STRIVE: A Co-Simulation-Based Testing Platform Enhanced with Runtime Monitors

Praanav Paatil, Daryna Datsenko, Mário Cardoso, Ana Sousa and André Matos Pedro
VORTEX-CoLab, Vila Nova de Gaia, Portugal
Email: {pranav.patil, daryna.datsenko, mario.cardoso, ana-cristina.sousa, andre.pedro}@vortex-colab.com

***Abstract*—As safety standards for autonomous driving systems continue to rise, the need for rigorous validation methods has become increasingly urgent. This paper combines three software engineering approaches for ensuring the safety and reliability of autonomous driving systems. The first approach evaluates system's performance through scripted simulations in a co-simulation platform, mimicking real-world conditions. The second approach focuses on generating monitors to identify faults, increasing confidence in the system's correctness and safety. The third approach includes continuous integration, enabling automatic testing throughout the development process.**

To achieve autonomous driving testing and monitoring, we propose integrating the co-simulation and monitoring pipeline into a testing framework. This framework allows us to continuously test the system's behavior in various scenarios, ensuring the safety and reliability of the autonomous driving system before deployment. We demonstrate the effectiveness of our approach through the STRIVE platform, which provides a solution for testing and monitoring autonomous driving systems. Our results show that this approach can help developers build safe and reliable autonomous driving systems, addressing a critical need in the industry from the first day of the system's development.

I. Introduction

The unpredictable nature of Autonomous Driving Systems (ADS) behavior needs rigorous validation methods to ensure the safety of Autonomous Vehicles (AVs). Traditional real-world test drives are time and cost-intensive [7], thus simulation-based solutions are being checked out. While simulation testing provides a solution, capturing all possible aspects of ADS in a single simulator is difficult.

Co-simulation, by addressing these limitations, presents a promising option for extensive testing. It enables a more comprehensive analysis of ADS behavior by merging multiple simulators which allows to cover various aspects of ADS. Furthermore, using a scenario-based testing approach within co-simulation allows for the development of scenario scripts that mimic real-world events.

In [9], Menzel et al. discussed the requirements for representing scenarios in various process steps defined by ISO 26262, a functional safety (FuSa) standard. According to FuSa, scenarios can be utilized to support the development process, including deriving requirements, developing hardware and software components, and proving the safety of these components in the test process. To ensure a common understanding of terms like scenario and scene, clear definitions of these terms are essential. In a recent survey [14], they presented their perspectives on this terminology, which we adopted throughout this paper. Moreover, the adoption of AI in safety-critical systems, such as autonomous driving, presents several challenges for FuSa and ISO 21448 – known as safety of the intended functionality (SOTIF). One of the significant concerns is ensuring the safety of pedestrians and passengers. ADS need to be able to make safe decisions in complex and unpredictable environments, such as when unexpected obstacles or weather conditions occur.

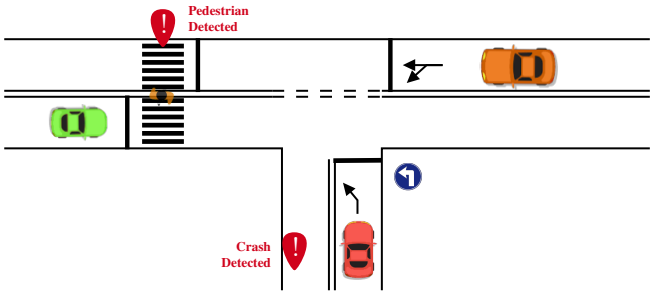


Fig. 1: The running example of co-simulation-based testing of an ego vehicle's behavior in an urban intersection scenario.

With co-simulation and scenario-based techniques, it is still difficult to detect potential runtime errors. To address this issue, we propose the use of dynamic verification, specifically Runtime Verification (RV). RV tools constructs software monitors based on system requirements, which integrate neatly into the co-simulation environment. These monitors continuously monitor the ADS without interfering with its functioning, allowing for real-time fault identification and analysis and ensuring compliance with established guidelines and standards.

Furthermore, by encompassing scenario-based testing within co-simulation along with monitoring pipeline, our framework facilitates Continuous Integration (CI). CI allows for automatic, and systematic testing, providing an effective option for incremental AV validation and requirements traceability. Incorporating multiple simulators, RV tools, and scenario-based testing, our co-simulation platform, known as STRIVE, serves as a virtual environment to test and validate ADS functionalities, aiming to enhance reliability and automate safety certification processes through integration with CI.

A. Running Example

Our platform validates two critical aspects such as communications and Ego rules in an urban intersection scenario (see Fig. 1) involving connected vehicles (*orange* and *green*), a vulnerable road user, and an approaching ego vehicle (*red*). To validate V2X Communication amongst vehicles and ensure compliance with ETSI standards, we couple the runtime monitors generated by the RV tool to monitor message exchanges. To validate that the Ego Vehicle Behavior adhere to traffic rules, as per the Vienna convention [10], we couple the runtime monitors within our platform which are generated by another RV tool to validate the safe behaviour of the ego vehicle.

We show how to monitor these aspects effectively. To monitor V2X communication, the platform incorporates the HAROS plugin [5], which generates runtime monitors. Additionally, to observe traffic rules, the platform integrates the RV tool described in [6]. Furthermore, we evaluate the resource consumption in both offline and online deployments, as well as the cost-effectiveness of operating these deployments on cloud infrastructure.

B. Contributions of the paper

This paper presents three contributions: i) an approach that integrates co-simulation-based testing, runtime monitoring, and CI; ii) a novel micro-service-based reference architecture that enables the monitoring of co-simulation-based testing; and iii) a dynamic verification model that illustrates the monitoring of autonomous vehicle behavior in an urban intersection.

C. Structure of the paper

The paper is structured into following sections: Section II overviews the related work in the field. Section III introduces the necessary definitions. The proposed approach is presented in Section IV, while the implementation details and reference architecture are described in Section V. Section VI presents the experimental results, and Section VII summarizes the findings and discusses future work.

II. Related Work

We observed various co-simulation frameworks, primarily an OpenCDA [17], a co-simulation platform for developing and testing Cooperative Driving Automation (CDA) systems where different CDA algorithms can be tested. Secondarily, a co-simulation platform [13], comprises two pieces of software namely Siemens Simcenter Amesim and Simcenter Prescan for testing Autonomous Driving Assistance Systems / Autonomous Driving (ADAS/AD), and lastly, a customized co-simulation environment [4], where readily available or custom data sets have been used for the development of automotive applications that can be utilized in ADAS. STRIVE integrates a monitoring pipeline which helps to detect runtime errors and enables continuous testing through CI, which are the notable differences with those co-simulation frameworks.

Temperekidis et al. [15] presents a technical approach for Runtime Verification (RV) of properties for the entire co-simulated system. The approach integrates a monitor synthesis tool at the master algorithm level of FMI-based co-simulation. Our approach is designed to be flexible and scalable, and do not impose constraints on the execution of the monitors. They can be easily coupled with existing robotic operating system (ROS) middleware using a publish/subscribe mechanism, which can run on different containers or virtual machines.

Nickovic and Yamaguchi [11] also demonstrate how online monitoring can be integrated with ROS and its usability in robotics applications. PerceMon [2] is another online monitoring tool that has been integrated with CARLA [8] and ROS to monitor the properties of object detection and tracking algorithms, while Zapridou et al. [18] uses the RTAMT library to enable the validation of autonomous driving control by online monitoring over realistic driving scenarios.

To the best of our knowledge, no previous work has specifically addressed the monitoring of the combined effects of V2X communication and traffic laws and rules on an autonomous vehicle's behavior. Furthermore, none has proposed a reference architecture that can readily integrate with a CI/CD pipeline.

III. Preliminaries

A. Co-Simulation

Co-simulation, an integration of multiple simulators covering various aspects of cyber-physical systems (CPS) for validation purposes. There is a master algorithm responsible for coordinating the interaction among simulators and managing the information exchange between them. It plays a crucial role in orchestrating the syntactic and semantic interaction among the simulators. To ensure the proper functioning of co-simulation, synchronization techniques and communication patterns should be taken care.

a) Synchronization Techniques: Co-simulation synchronization techniques are used to ensure that the simulators run in a coordinated manner and exchange data accurately [3]. The master-slave technique is often used in co-simulation, where one simulator acts as the master and others act as slaves. The master simulator controls the simulation and exchanges data with the slave simulators at regular intervals. This will be the adopted synchronization technique.

b) Communication Patterns: Co-simulation communication patterns involve simulators exchanging data [3]. In ROS subsystems, the publish/subscribe architecture enables asynchronous, message-based communication via particular topics. Furthermore, the service-oriented architecture (SOA) design pattern allows for dynamic, standardized communication across services. This combination is appropriate for large-scale, distributed systems with a variety of platforms and architectures. This communication pattern combination will be the choice for the remaining sections.

B. Runtime Monitoring

Runtime monitoring is a verification technique in which monitors observe and compare a system's behavior in runtime. These lightweight and non-intrusive monitors compare runtime behavior to expected behavior (to its formalized properties).

Listing 1: An example of two simple properties in HAROS Property Language

```
globally: some /vehicle/orange/CA_service/transmitter/ITSG5/CAM within 10 ms
globally: /vehicle/red/CA_service/receiver/CAM requires /vehicle/orange/CA_service/transmitter/ITSG5/CAM within 100 ms
```

Listing 2: An example of two properties in STEM's specification language

```
(property (always (not (overlap (prop "/vehicle/orange") (prop "/vehicle/red")))))
(property (not (eventually (overlap (expand (prop "/vehicle/orange") 1.) (expand (prop "/vehicle/red") 1.) ))))
```

Users get notified if deviations occur, ensuring safety and early error identification.

a) HAROS [5]: It is a framework for analysis and quality improvement of robotic software developed using ROS but it can be adapted to other applications in other environments beyond the ROS ecosystem. It offers various plug-ins, out of which property-based testing generates runtime monitors based on formal properties that interest us.

As an example, we outline two properties that we can monitor using HAROS plug-in within a co-simulation platform, which are provided in Listing 1. The first property requires the co-simulation to begin sending messages within 10 milliseconds, while the second specifies that a message dispatched from `orange` vehicle must be received in `red` vehicle within 100 ms.

b) STEM Tool [6]: It is a tool designed to address the generation of monitors for CPS that require a combination of space and time requirements. STEM generates source code monitors from high-level specification language and employs a non-linear satisfiability solver as the monitoring procedure for LTLxMS formulas. This tool implements a monitoring procedure for the combination of metric spaces and temporal logic with applications in the ADS domain.

As we did before, we also present two sample properties for monitoring with STEM in Listing 2. Informally, the first property indicates that neither the `red` nor the `orange` vehicles shall collide, while the second property adds a safety margin of 1. unit to this constraint using the `expand` operator.

These properties will be utilized throughout the paper in conjunction with our running example.

C. Microservices and Containers

Microservices involve building applications as small, independent services that can be deployed and scaled separately. Containers allow applications to run across different environments. Together, they enable developers to break down applications into smaller, more manageable components and deploy them more efficiently. The benefits include improved scalability, flexibility, and resilience. Microservices allow developers to iterate and release new features more quickly, while containers enable efficient resource utilization and faster deployment times. The following sections assume that the reader is familiar with these concepts.

IV. Approach and Considerations

The proposed approach involves utilizing OpenSCENARIO [1], OpenDRIVE, and CARLA [8] tools to define scenarios that defines the behavior of the Ego vehicle and the surrounding environment to be co-simulated [1]. Our approach facilitates integration with other models, such as topological network models that can feed simulators like Artery V2X [12] or control systems that can be interconnected with Matlab/Simulink simulation environments.

To illustrate, let us consider two models. One model contains all the physical dynamics and surrounding environment of a vehicle, while the other defines the physical communication layer. The co-simulation module manages communication between the two simulators and may obtain co-simulation results using the FMI 2.0 [3] interface, which is known as a common and appropriate interface. Since FMI does not support distributed simulations, we can integrate FMI with ROS communication middleware.

To enable coupling of the simulation with the physical devices, it is necessary to abstract each simulator into the concept of model and solver [3]. This abstraction is the basis for connecting the communication and control layers to hardware devices in two ways: i) by using sub-systems in a laboratory test bench, and ii) by integrating the hardware in an cooperative intelligent transport ecosystem. The control and interconnection layer function like a synchronous pipeline, which receives a set of input data with a timestamp, processes the data in a stateful manner, and outputs another set of data in the same initial format, FMI 2.0. This abstraction provides an interface that can be adapted to various contexts and increases reliability due to the coupling of monitors at Input/Output. These software monitors are generated using an automated, error-free approach [6], [5], which reduces risks at the interconnection of hardware subsystems.

A. Co-simulation-based Testing

To provide clarity for the remaining sections, let us provide some preliminary definitions.

Definition 1. *The Co-simulation-based testing consists of generating unit tests or integration tests for the target system using multi-domain models and environments of the system in several distinct simulators.*

B. Real-time Monitoring Pipeline

The monitoring pipeline (according to Definition 2) acts as a healthy checker of the co-simulation-based testing. Validation through monitoring is an important approach for co-simulation-based testing.

Definition 2. *The real-time monitoring pipeline involves integrating software probes into the existing co-simulation process to continuously monitor messages or frames that have real-time and space constraints.*

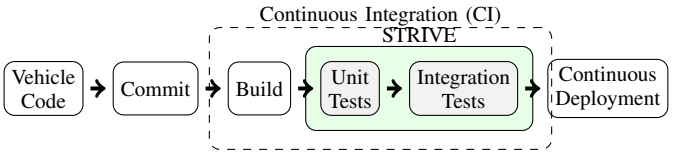


Fig. 2: Co-simulation-based Testing with Continuous Integration Approach: STRIVE inside conventional CI/CD pipeline.

The monitor is automatically constructed and independent of the ADS and has no direct impact on the system itself. The system evolves around the simulation of one or more pre-established scenarios and observations of the environment are sent to the system and a trace to the monitor that is capable of observing the space and time behaviors of the intervening vehicles. In turn, the ADS produces actions (turn left, accelerate, decelerate, among others) for the CARLA autonomous driving simulator, which causes a change in the simulation environment and produces new and endless observations (closed loop).

The monitor, upon receiving observations from the simulator, transforms them into a set of verdicts that indicate whether the system requirement is satisfied or not. Unlike the ADS, the monitor does not interfere with the simulation; only observes and produces verdicts for validation of the system during, after development, or during maintenance. We can observe in two ways. We can receive the trace and give verdicts as the system functions (online) or use it only to monitor a log or a ROS bag (offline). In this paper, we utilize both approaches.

C. Continuous Integration

The current Software Development Cycle does work with autonomous driving software but as the complexity increases it needs co-simulation for testing instead as the known test oracles and unit tests. Fig. 2 shows the reference CI pipeline that we approach in this paper. It is known that automation increases the quality of software and constant testing detects problems in the first phases of development.

When using co-simulation platforms for validation of autonomous vehicles, the question that may arises are 'How can we assure that testing is well-formed ?' or 'Are the messages being sent to the other sub-systems?'.

If no one is observing the experiments, 'How can someone observe the execution when co-simulation is automated ?' or even 'How can we trust the testing results?'. So our approach tries to fulfill this requirement. No human has to observe the testing. Therefore, we present our proposed architecture.

V. STRIVE Reference Architecture For Co-Simulation-based Testing

The STRIVE architecture is composed of four layers: Front End, Back End, Micro-services Communication, and Persistent Storage. Fig. 3 illustrates the logical architecture of STRIVE. The Front End layer includes the Dashboard and provides the capability to connect third-party applications, while the Back End layer is comprised of various services and pods.

The CARLA pod is responsible for providing vehicle dynamics, environment, and sensors. The virtual actuators simulate the physics of the Ego vehicle, while the environment models the world in which the vehicle operates, calculating the state of objects within it. The sensors include a variety of devices such as cameras, GNSS, LiDAR, radar, and IMU, all of which can be simulated within the driving simulator. The CARLA ROS Bridge serves as the interface between CARLA and the STRIVE co-simulation subsystems, and CARLA is not directly connected to any communication channel.

Another component of this architecture is Artery [12], which simulates V2X communications. This involves simulating a network from the packets to the physical layer using the simulator or actual RSUs to perform package sending via V2X protocols. Since Artery includes a proper ROS interface, rosomnet [16], there is no need for a ROS bridge. Artery communicates directly within the STRIVE namespace.

The STRIVE platform includes two monitor set containers: one for vehicle behavior and one for network performance. The vehicle behavior container evaluates whether the system under test has controlled the vehicle in compliance with defined rules, ground truth, or traffic laws. The network performance container evaluates whether communication metrics and rules are being met for communication to and from the Ego vehicle.

Finally, the STRIVE reference architecture includes a storage log component for managing and storing results from metrics. This allows users to generate reports and analyses on the data produced by the STRIVE platform.

A. Orchestration

The capabilities of the STRIVE platform include: enabling connection with external V2X Devices' clock; orchestrating co-simulation; deciding which component feeds the clock; providing automatic deployment; offering automatic monitoring; and scaling while adding new or existing ROS nodes. Regarding time synchronization, the STRIVE platform ensures that time is adjusted along the simulation through its time management component. This component coordinates the simulation time across all connected simulators and V2X devices.

The STRIVE platform includes the STRIVE Manager to facilitate co-simulation, which allows the joint simulation of multiple simulators that handle different domains or subsystems. This manager coordinates the simulation execution across all connected simulators and V2X devices, enabling the integration of different simulation tools.

B. Bundle with Testing Cases

The bundle in STRIVE is like a Git repository that includes the necessary artifacts and code for testing and validation of the ADS. This repository includes not only the code but also the unit tests required for the continuous integration process. The bundle is versioned using git, allowing for easy management of changes and rollbacks. This approach ensures that the

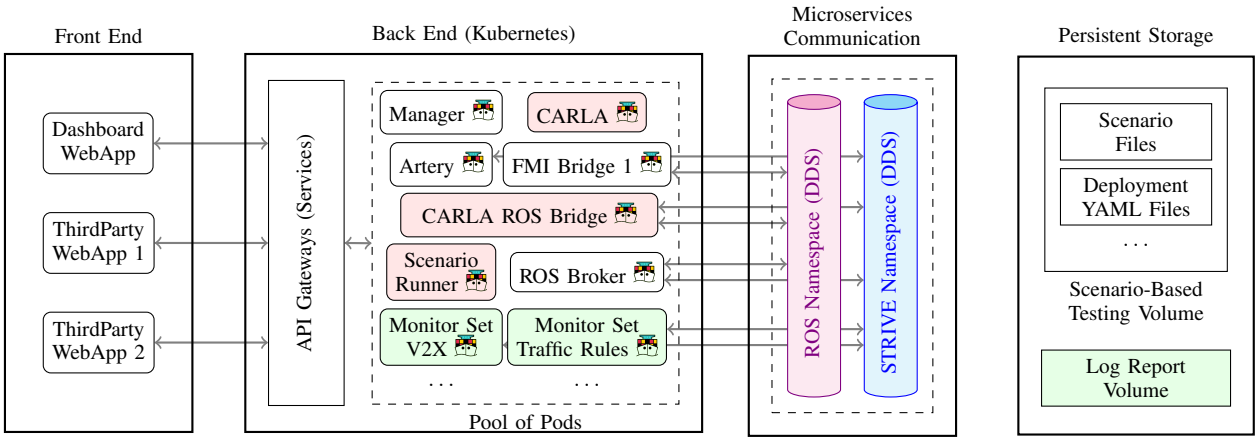


Fig. 3: The micro-services-oriented reference logical architecture.

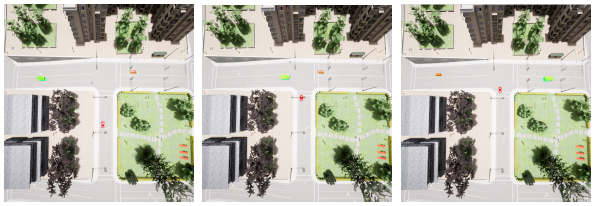


Fig. 4: Frame sequence of one CARLA run with two connected vehicles and one Ego vehicle using running example.

testing and validation process is repeatable and transferable, enabling the detection of potential faults and vulnerabilities early in the development process.

VI. Experimental Results

Since the initial deployment of our co-simulation monitors on the STRIVE platform, we have identified that there are delays in the messages exceeding 100 ms, with an upper bound of 200 s. This deviation from the ETSI EN 302 637-2 may affect the performance of the system during a run. Additionally, we have encountered issues with the initialization phase of the co-simulation platform, which is not functioning optimally. This illustrates that without monitoring from the beginning of development, we will be unable to make accurate detection and observations, meaning that real-time monitoring matters.

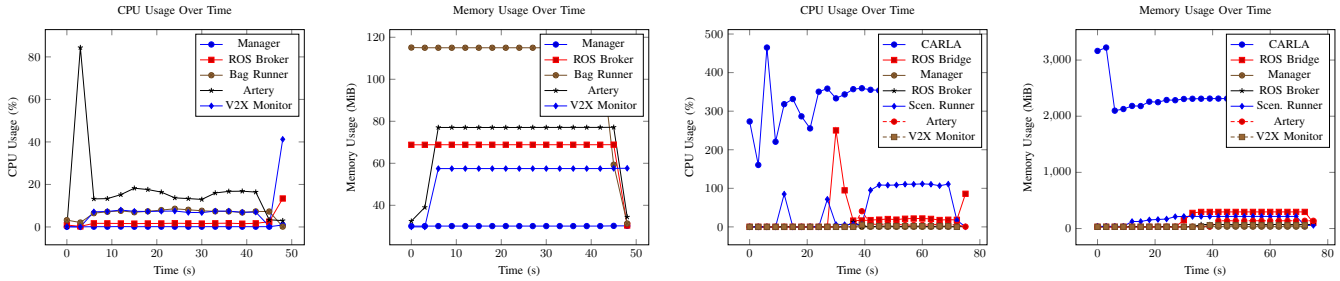
This section presents the resource utilization of the STRIVE software (see Fig. 5) during the validation of a scenario illustrated in Fig. 1, which further evolves during a run Fig. 4. The statistics were collected across all containers on following machine: Intel Core i7-9750H processor running at 2.60GHz with 12 threads, 16GB of DDR4 system memory (with a swap space of 2.0GB), and a TU106M GeForce RTX 2060 GPU.

We performed a comprehensive analysis of memory and CPU utilization across numerous containers in both ROS bag-based (offline) and CARLA-based (online) deployments measured every 3 seconds over a period of 90 and 120 seconds respectively. The average memory use for ROS bag deployment was 277 MB, whereas the deployment of CARLA required 2500 MB, primarily because of the CARLA container. The ROS bag and CARLA-based deployments showed a wide range of CPU consumption, ranging from 0.46% to 7.5% or 13.51% in totality and 0.083% to 301.55% or 352.92% in totality, respectively. These variations highlight the importance of continuously analyzing and optimizing the system to guarantee maximum scalability, informing resource allocation decisions, and prevent expensive performance issues.

We analyzed the CPU utilization of ROS bag and CARLA-based deployments to evaluate the cost-effectiveness of containerized applications in the cloud. ROS bag containers cost 0.5 USD per hour and CARLA containers cost 2 USD per hour under a flat price scheme (fixed price per core-hour, e.g., 0.5 USD). In other words, approximately 40 and 30 tests can be carried out per hour, costing 0.5 USD and 2 USD respectively for ROS bag and CARLA-based deployments. Our approach highlights the importance of taking cost into account when developing and implementing containerized applications and provides insights into cost-performance tradeoffs.

A. Orchestration/Management

Two possible configurations are available in the STRIVE: A ROS bags (offline) and CARLA-based (online) set-up. ROS bags created by the CARLA simulator, which are then obtained through the CARLA ROS Bridge. Scenario ROS bag runner, ROS Broker, Artery, and STRIVE manager are all part of the ROS bag setup. Upon the request of STRIVE manager, the scenario ROS bag runner initializes topics from the ROS bag file, enabling ROS Broker to generate vehicle threads for V2X CAM simulations. As long as Artery executes simulations, the generation of CAM messages by ROS Broker will continue until the ROS bag file ends. On the other hand, CARLA-based deployment uses CARLA simulator, Scenario runner, CARLA ROS Bridge, ROS Broker, Artery, and STRIVE manager. The scenario runner waits for the artery to be ready while the CARLA simulator runs the scenarios. Artery receives feeds from ROS Broker, which starts the scenario. Once scenario is finished, all services reach an end under the control of STRIVE manager.



(a) CPU – ROS bag. (b) Memory – ROS bag. (c) CPU – CARLA-based. (d) Memory – CARLA-based.

Fig. 5: CPU/Memory usage of test runs per container.

In a nutshell, ROS bag set-up is reasonably priced that help with identifying and correcting issues. User-provided services are possibility with a generic setup using STRIVE manager.

B. Co-Simulation-based Test Deployment

For deployment purposes, STRIVE employs a service-based, containerized approach for launching its test sets. The configurations for the services to be deployed are stored as Kubernetes deployment files (automatically translated from docker-compose file) and combined with test scenario files to form a complete set of test batches. Each batch is an independent unit that contains the necessary data for a single test deployment. To maintain interoperability, the test scenario files are also stored in persistent volumes that are accessible to the relevant running services.

VII. Conclusion and Future Work

STRIVE enables thorough testing of ADAS/AD systems by integrating CARLA and Artery simulators, forming a co-simulation for validation of AD behavior as well as V2X communication. Runtime monitors allows continuous observation of V2X communication and detects anomalies, while container-based deployment increases flexibility and scalability while saving time in offline deployments. It is critical for cost reduction to address inefficiencies and optimize resource consumption. As shown in Figure 5, the system takes an additional 5 seconds to complete a test run after almost finishing (approximately 10 seconds). This overhead must be minimized to improve efficiency. V2X monitors are faster than traffic rule monitors due to the intrinsic complexity of properties that combine time and space. As future work, we intend to reduce the overhead of data serialization between containers and assist runtime monitors with hardware accelerators.

Acknowledgments

This work is supported by the European Union/Next Generation EU, through Programa de Recuperação e Resiliência (PRR) [Project Route 25 with Nr. C645463824-00000063].

References

[1] Association for Standardisation of Automation and Measuring Systems, "OpenSCENARIO," retrieved 2023-03-23. [Online]. Available: https://www.asam.net/standards/detail/openscenario/

[2] A. Balakrishnan, J. Deshmukh, B. Hoxha, T. Yamaguchi, and G. Fainekos, "Percemon: Online monitoring for perception systems," in *RV*, ser. Lecture Notes in Computer Science, vol. 12974. Springer, 2021, pp. 297–308.

[3] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauß, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proceedings of the 9th International MODELICA Conference, September 3-5*, 2012.

[4] M. R. Cantas and L. Guvenc, "Customized co-simulation environment for autonomous driving algorithm development and evaluation," in *SAE Technical Papers*. SAE International, April 2021.

[5] R. Carvalho, A. Cunha, N. Macedo, and A. Santos, "Verification of system-wide safety properties of ROS applications," in *IROS*. IEEE, 2020, pp. 7249–7254.

[6] A. de Matos Pedro, T. Silva, T. F. Sequeira, J. Lourenço, J. C. Seco, and C. Ferreira, "Monitoring of spatio-temporal properties with nonlinear SAT solvers," in *FMICS*, ser. Lecture Notes in Computer Science, vol. 13487. Springer, 2022, pp. 155–171.

[7] F. Hauer, T. Schmidt, B. Holzmüller, and A. Pretschner, "Did we test all scenarios for automated and autonomous driving systems?" in *ITSC*. IEEE, 2019, pp. 2950–2955.

[8] S. Malik, M. A. Khan, and H. El-Sayed, "CARLA: car learning to act - an inside out," in *EUSPN/ICTH*, ser. Procedia Computer Science, vol. 198. Elsevier, 2021, pp. 742–749.

[9] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for development, test and validation of automated vehicles," in *Intelligent Vehicles Symposium*. IEEE, 2018, pp. 1821–1827.

[10] U. Nations, "Vienna convention on road traffic," 1968, retrieved 2022-04-11. [Online]. Available: https://unece.org/DAM/trans/conventn/Conv_road_traffic_EN.pdf

[11] D. Nickovic and T. Yamaguchi, "RTAMT: online robustness monitors from STL," in *ATVA*, ser. Lecture Notes in Computer Science, vol. 12302. Springer, 2020, pp. 564–571.

[12] R. Riebl, H. Gunther, C. Facchi, and L. C. Wolf, "Artery: Extending veins for VANET applications," in *MT-ITS*. IEEE, 2015, pp. 450–456.

[13] T. D. Son, A. Bhave, and H. V. der Auweraer, "Simulation-based testing framework for autonomous driving development," in *ICM*. IEEE, 2019, pp. 576–583.

[14] M. Steimle, T. Menzel, and M. Maurer, "Toward a consistent taxonomy for scenario-based development and test approaches for automated vehicles: A proposal for a structuring framework, a basic vocabulary, and its application," *IEEE Access*, vol. 9, pp. 147 828–147 854, 2021.

[15] A. Temperekidis, N. Kekatos, and P. Katsaros, "Runtime verification for fmi-based co-simulation," in *RV*, ser. Lecture Notes in Computer Science, vol. 13498. Springer, 2022, pp. 304–313.

[16] B. Vieira, R. Severino, E. V. Filho, A. Koubaa, and E. Tovar, "Copadrive - A realistic simulation framework for cooperative autonomous driving applications," in *ICCVE*. IEEE, 2019, pp. 1–6.

[17] R. Xu, Y. Guo, X. Han, X. Xia, H. Xiang, and J. Ma, "Opencda: An open cooperative driving automation framework integrated with co-simulation," in *ITSC*. IEEE, 2021, pp. 1155–1162.

[18] E. Zapridou, E. Bartocci, and P. Katsaros, "Runtime verification of autonomous driving systems in CARLA," in *RV*, ser. Lecture Notes in Computer Science, vol. 12399. Springer, 2020, pp. 172–183.