Seven Technical Issues That May Ruin Your Virtual Tests for ADAS

Mohamed El Mostadi, Hélène Waeselynck, Jean Marc Gabriel

▶ **To cite this version:**

Mohamed El Mostadi, Hélène Waeselynck, Jean Marc Gabriel. Seven Technical Issues That May Ruin Your Virtual Tests for ADAS. 32nd IEEE Intelligent Vehicles Symposium, Jul 2021, Nagoya, Japan. 10.1109/IV48863.2021.9575953 . hal-03377931v2

HAL Id: hal-03377931
https://hal.science/hal-03377931v2

Submitted on 6 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Seven Technical Issues That May Ruin Your Virtual Tests for ADAS

Mohamed El Mostadi[1,2], Hélène Waeselynck[1], and Jean-Marc Gabriel[2]

***Abstract*— A number of simulation platforms allow the validation of advanced driver assistance systems (ADAS) in virtual road environments. However, the development of virtual tests on top of such platforms may face technical issues. Some are related to the management of the modular and configurable architecture of the simulators. Others come from the physical aspects of the simulation. Also, time and concurrency issues may affect the control of dynamic scenarios. This paper shares our experience with the virtual testing of ADAS during a period of time of more than one year. The technical issues yielded simulation crashes, ill-controlled test executions, incorrect verdict assignments, and caused a waste of time in the running and analysis of useless tests. We discuss the issues and provide recommendations for the practitioners.**

I. Introduction

Advanced driver assistance systems (ADAS) and self-driving systems must be thoroughly validated in order to get confidence that they are reliable and safe. Their validation is done in part by test drives on public roads, but this strategy cannot be sufficient to meet safety requirements. It would take billions of accumulated miles to demonstrate that an automated driving system is safer than a human driver [1]. Hence, car manufacturers must also rely on virtual, simulation-based tests. Such tests can validate systems at various levels of maturity, from model-in-the-loop configurations to hardware-in-the-loop ones. Numerous existing simulation platforms (e.g., Virtual Test Drive [2], Simcenter Prescan [3], SCANeR [4], Carla [5], LGVSL [6]) provide facilities to create rich driving environments with virtual roads, other vehicles and pedestrians. They make it possible to consider a wide range of operational conditions, in order to identify the safety-relevant corner cases. Related work in the area has proposed test generation approaches based on combinatorial testing [7], metaheuristic search [8, 9], machine learning [10] or a hybridization of metaheuristic search and machine learning [11].

While the generation of tests has been much studied by academic work, there has been little feedback on the practical aspects of implementing the approaches in industrial settings. A notable exception is an experience report from Bosch [12]. They shared the lessons learned and impediments they had to overcome when applying search-based testing. Such practitioner's insights are important for the adoption of test approaches in industry.

In this paper, we also adopt a practitioner's view. The insights we provide do not pertain to a specific test generation approach, but rather to simulation-based testing in general. We discuss technical issues that may arise whenever tests are implemented on top of simulation platforms. The discussion is based on our experience with a commercial simulation platform during a period of time of more than one year. During this time, the platform was used by us and several R&D teams at Renault for test experiments targeting Autonomous Emergency Braking (AEB) systems. We report on the implementation issues that were faced. They yielded simulation crashes, ill-controlled test executions failing to trigger the intended scenarios, and incorrect test verdict assignments. The issues wasted time in the running and analysis of useless tests.

We believe that our experience may be useful to other practitioners. Indeed, simulators are very complex software artifacts. They typically have a modular and highly configurable architecture that the users may not perfectly understand. Bugs may affect any module. The configuring, interfacing, and orchestrating of all modules is difficult. Additionally, simulation involves aspects of the physics which may not be obvious to the users. Hence, the issues we faced are likely to also affect other users of similar platforms. This paper intends to raise awareness, as well as to provide recommendations on how to identify and address the issues.

The structure of the paper is as follows. Section II presents the simulation configurations we used. Sections III to IX present the issues. Each section starts with a generic description of the issue, then reports on the real-world examples we experienced, and finally provides recommendations.

II. Simulation Platform

All experiments involved model-in-the-loop (MIL) tests in an R&D context. Fig. 1 shows the architecture of the simulator that was used. It is based on a commercial simulation platform. We cannot disclose its identity.

The simulator is composed of several modules, including a set of generic modules of the platform plus two modules (Physical Car, Custom) that are specific to the system under test. The runtime of the platform (Scheduler) manages the execution of the modules based on their periodicity and priority. The user may select an execution mode. All test experiments used the synchronous mode with a fixed simulation time step, to have a full control of the simulation. The platform provides a scripting API to automate the experiments. A script triggers parametrized test actions such as SetSpeed(), ChangeLane(), etc.

In a conventional way, the simulator adopts a SUT-centric view that distinguishes the ego vehicle from the other ones. The target vehicles in the environment of the ego have a simplified behavior simulated by the generic traffic module. The simulation of the ego is paid more attention. The physical car module reproduces the characteristics of a real vehicle, and the custom module is in charge of the tested

[1]LAAS-CNRS, University of Toulouse, Toulouse, France. Contacts: mohamed.el-mostadi@laas.fr, Helene.Waeselynck@laas.fr.

[2]Renault Software Labs, Toulouse, France. Contacts: mohamed.m.el-mostadi@renault.com, jean-marc.gabriel@renault.com.

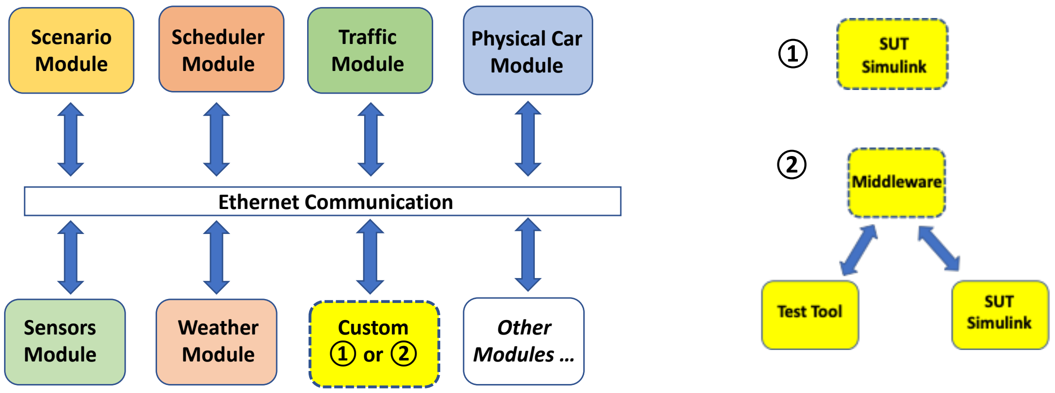


Figure 1. Architecture of the simulator. Left side: overview. Right side: two alternative configurations for Custom.

AEB logic for this vehicle. The experiments involved two configurations for Custom, shown in the right-hand side of Fig.1. In the first one, Custom is simply the external Simulink module that executes the AEB model. This is the standard configuration for MIL testing. It has the disadvantage that each time a new model is tested, there is some effort to interface it to the rest of the platform. The second configuration aims to reduce the interfacing effort: it provides a middleware-mediated connection and communication. In addition, this configuration includes a test tool that supports the execution of graphical test scenarios in place of scripts. Compared to the first configuration, the second one involves more modules and induces longer simulation times.

The issues reported in the rest of this document could occur in either configuration, for manually created tests or generated ones. The tests could come from us or from R&D engineers. They correspond to classical types of scenarios for an AEB like Cut-in, Cut-out, etc.

III. The Big Bang Integration Issue

The execution of virtual tests requires many modules to be connected and work together. Each of them may contain bugs, or they may not be properly interfaced. When a simulation run fails, it may be very difficult to diagnose the root cause of the problem.

A. *Real-world Example from our Experience*

Many problems were difficult to diagnose. We report here on a bug that required the application of advanced data analysis techniques to be understood.

The bug surfaced in the second configuration of the simulator. We were experimenting with a test generation method that uses an evolutionary algorithm coupled with decision trees [11]. About 30% of the generated tests crashed at the beginning of their execution. No error message was produced to help us. We tried to execute a sample of 1000 tests in the first configuration and did not observe any crash. The bug was thus specific to the second configuration with the middleware and the test tool, but we could not see what was going wrong.

We decided to determine whether the crashing tests shared some common characteristics. All tests were instances of a parametrized cut-in scenario where a target car merges into the lane just in front of the ego vehicle. Only the input parameters varied (the speed of the vehicles, etc.). The use of decision trees in the generation process gave us the idea to also use them for diagnosis purposes. We trained a decision tree with the input parameters of 300 tests that crashed and 700 that did not. We visualized the most relevant branches of the tree and also produced parallel coordinates plots to aid in the analysis. We determined that a discriminating parameter for crashes was the position of the target vehicle relative to the ego at the beginning of the test. We used the simulator to visualize static scenes corresponding to the identified subranges of positions and discovered that the crashes occurred when the target was placed outside of the field of view of ego's lidar. Fig.2 summarizes this line of investigation, starting from a raw system crash report and ending with the identified relation to the lidar's vision.

Once we had understood the relation to lidar data, we could identify the bug. Our tests involved simplified virtual environments with just a flat road, the ego and a target vehicle. Hence, the lidar had strictly nothing to see when the target was outside its field of view. It did not deliver any perception data. The absence of data was not properly handled by the middleware module, which caused the crashes. The middleware was fixed.

B. *Recommendations*

Preventive measures must be taken to avoid the kind of time-consuming debugging we've just described. It is very important to thoroughly validate each new module added to a simulation platform. The unit validation should consider both nominal and robustness cases (like the case of missing data

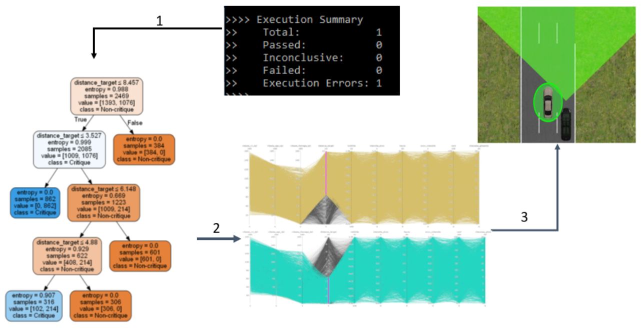

```
>>>> Execution Summary
>>    Total:            1
>>    Passed:           0
>>    Inconclusive:     0
>>    Failed:           0
>>    Execution Errors: 1
```

Figure 2. Investigation of the causes of the crash: 1) decision tree, 2) parallel coordinates, 3) visualization of the field of view.

for the middleware). The integration of the module into the rest of the platform should then be gradual, starting from the implementation of toy experiments in simple configurations, and adding more complexity in several steps.

When bugs surface in complex configurations, the opposite direction should be taken: trying to diminish the complexity while keeping the essence of the problem. In our case, whenever a problem occurred in the second configuration, we tried to reproduce it in the first one that has less modules. We also had other simplification strategies (not illustrated by the middleware example) like simplifying the logic of the tests or replacing the ego by a simple target vehicle.

IV. Insufficient Consideration for Configuration Files

Simulation platforms are highly configurable. The configuration files contain information such as the simulation mode (synchronous, asynchronous), the time step, and the set of modules to include as well as their properties (frequency of execution, priority, etc.). The simulated behavior highly depends on the configuration. Yet, users often pay less attention to configuration data than to code.

A. Real-World Example from our Experience

Since we were working in a R&D context, the simulator was not stable. There were frequent updates to the modules' code (e.g., the physical car module, or the middleware). We also received a new release of the commercial simulation platform. Each time, regression testing was performed by re-running a set of AEB simulation scenarios.

While re-running the scenarios after the new release of the platform, we observed that some scenarios we had passed turned to failures; new collisions were observed. We first thought that this was due to the new platform code. But the code was not the cause. The cause was an undocumented update of the configuration files of the tests: the frequencies of execution of the modules had been lowered. Less frequent sensing and actuating made the task of the AEB more challenging. By aligning the frequencies with the previous ones, we obtained the same pass/fail verdicts as previously.

B. Recommendations

Configuration files are as important as the test scripts and the simulator's version. They should be managed and documented just as code. In the above example, the configuration files were not stored in version control. We had to notice the change and manually roll back the configuration to the previous version. We highly recommend that the test team uses version control for all artifacts needed to reproduce a test experiment.

V. Incorrect Test Setup phase

The test setup phase initializes the simulation by launching the modules and creating the virtual road, the vehicles and so on. It is a tricky phase because there is a lot of instability before everything is properly initialized. Errors during the test setup may induce various effects, from simulations that do not run to ones that run in an incorrect way.

A. Real-World Examples from our Experience

Several reported problems were related to the setup phase.

Example 1 – In the second configuration of the simulator, the simulation could not run unless the modules were initialized in a specific order: first Simulink, second the test tool, and then the rest of the platform.

Example 2 – Also in the second configuration, the first test action could be lost or only partially executed if triggered before the simulator reaches a stable state. As there was no means to determine the stability of the simulator, the solution was to empirically add some warm-up time before the first action.

Example 3 – There were intriguing collision cases where the AEB system did not even try to brake while approaching an obstacle. The cause was not a fault in the AEB logic. It was an incorrect test setup that did not launch the sensors. The AEB could not perceive the obstacle, and so did not react to it. Such useless tests occurred in the first configuration of the simulator but could have occurred in the second one as well.

B. Recommendations

The test setup scripts should be carefully designed and validated before massive experiments are launched. The design should pay attention to the order of the initialization commands, and include systematic checks of their success. Any detected error should stop the initialization process and issue a report. The aim is to avoid as much as possible the silent errors that let the simulation run in an incorrect state, yielding useless results. In cases where the simulation platform does not allow for a synchronization point at a stable initial state, some conservative amount of warm-up time must be provisioned**.** The validation of the setup may then use a validation simulation scenario to verify (i) the error checking logic (e.g., by disabling each module in turn), (ii) the creation of the driving scene, and (ii) the correct execution of the first test action.

VI. Incorrect Teardown Phase

At the end of a run, the teardown phase stops the modules and cleans up the execution environment. Like the startup phase, the teardown may cause problems if not properly defined and implemented.

A. Real-World Examples from our Experience

Example 1 – The batch execution of 1,000 tests was performed in the second configuration of the simulator. As shown in Fig. 3, we observed an increase of the execution time of the tests, the last tests in the suite requiring more than twice the time of the first ones. The bug was in the commercial platform: it provides facilities to clean up the environment, but (as acknowledged in the documentation) sometimes some modules were not closing properly and kept running in the background. We added scripted instructions to systematically kill all modules after each test, which fixed the problem.

Example 2 – A teardown problem caused wrong verdict assignments in a test suite executed in the first configuration.

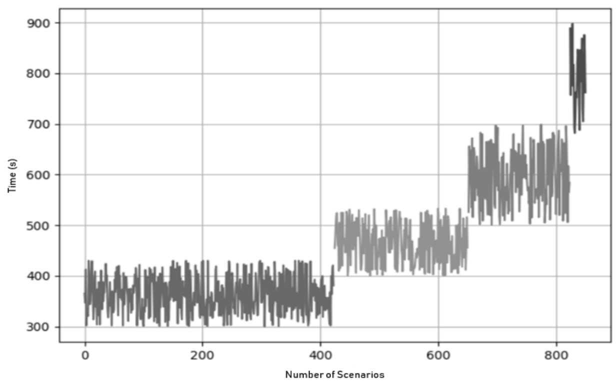


Figure 3. Execution time depending on the order in the test suite.

To spare simulation time, the execution of any test was immediately stopped whenever a collision event occurred. Unfortunately, some of the modules were killed before they could log their most recent data, so that the recorded test trace did not show the collision. The oracle procedure, which was based on the postprocessing of the logged traces, considered the tests as passed. The problem was fixed by changing the stopping script.

B. Recommendations

An incomplete clean-up may cause resource leaks and side effects in a series of runs. To check for the absence of such side effects, a possibility is to repeat a simulation run some number of times and to check that the behavior remains the same. The design of the teardown logic must take care that the current simulation step is fully completed, and that all modules are systematically closed after they are done.

VII. Misunderstanding the Scope of the Simulation

Any simulator has a certain degree of fidelity in the way it reproduces the real-world aspects. There is always a simplification of reality. A good understanding of the scope of the simulation (which aspects are simulated and which are not) is necessary in order to identify what can or cannot be tested in the virtual environments.

A. Real-World Example from our Experience

The issue occurred in our first steps with the simulator. We explored test generation in the case of a parametrized scenario where the ego vehicle is approaching a stationary car. The parameters included the weather parameters offered by the platform, to challenge the AEB system with fog and rain. Surprisingly, we did not find any impact of the weather conditions on the behavior under test.

Actually, the weather conditions were not in the scope of the simulation. The weather module of the platform focuses on camera-based vision. The vehicle under test had a lidar-based vision. While adverse weather conditions also affect lidars [13], the platform does not simulate the disturbance of the perception in this case. Moreover, as other R&D engineers explained to us, the physical car module we used did not simulate how the braking distance is affected by slippery roads.

As a result, the weather parameters could have no impact on the tested behavior. By designing scenarios which included these parameters, we unnecessarily enlarged the input space and generated redundant tests that did not bring any useful outcome.

B. Recommendations

The scope of the simulation may be unclear when reusing off-the-shelf modules. Some facilities seemingly offered by the platform may actually be useless in the specific configuration of a simulator. New users should provision learning time to experiment with the simulator and understand its scope. If it is unclear how the simulator handles an aspect of reality (like the weather conditions in the above example), users should implement a simple parametrized scenario that illustrates this aspect. The parameters should allow the systematic study of gradual situations, starting from a baseline case with no impact (e.g., sunny weather, dry road) to cases where a high impact is expected (e.g., heavy rainfall). It is then possible to perform a sensitivity analysis and observe how the parameter values change (or do not change) the behavior under test.

VIII. Scenarios Having an Unrealistic Physics

The physics of the simulation are not the ones of the real world. From an implementation perspective, this difference means that there are simulator-dependent tricks to avoid generating and running unrealistic scenarios.

A. Real-World Examples from our Experience

Our simulator mixes high-fidelity components (physical car, which simulates the physics of the ego) and low fidelity ones (traffic, which simulates the target vehicles). Each of them introduced simulation-specific effects, which we had to cope with.

Example 1 (Ego) – When trying to initialize ego with a non-zero speed, the obtained behavior is the one shown in Fig. 4. The speed spuriously decreases and increases before it is stabilized at the desired value. For the example in Fig. 4, it takes 500 simulation steps to reach a stable speed of 100 km/h. This is due to the dynamics model in the physical car module: it expects an initialization to a stationary vehicle. Hence, it is not recommended to directly start a scenario with ego in motion in a road and traffic environment. In order to avoid spurious behaviors, one has to proceed as follows. First, the ego is placed on the road with a null speed. Second test action accelerates the ego until the desired speed is achieved (note that the maximal acceleration is constrained by the physical model). And finally, the target vehicles are placed relatively to the ego. In their case, it is possible to have an initial speed. This test procedure typically results in 80% of the simulation time being spent in the initial acceleration of the ego vehicle.

Example 2 (Targets) – The simplified physics of target vehicles proved a serious nuisance for the implementation of automated test generation techniques. We experimented with evolutionary testing to search for critical collision scenarios. The evolutionary search converged towards unrealistic scenarios, like the target vehicle in front of the ego decelerating from 120 km/h to 0 km/h in a very short time. Such scenarios do not bring much insight into the AEB performance. A solution is to better constrain the generation

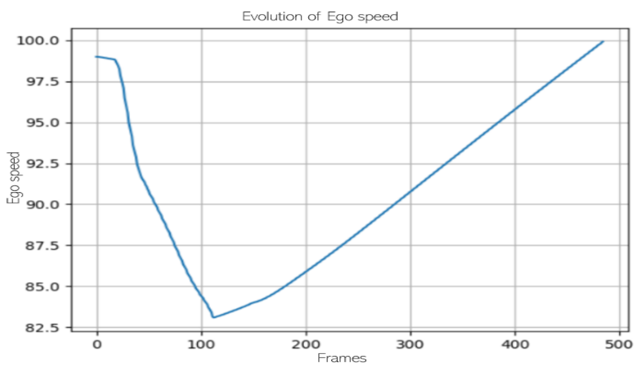


Figure 4. Spurious physical behavior of ego when forced to an initial speed.

process, in order to eliminate some of the unrealistic cases allowed by the simplified physics.

B. *Recommendations*

The physics of the scenarios must be considered when developing virtual tests. High-fidelity physics restrict the test actions that can be applied (e.g., in our case, the state of the ego vehicle could not be directly forced to arbitrary values), which may complicate the implementation of the tests. Conversely, low fidelity vehicle dynamics allows significant freedom for test actions, but at the risk of producing unrealistic scenarios. The test generation algorithms can be constrained for better realism. But this has limitations: it would ultimately require developing complex physical models.

In practice, whenever randomized generation techniques are used, the testers should expect numerous unrealistic scenarios. They should provision effort to determine whether the test results do point to real problems, and should have direct access to experts in the system under test to aid them in the analysis.

IX. Improper Management of Time And Concurrency

For MIL and SIL testing, it is recommended to run the simulation in a synchronous mode. It allows precise control of the test actions performed during a run, e.g., actions to run in parallel or sequentially, triggered at some specific simulation step and for a given duration. However, issues in the management of time and concurrency may severely impair the controllability of the tests. It yields significant differences between the scenarios specified in the test script and the scenarios executed in simulation.

A. *Real-World Examples from our Experience*

All test experiments faced controllability problems. We could only explain them by potential issues in the runtime of the commercial platform. We managed to reproduce many time-related and concurrency-related problems by means of toy experiments simpler than the original tests. In this way, we could confirm the following undesirable behavior:

- Imperfect control of the time-driven triggers. Suppose that the test script requires test action 2 to occur 4s after test action 1. The simulation step is 10ms. During the execution of the test, the action actually occurs at 4.07s, i.e., 7 steps later than prescribed.
- Imperfect control of the duration of test actions. Some test actions, like a lane change, have a parametrized duration. The observed duration is longer than the prescribed one, lasting a few extra simulation steps.
- Lost actions. When a test action lasts several simulation steps, concurrent test actions planned at those steps may be unexpectedly discarded.
- Imprecise positions and speeds. Suppose that at some step, we put a target vehicle 200m ahead of another vehicle. We would expect the position data log at this step to indicate 200m, but instead it indicates a slightly smaller or higher distance. The speed values are also imprecise. Actually, this is due to the sequencing of updates inside a simulation step. For example, the new vehicle is first put at 200m but then the position of the other vehicle is updated according to its speed, which slightly decreases the distance between the two. In the platform we used, the order of updates seemed deterministic but was hard to predict. We could empirically observe that it depended on multiple factors such as: (i) the order in which test actions appear edin the script (T1 || T2 did not yield the same result as T2 || T1), (ii) the priorities of the modules, (iii) the order in which the vehicles were declared.

To illustrate the major impact of these problems on test campaigns, consider an example set of 124 virtual tests that we analyzed. They correspond to various instances of a Cut-out scenario (see Fig. 5). In this scenario, the car in front of the ego (target 1) changes lane to avoid a stopped vehicle ahead (target 2), leaving the AEB a short time to respond to the situation. The cut-out instances involve concurrent test actions to control the relative positions of the vehicles at the beginning of the lane change, their speed, and the abruptness of the change. The tests were run on the first configuration of the platform. From the analysis of the logs, we could determine that none of runs matched the intended tests. For 55 tests (44% of the 124 ones), there was no cut-out at all since the ChangeLane() test action was either lost or incomplete (see Fig. 6.b). For the remaining 69 tests (56%), a lane change occurred. But the relative positions, speeds, start time and duration of the change were not the specified ones (see an example in Fig. 6.a). It is not unfair to say that the accumulation of problems made the cut-out tests uncontrollable.

B. *Recommendations*

For MIL and SIL testing, the controllability of the tests should be a major criterion when selecting and assessing a simulation platform. The user should make sure to understand the offered execution modes (e.g., synchronous or asynchronous, with fixed or varying simulation steps). We strongly recommend that example test scenarios be run in a mode that is supposed to provide full control, in order to check whether the test actions occur at the right time, have the right duration, and provide the expected effect. Some of

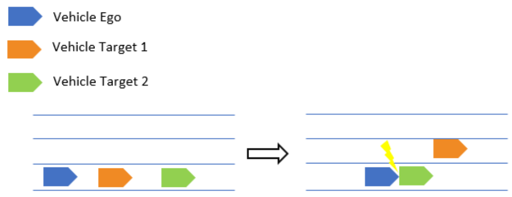
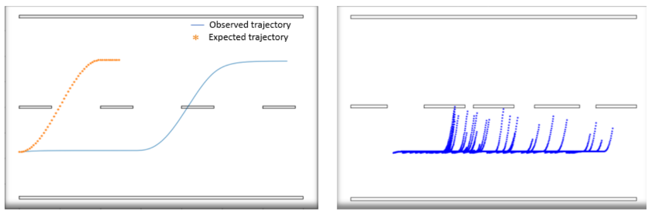


Figure 5. A Cut-out scenario.



(a) A complete lane change occurring too late

(b) Discarded or partially executed lane changes (55 trajectories)

Figure 6. Lane change trajectories of Target 1 in the Cut-out tests.

the scenarios should include concurrent test actions in order to verify their proper management by the platform's runtime.

In the daily usage of the platform as well, it may be wise to develop automated facilities to analyze test logs. The test developers should be able to check that the executed tests correspond to the specified ones, and to assess the deviation. Some imprecision in the control of speed and position values may be unavoidable, but the test should closely reproduce the intended scenarios.

X. Conclusion

This paper has discussed examples of technical issues that arise in the development of virtual tests. Some are due to the difficulty of managing the modular and configurable architecture of the simulator: the *Big bang integration issue*, the *Insufficient consideration for configuration files*, and the *Incorrect test setup and teardown phases*. Others come from having to cope with the physical aspects of the simulation: *Misunderstanding the scope of the simulation*, and *Scenarios having an unrealistic physics*. Finally, the *Improper management of time and concurrency* impairs the ability to control the behavior of dynamic test agents (e.g., adversarial target vehicles) in the environment of the ego.

This list is not exhaustive and practitioners may well experience other issues. For example, our previous work on the virtual testing of an autonomous robot [14] faced a dependency issue: the simulator used open-source technologies and heavily relied on external libraries, the update of which could suddenly break the simulator and the tests. We did not experience such a dependency hell [15] with the commercial platform. But interestingly, two spurious fail cases mentioned in previous work [14] are related to some of the issues we identified for the ADAS tests: one is a teardown bug and the other is a scenario having an unrealistic physics. We believe that the seven issues discussed in this paper are not uncommon and should match the experience of many developers of virtual tests.

The examples in this paper show how those issues can undermine the test activities. We provide recommendations on several aspects like the selection of the simulation platform, the learning phase to get comfortable with a simulator, the validation of the logic of the tests and the management of the configurations. More generally, we would like to insist on the need for establishing engineering practices dedicated to the implementation of the tests. Virtual testing is expected to play an increasing role in the validation of autonomous systems and functions. While a great deal of effort is put in the design of the test approaches, the implementation effort should not be underestimated. By sharing our experience, we hope to contribute to a reflection on the processes, techniques and tools to assist test developers in their tasks.

References

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

[2] Virtual Test Drive [online] http://www.mscsoftware.com/product/virtual-test-drive, accessed: 2021-01-20.

[3] Simcenter Prescan [online] https://www.plm.automation.siemens.com/global/en/products/simulation-test/active-safety-system-simulation.html, accessed: 2021-01-20.

[4] SCANeR Studio [online] https://www.avsimulation.com/scanerstudio/, accessed: 2021-01-20.

[5] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An Open Urban Driving Simulator," in *Proc. 1st Annual Conference on Robot Learning*, pp. 1-16, 2017.

[6] LGVL [online] https://www.lgsvlsimulator.com/, accessed: 2021-01-20.

[7] Y. Li, J. Tao, and F. Wotawa, "Ontology-based test generation for automated and autonomous driving functions," *Information & Software Technology*, Vol. 117, Jan 2020.

[8] F. Hauer, A. Pretschner, and B. Holzmüller, "Fitness functions for testing automated and autonomous driving systems," in *Proc. International Conference on Computer Safety, Reliability, and Security (Safecomp 2019)*, Springer, 2019, pp. 69–84.

[9] G. Li et al., "AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems," *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE),* 2020, pp. 25-36.

[10] M. Koren, S. Alsaif, R. Lee and M. J. Kochenderfer, "Adaptive Stress Testing for Autonomous Vehicles," *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp.1-7.

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

[12] C. Gladisch, T. Heinz, C. Heinzemann, J. Oehlerking, A. von Vietinghoff and T. Pfitzer, "Experience Paper: Search-Based Testing in Automated Driving Control Applications," *2019 IEEE/ACM 34th International Conference on Automated Software Engineering (ASE)*, 2019, pp. 26-37.

[13] Y. Li, P. Duthon, M. Colomb, and J. Ibanez-Guzman, "What happens to a ToF LiDAR in fog?," to appear in *IEEE Trans. on Intelligent Transportation Systems*, DOI: 10.1109/TITS.2020.2998077.

[14] C. Robert, T. Sotiropoulos, H. Waeselynck, J. Guiochet and S. Verhnes, "The virtual lands of oz: testing an agribot in simulation," *Empirical Software Engineering (EMSE)*, vol. 25, no. 3, pp. 2025-2054, May 2020.

[15] Wikipedia page on dependency hell [online], https://en.wikipedia.org/wiki/Dependency_hell, accessed: 2021-05-19.