Continuous Integration and Testing for Autonomous Racing Software: An Experience Report from GRAIC

Minghao Jiang[1], Kristina Miller[1], Dawei Sun[1], Zexiang Liu[2], Yixuan Jia[1], Arnab Datta[1], Necmiye Ozay[2] and Sayan Mitra[1]

[1]University of Illinois at Urbana-Champaign [2]University of Michigan

***Abstract*— Running a simulation-based autonomous race is a complex software testing task. The autonomy software has to be closed with a simulator with proper interfaces, and the tests for this closed system have to evaluate collisions and other scoring functions, ideally with deterministic results. Since autonomy software is constructed as a pipeline of heterogeneous modules, achieving scalability and determinism of these tests is a challenge. In this work, we present Autonomous System Operations (`AutOps`) a continuous integration (CI) and testing framework for autonomy software and we evaluate it in the context of a new autonomous racing competition called GRAIC, currently integrated with the CARLA vehicle simulator. `AutOps` is built using Jenkins, Docker, and Amazon Web Services. It allows completely automated, concurrent testing of controllers in complex simulated environments. We report experimental results evaluating the determinism and test isolation achieved with our design.**

I. Introduction

There has been an explosion of interest around autonomous racing in recent years [1]–[8]. In this paper, we explore the challenging problem of software testing for such autonomous races. Running a simulation-based race poses a difficult testing problem: the autonomy software has to be executed in a simulation environment and the results have to be computed. In turn, these results depend on collision detection and timings. Autonomous software is constructed as a pipeline of different modules for perception, planning, decision-making, and controls. These modules are highly *heterogenous* in terms of the data structures, interfaces, levels-of concurrency, sparsity of computations, and types of libraries used for implementing them. Heterogeneity makes end-to-end testing challenging. Tests can be *flaky* [9] and individual tests are usually excessively long. Additionally, tests can be difficult to automate because the pipeline must be closed and instantiated with a simulation environment.

In this paper, we show how some of these challenges can be met by a *continuous integration (CI)* pipeline for autonomy software. Our prototype system is called `AutOps`. Continuous integration and delivery (CI/CD) are software engineering techniques that allow developers to commit code changes frequently to a version control system such that each code commit triggers a large number of automated builds, tests, and possibly even deployments. Continuous practices have been shown to dramatically reduce the cost and the time for feedback, patching, and releases for large codebases [10]. CI has become standard practice, and increasingly, advanced testing and verification approaches are being deployed through CI at software firms like Google, Amazon, and Facebook [11]–[15].

In designing `AutOps` for testing autonomy software, we have aimed to make testing automatic, deterministic, concurrent, and non-interfering (for concurrent instances). These requirements are explained in more detail in Section III. Our implementation relies on state-of-the-art CI, serverless computing, and containerization tools like Jenkins, AWS Lambda, Docker. We evaluate `AutOps` in the context of a new autonomous racing competition called GRAIC (Section II). The evaluations show that while `AutOps` can achieve full automation, concurrency, and some degree of test-isolation, determinism remains a challenge for certain types of autonomous races.

II. GRAIC Software Framework

Why yet another race? Recently, there has been significant advances in controller synthesis algorithms [16]–[22] which is an important sub-problem of autonomous racing. At the same time, we are still sorely missing benchmark suites for comparing these algorithms. Unlike perception benchmarks like ImageNet [23], creating benchmarks for control and autonomy is much more challenging because they require a complete executable specification for a closed-loop system, the dynamics of the ego vehicle, the static environment, the behaviors of active and passive agents in the environment, their interactions, and the perception and control interfaces for the vehicle. Each of the controller synthesis tools mentioned above, for example, use unique vehicle models, different computing platforms (Matlab, Python, Java), and different abstractions for perception, all of which make fair comparison a Herculean task.

Our response to this challenge is the *Generalized Racing Intelligence Competition (GRAIC)* [24] benchmarking framework for autonomous racing. Figure 1 is an example screenshot from GRAIC. In this competition, each participant creates a *controller function* that drives an *ego-vehicle* in different simulated tracks with active and passive obstacles. Unlike the competitions mentioned above, the GRAIC framework can work with arbitrary vehicle models. Our `AutOps` pipeline (Section III) tests the participant's controller code automatically on four different vehicles and three different tracks, and generates a score which depends on safety and speed.

This competition focuses on racing strategy, decision, and control, and *not* on perception. Because of this focus, we provide a *perception oracle*. All the interfaces built using Robot Operating System (ROS) [25]. Currently GRAIC uses the CARLA Simulator (CARLA) [26] and because of the standard ROS interfaces, in the future it should be possible, and indeed we plan to, connect it with other simulators like Gazebo [27], FlightMare [28]. Figure 2 shows the architecture of GRAIC.

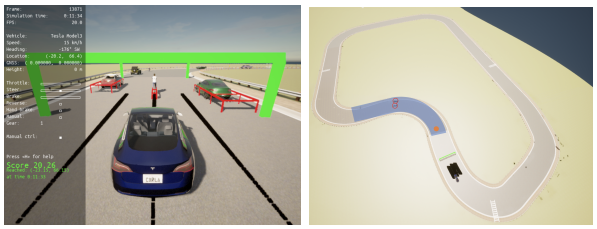

Fig. 1: Screenshot of ego vehicles point of view with perception oracle outputs (*left*). The lane markers are shown in black, the dynamic obstacles are bounded with red rectangles, and the goal waypoints are shown in green. One of the race tracks is shown on the right.

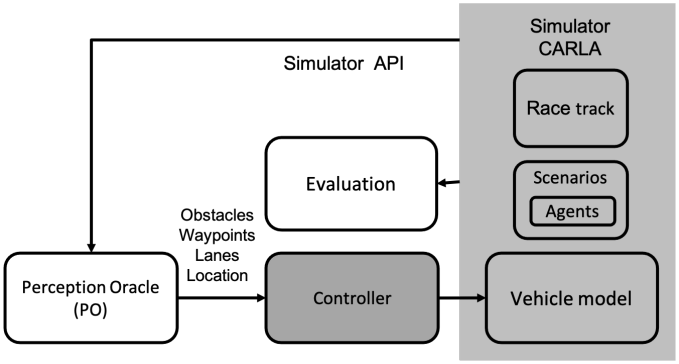

Fig. 2: The architecture of the GRAIC software framework. The participants submit controller code, which is composed with the perception oracle, a vehicle system, a track, and a scenario for executing a race. The evaluation module checks for collisions and calculates the score based on collisions and timing.

In this section, we first discuss our perception oracle abstraction. Next, we detail the vehicle control interface. Finally, we describe the different scenarios, environments, and race configurations that the competitors algorithms will have to deal with.

Perception Oracle (PO): GRAIC's perception oracle periodically outputs ground-truth object detection results in the neighborhood of the ego vehicle as shown in Figure 1. This notion of a PO for control design was introduced in [19]. In more detail, the PO publishes information in several `rostopics` for (i) obstacles like vehicles, cyclists, and pedestrians, (ii) forthcoming waypoints, (iii) current and adjacent lanes, and (iv) global position, orientation, and velocity of the ego vehicle.

Vehicle Control Interface: We have two types of vehicle models simulated in GRAIC: (a) complex vehicle models from CARLA which can be four or two-wheel vehicles and (b) kinematic Dubin-type models. The CARLA models are not available to the participants in any analytical form, beyond some basic information such as length, mass, and wheelbase. For the kinematic Dubin-type model, detailed lateral and longitudinal dynamics are released for this type of vehicles. In GRAIC, the controller function has to publish to the same rostopics to control both types of vehicles. These rostopics can either be ackermann control from ROS or vehicle control from CARLA.

Tracks, scenarios, and race configurations: To test or race autonomy software (controller code, perception oracle) we have to fix a vehicle, a track, and the behavior of all other agents on the track. In the parlance of CARLA, which we adopt for this paper, a *scenario* is a set of actors (vehicles, pedestrians, etc.) with specific behavior (e.g., pedestrian crossing the road) that can be spawned at particular positions (*spawn points*) on a track. The spawning can be controlled by *triggers* such as the ego vehicle coming within some distance of the spawn-point. The collection of all the scenarios and spawn points together define what we call a *race configuration* or simply a *race*.

Once the ego vehicle and its software is fixed, and a race is fixed, if the agents are deterministic, then the overall closed system should have a unique execution. This is an idealized view of the closed system. Even with perfectly deterministic algorithms for the ego vehicle and other agents, the ROS interfaces and simulators introduce enough nondeterminism and break this ideal. As we shall see in Section IV, this makes testing challenging. We will also discuss our design of GRAIC for making the races more deterministic and the experimental results.

III. `AutOps`: CI Pipeline for Autonomous Racing

In this section, we discuss the design of `AutOps`—an end-to-end pipeline for continuous testing of autonomous racing software. We discuss `AutOps` as it is currently deployed for GRAIC at CPS-IoTWeek 2021 [24].

The key design requirements of `AutOps` are: (i) **Automated.** When a participant submits a controller, tests/races for different vehicles and tracks should be triggered automatically, i.e., without manual intervention. (ii) **Concurrent.** A set of controllers should be testable on different tracks, concurrently on the same testing workstation. (iii) **Deterministic.** For a deterministic race configuration and controller, the result or the race score should be unique. (iv) **Non-Interference** Concurrent tests should have minimal impact on each other in performance.

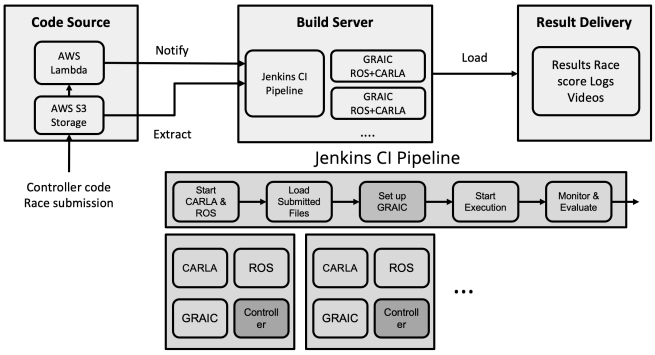

Fig. 3: `AutOps` continuous integration and testing system.

The design of `AutOps` (Figure 3) consists of three main columns: (1) the code source, (2) the build server, and

(3) the results delivery subsystem. The participants upload their controller code to the code source, then the build server extracts the code to compile, launch, and execute the appropriate races/tests. Finally, the results delivery subsystem aggregates and sends out the results, logs, and videos, as emails or web updates.

A. Code Source

In our design, the code source is developed using AWS S3 Storage and AWS Lambda. AWS S3 cloud storage stores the participant's submitted controller. AWS Lambda is a serverless computing service that can execute programs without provisioning and managing servers. Once controller files are uploaded, AWS S3 triggers AWS Lambda. AWS Lambda contains a Python program which sends a HTTP request to notify our Build Server to extract controller files and perform further processing. In this way, the tests can be automatically triggered by submissions.

B. Build Server

The build server is the heart of `AutOps` and is responsible for creating complete executable containers for each race.

A *container* provides a computing environment for software applications to run in virtual isolation, on the same physical platform or on the cloud. Inside the container, programs and dependencies are packaged so that the container can be quickly and reliably run across different hosting operating systems (Linux, Windows, MacOS) on different hardware platforms (desktop, laptops, servers, etc.). The containerization tool we use in `AutOps` is Docker [29]. When multiple Docker containers run concurrently, the software in each container are encapsulated and isolated. Each container runs on virtual memory so that it is separated from the host operating system. This level of encapsulation and isolation ensures the security for both the container and host operating system.

`AutOps` uses *continuous integration (CI)*. Specifically, the Jenkins [30] CI tool is used to run the scripts that orchestrate the execution of races in Docker containers. As shown in Figure 3, both Jenkins and GRAIC are deployed in different Docker containers for isolation. In the Jenkins Docker, we configure a multi-stage pipeline to perform the workflow needed for setting up the GRAIC and executing controller code. In a GRAIC Docker, there are 4 main components: CARLA, ROS, GRAIC, and the participants' submitted controller files. Multiple GRAIC containers can run concurrently with CARLA and ROS set to run on different network ports.

Our scripts through Jenkins orchestrate the GRAIC containers by sending out commands to every GRAIC container to launch CARLA, ROS, and GRAIC. The controller files are extracted from the code source by Jenkins and sent to every GRAIC container. After the software components are set up, the execution of the race (with the participants' controller) begins. During execution, Jenkins can monitor and log the performance and output of GRAIC containers. Once the executions are complete, Jenkins can load the results including score files, logs, and videos from every GRAIC container.

C. Result Delivery System

When the tests are finished, the results are extracted from the build server. The results include score, runtime logs, and videos recorded during the execution of vehicle controller. The score is calculated based on some score function. For example, this can be a summation of time (in seconds) that the ego vehicle spends to reach the next milestone waypoint. During the race, if the ego vehicle hits an obstacle or deviates from the road, penalty points are added to the score. The controllers that receive lower scores are ranked higher in the leader boards. A video is created by subscribing to a rostopic that publishes images from a third-person view camera of the ego vehicle. The output images are transformed into a video. Finally, the results are uploaded to online leader boards and the competitors are notified via email.

IV. Experimental Evaluation

In this section, we present a preliminary evaluation of `AutOps` with respect to the determinism and isolation requirements. All tests reported here are conducted on an Ubuntu Desktop with Intel Xeon(R) Silver 4110 CPU (2.10GHz), 32 GB RAM, and Nvidia Quadro P5000 GPU (16 GB).

A. More Deterministic Races

Determinism is necessary for repeatability of tests. Without determinism, the race winner may not be determinable or the results may be unfair. Achieving determinism of complex, system-level tests is known to be a challenging problem [31]. The problem is exacerbated for GRAIC since vehicle simulators are prone to introduce nondeterminism through collisions, concurrency, and complex dynamical models. In our own experiments, the simulations of a simple vision-based, lane-tracking PID controller diverged significantly from identical initial conditions because of differences in ROS message update delays.

In our initial implementation of GRAIC, the races were far from deterministic because the spawning of scenarios was randomized and also the start-ups of the controller, the CARLA simulator, and the ROS nodes were not synchronized. Towards having more deterministic races we have taken the following measures:

- We configured ROS and CARLA to run in synchronous mode by setting the ROS to be the only CARLA client that could perform the `tick()` operation for advancing simulation time. This minimizes error caused by the delay of control input transported over ROS networks.
- We used bash scripts to ensure that the initiation of the above modules is done in the exact order and that the controller code is started at the same state in the environment.
- We used the CARLA `scenario_runner` [32] framework to customize the triggering of scenarios. Each scenario is triggered when the ego vehicle approaches a spawn point on the tracks.

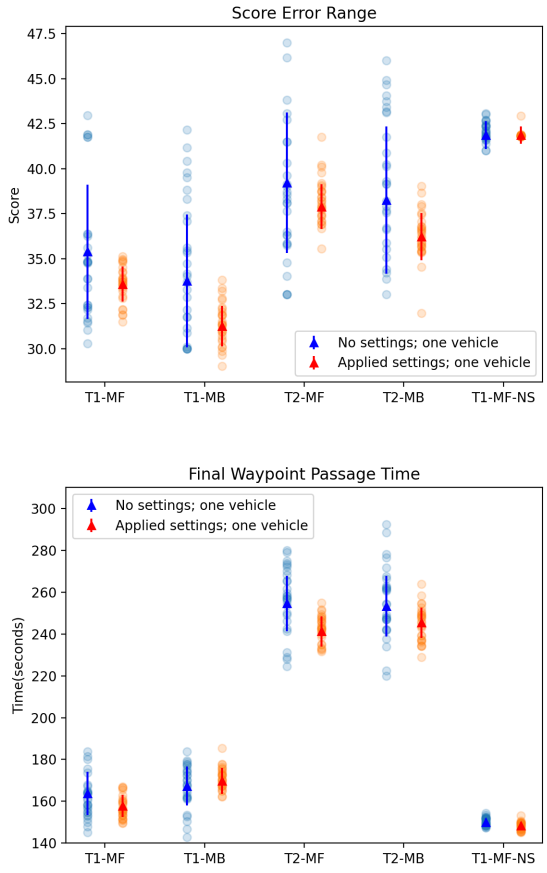

Fig. 4: Raw scores and standard deviations of scores (top) and final waypoint passage time (bottom) before and after we made races more deterministic. T1 denotes "Track1", T2 denotes "Track2", MF denotes the CARLA vehicle, MB denotes the kinematic vehicle, and NS denotes a race with no obstacles.

Results: We tested `AutOps` running GRAIC with the two tracks and two types of vehicle models. For each of these four races we look at results both before and after applying the determinism settings described above. The obstacle in these tests is another vehicle that moves forward with a constant speed. For testing before applying determinism settings, the obstacle vehicle can appear anywhere along the track. In contrast, for testing after applying determinism settings, the vehicle starts to move when the ego vehicle approaches within a certain fixed distance. This gives us 4 race configurations: Track1 CARLA vehicle (T1-MF), Track1 kinematic vehicle (T1-MB), Track2 CARLA vehicle (T2-MF), Track2 kinematic vehicle (T2-MB). In addition, we have a fifth race configuration without obstacles on the track (T1-NS).

Using `AutOps` we ran 28 tests with a baseline vehicle controller, on each these five configurations, both before and after applying the determinism settings. The CPU usage across different races stayed between 30-35% and the average memory usage was around 4.5%. The collected scores and race completion times are shown in Figure 4. It is clear from these plots that the measures taken above make the races more deterministic. On average, the standard deviation of the score is decreased by 70% across the board. We also observe that the races with fewer collisions are more deterministic; an extreme version of this is the one with no obstacles (T1-NS).

Discussion: We believe that the remaining variations in the tests have two contributing factors, and we do not see a clear remedy for either: (1) ROS's TCP/IP-based messaging. The delay in transmission of packets can result in the actuator receiving control inputs at slightly different timestamps; eventually, it leads to discrepancy in the behavior of ego-vehicle. Upgrading to ROS2 [33] with support of real-time programming can potentially alleviate this issue. (2) Non-determinism introduced by collisions and contact [34]. Every time the ego vehicle collides with an obstacle in a given race configuration, the outcome can be different. As shown in Figure 4, the standard deviation of the race with no collisions is the smallest. These two sources of nondeterminism are likely to bedevil future autonomous racing competitions.

B. Interference across Concurrent Instances

In order to study the non-interference requirement of `AutOps`, we have conducted another set of experiments. Here, we fix a race configuration across all runs. First, we run a race by itself. Next, we run two races concurrently. Then, we run three races concurrently. We continue this until we run up to five races concurrently. For each of these races, we report the returned score and the total CPU usage.

Results: We are reassured to observe that as the number of concurrent instances increases, the CPU usage percentage increases as expected. Furthermore, the scores for races remain roughly the same. This suggests that scaling up the number of concurrent executions has limited effect on individual races.

Discussion: We are currently exploring the limits of non-interference in `AutOps`. We expect the nice non-interference observed in the above experiment to break once the number of instances and the load of each race reaches the limits of compute capability the underlying workstation.

V. Conclusions

We explored the problem of testing for autonomous systems and introduced introduced a *continuous integration (CI)* pipeline for autonomy software called `AutOps`, which is implemented using Jenkins, Amazon Web Services, and Docker. We evaluated `AutOps` in the context of a competition called GRAIC. Our experiments with `AutOps` show that it can achieve fully automatic and concurrently running test instances; it also offers test-isolation up to a point. The experiments confirm the difficulty of achieving full determinism for testing autonomous races. The testing capabilities of `AutOps` are exciting, as it allows for automatic benchmarking of various control and autonomy approaches.

References

[1] M. A. Brady and M. O'Kelly, "The official home of f1/10." [Online]. Available: http://f1tenth.org/

[2] P. Foehn, D. Brescianini, E. Kaufmann, T. Cieslewski, M. Gehrig, M. Muglikar, and D. Scaramuzza, "Alphapilot: Autonomous drone racing," *arXiv preprint arXiv:2005.12813*, 2020.

[3] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "Torcs, the open racing car simulator," *Software available at http://torcs. sourceforge. net*, vol. 4, no. 6, p. 2, 2000.

[4] R. Madaan, N. Gyde, S. Vemprala, M. Brown, K. Nagami, T. Taubner, E. Cristofalo, D. Scaramuzza, M. Schwager, and A. Kapoor, "Airsim drone racing lab," *arXiv preprint arXiv:2003.05654*, 2020.

[5] J. Betz, A. Wischnewski, A. Heilmeier, F. Nobis, T. Stahl, L. Hermansdorfer, B. Lohmann, and M. Lienkamp, "What can we learn from autonomous level-5 motorsport?" in *9th International Munich Chassis Symposium 2018*. Springer, 2019, pp. 123–146.

[6] A. Censi, L. Paull, J. Tani, and M. R. Walter, "The ai driving olympics: An accessible robot learning benchmark," in *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.

[7] M. O'Kelly, H. Zheng, A. Jain, J. Auckley, K. Luong, and R. Mangharam, "Tunercar: A superoptimization toolchain for autonomous racing," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2020, pp. 5356–5362.

[8] M. Wen, J. Park, and K. Cho, "A scenario generation pipeline for autonomous vehicle simulators," *Human-centric Computing and Information Sciences*, vol. 10, pp. 1–15, 2020.

[9] J. Micco, "The state of continuous integration testing@ google," 2017.

[10] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[11] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[12] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 11–20. [Online]. Available: https://doi.org/10.1145/3377813.3381347

[13] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous formal verification of amazon s2n," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 430–446. [Online]. Available: https://doi.org/10.1007/978-3-319-96142-2_26

[14] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 13–25. [Online]. Available: https://doi.org/10.1145/3209108.3209109

[15] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 21–30.

[16] S. L. Herbert, M. Chen, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin, "Fastrack: A modular framework for fast and guaranteed safe motion planning," in *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*. IEEE, 2017, pp. 1517–1522.

[17] S. Vaskov, U. Sharma, S. Kousik, M. Johnson-Roberson, and R. Vasudevan, "Guaranteed safe reachability-based trajectory design for a high-fidelity model of an autonomous passenger vehicle," in *2019 American Control Conference (ACC)*. IEEE, 2019, pp. 705–710.

[18] C. Fan, K. Miller, and S. Mitra, "Fast and guaranteed safe controller synthesis for nonlinear vehicle models," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 629–652.

[19] K. Miller, C. Fan, and S. Mitra, "Planning in dynamic and partially unknown environments," in *In Proceedings of 7th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS'21)*, July 2021.

[20] M. Rungger and M. Zamani, "Scots: A tool for the synthesis of symbolic controllers," in *Proceedings of the 19th international conference on hybrid systems: Computation and control*, 2016, pp. 99–104.

[21] D. Q. Mayne, "Model predictive control: Recent developments and future promise," in *Automatica*, vol. 50. Pergamon, 2014, pp. 2967–2986.

[22] M. N. Zeilinger, C. N. Jones, and M. Morari, "Real-time suboptimal model predictive control using a combination of explicit MPC and online optimization," *IEEE TAC*, vol. 56, no. 7, pp. 1524–1534, 2011.

[23] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[24] Minghao Jiang and Zexiang Liu and Kristina Miller and Dawei Sun and Arnab Datta and Yixuan Jia and Sayan Mitra and Necmiye Ozay, "Graic: A software framework for autonomous racing," https://popgri.github.io/Race/, 2021.

[25] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[26] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[27] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, pp. 2149–2154.

[28] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," *arXiv preprint arXiv:2009.00563*, 2020.

[29] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[30] CloudBees, "Jenkins," https://www.jenkins.io/, 2011.

[31] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 830–840.

[32] "CARLA ScenarioRunner," https://carla-scenariorunner.readthedocs.io/en/latest, 2018.

[33] D. Thomas, W. Woodall, and E. Fernandez, "Next-generation ROS: Building on DDS," in *ROSCon Chicago 2014*. Mountain View, CA: Open Robotics, sep 2014. [Online]. Available: https://vimeo.com/106992622

[34] M. Posa, C. Cantu, and R. Tedrake, "A direct method for trajectory optimization of rigid bodies through contact," *The International Journal of Robotics Research*, vol. 33, no. 1, pp. 69–81, 2014.