selection in Self-driving Cars Software with SDC-Scissor

Christian Birchler*, Nicolas Ganz*, Sajad Khatiri†, Alessio Gambi‡, Sebastiano Panichella*
*Zurich University of Applied Sciences, Switzerland
† Software Institute - USI Lugano and Zurich University of Applied Sciences, Switzerland
‡University of Passau, Germany

***Abstract*—Simulation platforms facilitate the continuous development of complex systems such as self-driving cars (SDCs). However, previous results on testing SDCs using simulations have shown that most of the automatically generated tests do not strongly contribute to establishing confidence in the quality and reliability of the SDC. Therefore, those tests can be characterized as *"uninformative"*, and running them generally means wasting precious computational resources. We address this issue with SDC-Scissor, a framework that leverages Machine Learning to identify simulation-based tests that are unlikely to detect faults in the SDC software under test and skip them before their execution. Consequently, by filtering out those tests, SDC-Scissor reduces the number of long-running simulations to execute and drastically increases the cost-effectiveness of simulation-based testing of SDCs software. Our evaluation concerning two large datasets and around 12'000 tests showed that SDC-Scissor achieved a higher classification F1-score (between 47% and 90%) than a randomized baseline in identifying tests that lead to a fault and reduced the time spent running uninformative tests (speedup between 107% and 170%).**
Webpage & Video: *https://github.com/ChristianBirchler/sdc-scissor*
***Index Terms*—Self-driving cars, Software Simulation, Regression Testing, Test Case Selection, Continuous Integration**

I. Introduction

Cyber-physical systems (CPSs) are complex systems that leverage physical capabilities from hardware components [3] and are expected to drastically improve the quality of life of citizens and the economy [10]. CPSs find applications in various domains including Robotics, Transportation and Healthcare. For instance, in the transportation domain, they take the form of self-driving cars (SDCs) and are expected to impact our society profoundly. Indeed, human errors cause more than 90% of driving accidents [17] and automated driving systems have the potential to reduce such errors and eliminate most accidents. However, the recent fatal crashes involving SDCs suggest that the advertised large-scale adoption of SDCs appears optimistic [3], [27].

Reducing the risk of releasing SDCs equipped with defective software that might become erratic and lead to fatal crashes is one of the main challenges concerning the usage of autonomous vehicles. We argue that enabling cost-effective testing automation in Continuous Integration (CI) pipelines for SDCs is a paramount challenge to address for ensuring the safety and reliability of SDCs [4], [17], [20]. However, current SDC testing practices have several limitations: (i) difficulty in testing SDCs using representative, safety-critical tests [16], [27]; (ii) difficulty in assessing SDC's behavior in different environmental conditions [17].

The usage of simulation environments can potentially address several of the aforementioned challenges [7], [11], [22] because they enable more efficient test execution, reproducible results, and testing under critical conditions [12]. Additionally, simulation-based testing can be as effective as traditional field operational testing [2], [11]. However, the testing space of simulation environments is infinite, which poses the challenge of exercising the SDC behaviors adequately [1], [13]. Given the limited budget devoted to testing activities, it is paramount that developers test SDCs in a cost-effective fashion: using test suites optimized to reduce testing effort (time) without affecting their ability to identify faults [1], [21], [28].

To increase SDC testing cost-effectiveness, we propose **SDC-Scissor** (**SDC** co**S**t-effe**C**t**I**ve te**S**t **S**elect**OR**), a framework that leverages Machine Learning (ML) to enable cost-effective simulation-based testing via test selection. SDC-Scissor identifies (i.e., predicts) tests that are unlikely to detect faults and skips them before their execution; hence, it reduces the time spent in executing such tests. Specifically, we refer to tests that do not expose a fault as *safe* and deem them irrelevant. On the contrary, we consider tests that expose a fault (e.g., an SDC drives out of the road) as relevant and refer to them as *unsafe*. SDC-Scissor exploits ML models trained on features of SDC simulation-based tests that can be computed before the actual test execution (i.e., input features) to classify whether SDC tests are safe or unsafe. We briefly discuss input features in Section II-C and remand to our previous works on test selection [19] and test prioritization [6] for more detailed information about SDC test features.

Through a large study on two datasets containing over 12'000 SDC simulation-based tests, we assessed the performance of SDC-Scissor in optimizing simulation-based testing. Our evaluation shows that SDC-Scissor accurately identified more unsafe tests and reduced the time spent in running safe tests than a random baseline.

II. The SDC-Scissor Tool

In this section, we overview SDC-Scissor's software architecture and its main usage scenarios (Fig. 1); we describe the simulation environment it uses (i.e., BeamNG.tech); and,

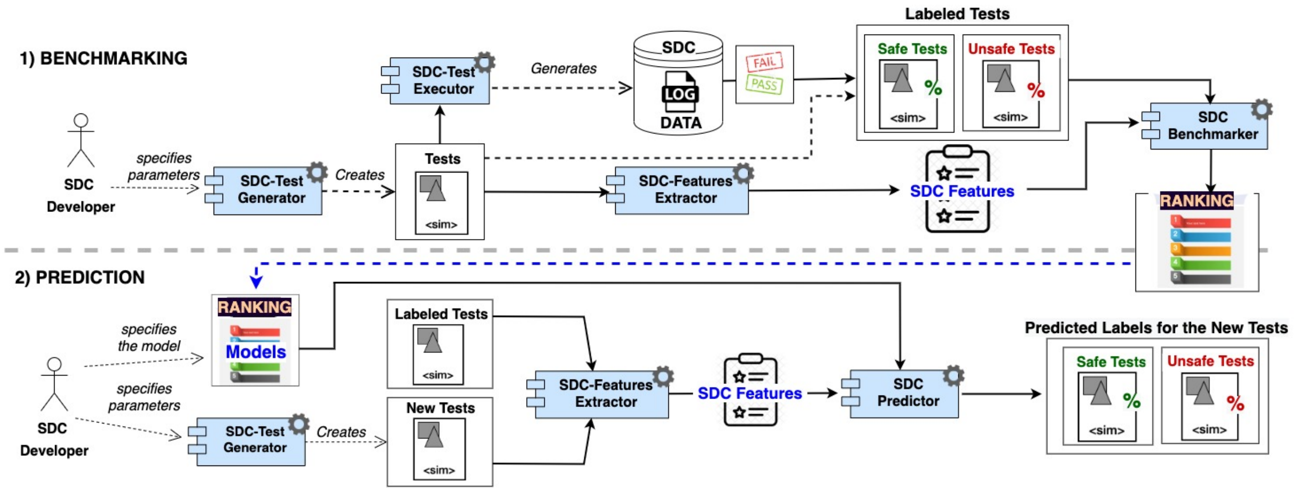

Fig. 1. The SDC-Scissor's architecture.

finally, we discuss in detail the components, the approach and the technologies behind SDC-Scissor.

A. SDC-Scissor Architecture Overview & Main Scenarios

SDC-Scissor supports two main usage scenarios: *Benchmarking* and *Prediction*. In the *Benchmarking* scenario, developers leverage SDC-Scissor to determine the best ML model(s) to classify SDC simulation-based tests as safe or unsafe. In the *Prediction* scenario, instead, developers use those model(s) to classify and select newly generated test cases.

SDC-Scissor Software Architecture implements these scenarios by means of five main software components: (i) `SDC-Test Generator` generates random SDC simulation-based tests, and (ii) `SDC-Test Executor` executes them. The test results produced by `SDC-Test Executor` are recorded and used to label tests as safe or unsafe; (iii) `SDC-Features Extractor` extracts input features of the executed SDC tests, while (iv) `SDC-Benchmarker` uses these features and corresponding labels as input to train the ML models and determine which model best predicts the tests that are more likely to detect faults in SDCs; finally, (v) `SDC-Predictor` uses the ML models to classify newly generated test cases and enables test selection.

B. BeamNG.tech's Simulation Environment

SDC-Scissor uses BeamNG.tech to execute SDC tests as physically accurate and photo-realistic driving simulations. BeamNG.tech can procedurally generate tests [13] and was recently adopted in the ninth edition of the Search-Based Software Testing (SBST) tool competition [23].

BeamNG.tech is organized around a central *game engine* that communicates with the *physics simulation*, the *UI*, and the *BeamNGpy API*[1]. The UI can be used for game control and manual content creation (e.g., *assets*, *scenarios*). For example, developers can use the world editor to create or modify the virtual environments that are used in the simulations; testers, instead, can create test scripts implementing driving scenarios (i.e., the tests). The API, instead, allows the automated generation and execution of tests, the collection of simulation data (e.g., camera images, LIDAR point clouds) for training, testing, and validating SDCs. It also enables driving agents to drive simulated vehicles and get programmatic control over running simulations (e.g., pause/resume simulations, move objects around). The *game engine* manages the simulation setup, camera, graphics, sounds, gameplay, and overall resource management. The *physics core*, instead, handles resource-intensive tasks such as collision detection and basic physics simulation; it also orchestrates the concurrent execution of the vehicle simulators. The *vehicle simulators* —one for each of the simulated vehicles— simulate the high-level driving functions and the vehicle sub-systems (e.g., drivetrain, ABS).

We employ the BeamNG.AI[2] lane-keeping system as the test subject for our evaluation: the driving agent is shipped with BeamNG.tech and drives the car by computing an ideal driving trajectory to stay in the center of the lane while driving within a configurable speed limit. As explained by BeamNG.tech developers, the *risk factor* (RF) is a parameter that controls the driving style of BeamNG.AI: low-risk values (e.g., 0.7) result in smooth driving, whereas high-risk values (e.g., 1.7 and above) result in an edgy driving that may lead the ego-car to cut corners [19].

C. The SDC-Scissor's Approach and Technology Overview

SDC-Scissor integrates the extensible testing pipeline defined by the SBST tool competition[3] in its `SDC-Test Executor`. We use the SBST tool competition infrastructure since it allows to (i) seamlessly execute the tests in BeamNG.tech and (ii) distinguish between *safe* and *unsafe* tests based on whether the self-driving car keeps its lane (non-faulty tests) or depart from it (faulty tests) [13]. Consequently, SDC-Scissor can accommodate various `SDC-Test Generators` for generating SDC simulation-based tests. In

[1] `beamngpy` is available on PyPI and Github (https://github.com/BeamNG/BeamNGpy)

[2] https://wiki.beamng.com/Enabling_AI_Controlled_Vehicles#AI_Modes

[3] https://github.com/se2p/tool-competition-av

TABLE I
FULL ROAD ATTRIBUTES EXTRACTED BY THE *SDC-Features Extractor*

Feature	Description	Range
Direct Distance	Euclidean dist. between start and end (m)	[0 – 490]
Length	Tot. length of the driving path (m)	[50.6 – 3,317]
Num L Turns	Nr. of left turns on the driving path	[0 – 18]
Num R Turns	Nr. of right turns on the driving path	[0 – 17]
Num Straight	Nr. of straight segments on the driving path	[0 – 11]
Total Angle	Cumulative turn angle on the driving path	[105 – 6,420]

TABLE II
FULL ROAD STATISTICS EXTRACTED BY THE *SDC-Features Extractor*

Feature	Description	Range
Median Angle	Median turn angle on the driving path (DP)	[30 – 330]
Std Angle	Std. Dev of turn angles on the DP	[0 – 150]
Max Angle	Max. turn angle on the DP	[60 – 345]
Min Angle	Min. turn angle on the DP	[15 – 285]
Mean Angle	Average turn angle on the DP	[52.5 – 307.5]
Median Radius	Median turn radius on the DP	[7 – 47]
Std Radius	Std. Dev of turn radius on the DP	[0 – 22.5]
Max Radius	Max. turn radius on the DP	[7 – 47]
Min Radius	Min. turn radius on the DP	[2 – 47]
Mean Radius	Average turn radius on the DP	[5.3 – 47]

this paper, we demonstrate SDC-Scissor by using the Frenetic test generation [9], one of the most effective tool submitted to the SBST tool competition.

SDC-Scissor predicts whether the tests are likely to be safe or unsafe before their execution using input features extracted by `SDC-Features Extractor`. Specifically, this component extracts *Full Road Features* (FRFs), i.e., a set of SDC features that describe global characteristics of the tests. Those features include the main *road attributes* (see Table I) and *road statistics* concerning the road composition (see Table II). Road statistics are calculated in three steps: (i) extraction of the *reference driving path* that the ego-car has to follow during the test execution (e.g., the road segments that the car needs to traverse to reach the target position); (ii) extraction of metrics available for each road segment (e.g., length of road segments); and (iii) computation of standard aggregation functions on the collected road segments metrics (e.g., minimum and maximum).

SDC-Scissor relies on the `SDC-Benchmarker` to determine the ML model that best classifies the SDC tests that are likely to detect faults. It follows an empirical approach to do so: given a set of labeled tests and corresponding input features, `SDC-Benchmarker` trains and evaluates an ensemble of standard ML models using the well-established `sklearn`[4] library. Next, it assesses ML models' quality using either 10-fold cross-validation or a testing dataset; and, finally, selects the best performing ML models according to Precision, Recall, and F1-score metrics [19]. Noticeably, SDC-Scissor can use many different ML models; however, in this work, we consider only Naive Bayes [8], Logistic Regression [25], and Random Forests [15]. We do so because these ML models have been successfully used for defect prediction or other classification problems in Software Engineering [5], [18], [24], [26].

Finally, the `SDC-Predictor` uses the ML models to predict the likelihood that newly generated SDC tests are safe or not. Specifically, developers have the possibility to select the ML models recommended by the `SDC-Benchmarker` (considered most accurate), or they can select other models of their choice.

[4]https://scikit-learn.org/

III. USING SDC-SCISSOR

SDC-Scissor tool is openly available and can be used as a Python command-line utility via `poetry`[5] as follows:

```
poetry install
poetry run python sdc-scissor.py [COMMAND] [OPTIONS]
```

To simplify SDC-Scissor's usage, we also enable to execute it as a Docker[6] container:

```
docker build --tag sdc-scissor .
docker run --volume "$(pwd)/results:/out" --rm
    sdc-scissor [COMMAND] [OPTIONS]
```

As we detail below, SDC-Scissor's command-line supports the execution of the main usage scenarios described in Section II-B by taking appropriate commands and inputs (see Fig. 2).

Test generation. To generate SDC tests by running the Frenetic generator within a given time budget (e.g., 100 seconds) SDC-Scissor requires the following command:

```
generate-tests --out-path /path/to/store/tests
    --time-budget 100
```

Automated test labeling. SDC-Scissor labels tests as safe and unsafe by executing them in BeamNG.tech. Since BeamNG.tech cannot be run as a Docker container, labelling tests can be only run locally (i.e., outside Docker). This labeling facility allows developers to create datasets that can be used for the training and validation of ML models (e.g., ML-based prediction of unsafe tests). Generating a labeled dataset, requires a set of already generated SDC tests and the execution of the following command:

```
label-tests --road-scenarios /path/to/tests
    --result-folder /path/to/store/labeled/tests
```

ML models evaluation. For identifying the models that SDC-Scissor could use for the prediction, SDC-Scissor implements a 10-fold cross-validation strategy on the input labeled dataset. The following command tells SDC-Scissor to benchmark all the configured ML models:

```
evaluate-models --tests /path/to/train/set --save
```

Note: the optional `save` flag forces SDC-Scissor to store the ML models' metadata for later inspection and usage.

[5]https://python-poetry.org/
[6]https://www.docker.com

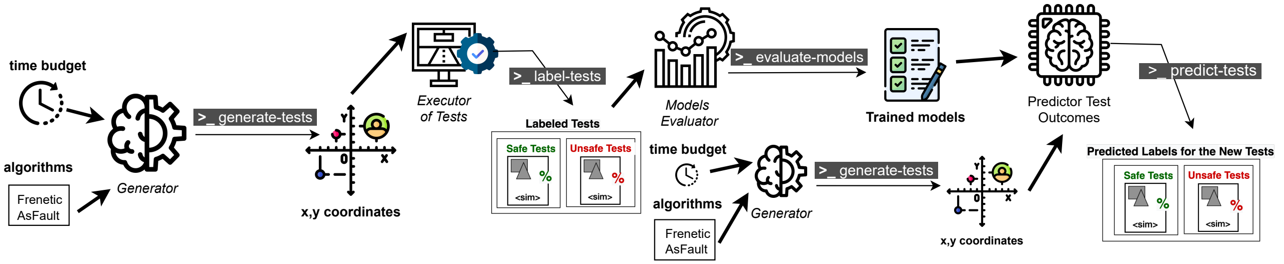

Fig. 2. The SDC-Scissor's fine-grained view.

Train and test data generation. Evaluating the prediction ability of SDC-Scissor requires separate training and testing datasets. The following command lets developers to split the available tests to achieve an 80/20 split:

```
split-train-test-data --tests /path/to/tests
    --train-dir /path/for/train/data
    --test-dir /path/for/test/data
    --train-ratio 0.8
```

Test outcome prediction. SDC-Scissor classifies unlabeled tests, i.e., it predicts their outcome, using a trained ML model with the following command:

```
predict-tests --tests /path/to/tests
    --predicted-tests /path/for/predicted/tests
    --classifier /path/to/model
```

Random baseline evaluation. SDC-Scissor allows to select tests using a random strategy that provides a baseline evaluation with the following command:

```
evaluate-cost-effectiveness
    --tests /path/to/tests
```

Prediction performance. SDC-Scissor allows to asses the performance of a classifier with the following command:

```
evaluate --tests /path/to/tests
    --classifier /path/to/model
```

IV. Evaluation

We evaluated SDC-Scissor conducting a large study on two datasets, referred as *Dataset 1* and *Dataset 2*, that contain over $12,000$ SDC tests (see Table III). We adopted the following experimental setup to obtain comprehensive and unbiased training datasets. For *Dataset 1*, we *randomly* generated $3,559$ valid tests using Frenetic [9], collected input features and executed them to collect labels. For the *Dataset 2*, instead, we generated $8,545$ tests using AsFault [13].

It is important to note that in executing all those tests, we experimented with different BeamNG.AI's risk factor as it influences the ego-car driving style. Specifically, we considered three configurations: cautious (RF 1.0), moderate (RF 1.5), and reckless (RF 2.0) driver. Using different values for the risk factor enabled us to study the effectiveness of SDC-Scissor on various SDCs' driving styles. We empirically validated our expectations by running the moderate driver using *Dataset 1* tests and running all the three configurations for *Dataset 2* tests. From Table III we can observe that the number of unsafe tests increased with increasing values of BeamNG.AI's risk factor. Hence, this result confirms that the risk factor indeed strongly influences the safety of BeamNG.AI and the outcome of tests.

To assess the performance of SDC-Scissor in optimizing simulation-based SDCs testing via test selection (i.e., in selecting unsafe tests before executing them), for both *Dataset 1* and *Dataset 2* we experimented with the ML models mentioned in Section II-C trained and validated using an 80/20 split.

As reported in Table IV, on *Dataset 1* SDC-Scissor accurately identified unsafe tests, with F1-score ranging between 35.1% and 56.1%. On *Dataset 2*, instead, it identified unsafe tests with F1-score ranging between 52.5% and 96.4%.

Complementary to the previous experiments, we investigated, in the context of *Dataset 2*, SDC-Scissor's ability to be more cost-effective compared to a *random-based baseline* that randomly selects from the dataset the tests to be executed

TABLE III
Datasets Summary

Dataset	Test Subject	Data Points Unsafe	Safe	Total
Dataset 1	BeamNG.AI moderate	1'334 (37%)	2'225 (63%)	3'559
Dataset 2	BeamNG.AI cautious	312 (26%)	866 (74%)	1'178
	BeamNG.AI moderate	2'543 (45%)	3'095 (55%)	5'638
	BeamNG.AI reckless	1'655 (96%)	74 (4%)	1'729
	Total	5'844 (48%)	6'260 (52%)	12'104

TABLE IV
Performance of the ML models with dataset split 80/20. The best results are shown in boldface.

Dataset	RF	Model	Prec.	Recall	F1-score
Dataset 1	**RF 1.5**	Logistic	**45.8%**	60.9%	52.3%
		Naïve Bayes	40.2%	**92.5%**	**56.1%**
		Random Forest	41.3%	30.5%	35.1%
Dataset 2	**RF 1**	Logistic	**43.3%**	87.3%	**57.9%**
		Naïve Bayes	36.7%	**92.1%**	52.5%
		Random Forest	40.7%	79.4%	53.8%
Dataset 2	**RF 1.5**	Logistic	78.1%	**65.3%**	**71.1%**
		Naïve Bayes	**79.3%**	53.2%	63.6%
		Random Forest	75.8%	62.7%	68.6%
Dataset 2	**RF 2**	Logistic	**99.6%**	82.8%	90.4%
		Naïve Bayes	98.7%	**94.3%**	**96.4%**
		Random Forest	**99.7%**	92.7%	96.1%

[19]. Specifically, SDC-Scissor was trained on 70% of tests from *Dataset 2* and tested on the remaining 30% of tests, while the random-based baseline randomly selected the same amount of tests directly from the remaining 30% of tests. Our evaluation on *Dataset 2* shows that for all RF values the best performing ML model of SDC-Scissor (i.e., Logistic) reduced the time spent in running safe/unnecessary tests than a random baseline strategy with a speed-up of circa 170%. On *Dataset 1*, instead, SDC-Scissor speed up testing up to 158% for the Naïve Bayes and 107% for Logistic.

V. Conclusions

This paper presented SDC-Scissor, a ML-based test selection approach that classifies SDC simulation-based tests as likely (or unlikely) to expose faults before executing them. SDC-Scissor trains ML models using input features extracted from driving scenarios, i.e., SDC tests, and uses them to classify SDC tests before their execution. Consequently, it selects only those tests that are predicted to likely expose faults. Our evaluation shows that SDC-Scissor successfully selected unsafe test cases across different driving styles and drastically reduced the execution time dedicated to executing safe tests compared to a random baseline approach.

As future work, we plan to replicate our study on further SDC datasets, AI engines and SDC features to study how the results generalize in the autonomous transportation domain. Additionally, given our close contacts with the BeamNG.tech team, we plan the integration of SDC-Scissor into BeamNG.tech environment to enable researchers and SDC developers to use SDC-Scissor as a cost-effective testing environment for SDCs. Finally, we plan to investigate the use of SDC-Scissor in other CPS domains, such as drones, to investigate how it performs when testing focuses on different types of safety-critical faults. Specifically, important for this is to investigate approaches that are more human-oriented or are able to integrate humans into-the-loop [14], [24], [26], [27].

Acknowledgements

We gratefully acknowledge the Horizon 2020 (EU Commission) support for the project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS) and the DFG project STUNT (DFG Grant Agreement n. FR 2955/4-1).

References

[1] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1016–1026. IEEE, 2018.

[2] A. Afzal, D. S. Katz, C. L. Goues, and C. S. Timperley. A study on the challenges of using robotics simulators for testing, 2020.

[3] R. Baheti and H. Gill. Cyber-physical systems. *The impact of control technology*, 12(1):161–166, 2011.

[4] C. Berger. Towards continuous integration for cyber-physical systems on the example of self-driving miniature cars. In J. Bosch, editor, *Continuous Software Engineering*, pages 117–126. Springer, 2014.

[5] M. E. R. Bezerra, A. L. I. Oliveira, and S. R. L. Meira. A constructive rbf neural network for estimating the probability of defects in software modules. In *International Joint Conference on Neural Networks*, pages 2869–2874, 2007.

[6] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella. Automated test cases prioritization for self-driving cars in virtual environments. *CoRR*, abs/2107.09614, 2021.

[7] E. Bondi, D. Dey, A. Kapoor, J. Piavis, S. Shah, F. Fang, B. Dilkina, R. Hannaford, A. Iyer, L. Joppa, and M. Tambe. AirSim-w: A simulation environment for wildlife conservation with uavs. In E. W. Zegura, editor, *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies, COMPASS*, pages 40:1–40:12. ACM, 2018.

[8] R. Caruana and A. Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *In Proc. 23 rd Intl. Conf. Machine learning (ICML'06*, pages 161–168, 2006.

[9] E. Castellano, A. Cetinkaya, C. H. Thanh, S. Klikovits, X. Zhang, and P. Arcaini. Frenetic at the SBST 2021 tool competition. In *International Workshop on Search-Based Software Testing*, pages 36–37. IEEE, 2021.

[10] H. Chen. Applications of cyber-physical system: A literature review. *Journal of Industrial Integration and Management*, 02(03):1750012, 2017.

[11] A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun. CARLA: an open urban driving simulator. In *Conference on Robot Learning*, volume 78 of *Machine Learning Research*, pages 1–16, 2017.

[12] A. Gambi, T. Huynh, and G. Fraser. Generating effective test cases for self-driving cars from police reports. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM Press, 2019.

[13] A. Gambi, M. Mueller, and G. Fraser. AsFault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, May 2019.

[14] G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. Gall. Exploring the integration of user feedback in automated testing of Android applications. In *Int'l Conf. on Software Analysis, Evolution and Reengineering*, 2018.

[15] T. K. Ho. The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

[16] F. Ingrand. Recent trends in formal validation and verification of autonomous robots software. In *International Conference on Robotic Computing*, pages 321–328, 2019.

[17] N. Kalra and S. Paddock. Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? *Transportation Research Part A: Policy and Practice*, 94:182–193, 12 2016.

[18] A. Kaur and R. Malhotra. Application of random forest in predicting fault-prone classes. In *2008 International Conference on Advanced Computer Theory and Engineering*, pages 37–43, 2008.

[19] S. Khatiri, C. Birchler, B. Bosshard, A. Gambi, and S. Panichella. Machine learning-based test selection for simulation-based testing of self-driving cars software, 2021.

[20] J. Kim, S. Chon, and J. Park. Suggestion of testing method for industrial level cyber-physical system in complex environment. In *International Conference on Software Testing, Verification and Validation Workshops*, 2019.

[21] D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia. A test case prioritization genetic algorithm guided by the hypervolume indicator. *IEEE Trans. Software Eng.*, 46(6):674–696, 2020.

[22] NVIDIA Corporation. NVIDIA DRIVE Constellation - NVIDIA Developer, Dec 2020.

[23] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio. Sbst tool competition 2021. In *International Conference on Software Engineering, Workshops*. ACM, 2021.

[24] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *International Conference on Software Maintenance and Evolution*, pages 281–290. IEEE, 2015.

[25] C. Sammut and G. I. Webb, editors. *Logistic Regression*, pages 631–631. Springer US, Boston, MA, 2010.

[26] A. D. Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *International Symposium on Foundations of Software Engineering*, pages 499–510. ACM, 2016.

[27] The Washington Post. Pedestrian in self-driving uber crash probably would have lived if braking feature hadn't been shut off, ntsb documents show, 2019.

[28] S. Yoo and M. Harman. Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689–701, 2010.