Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles

Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, Michael von Wenckstern
Software Engineering, RWTH Aachen University, Germany

***Abstract*—Software for self-driving vehicles requires intensive testing to avoid fatal accidents and to allow correct operation in real-world environments. Simulation frameworks are tools that extend component and functional tests to address interconnections between sensors, actuators, and controllers in virtual and predefined environments. Existing simulators can be separated into high-level and low-level ones. Both are designed for very specific scenarios and are not suitable for addressing all driving situations. While high-level simulators are suitable for mastering large testing environments such as cities, they lack fine-grained simulation capabilities, e.g., turning of wheels. In contrast, low-level simulators provide a high level of detail with realistic motion profiles. This is usually only possible in small testing environments. In this paper, we present an approach that combines the benefits of both high-level and low-level simulators to execute component and connector models. Simulation developers can choose the most suitable level of detail and integrate real-world environment data from OpenStreetMap. Moreover, the simulator allows for adaptations and extensions of the physical vehicle configuration including new sensors, actuators and control systems. Another feature of our simulator is its automated testing support and its ability to visualize 3D simulations in a browser. The here presented MontiCAR Simulator framework supports self-driving car developers with a distributed and collaborative development and testing approach as well as with the ability to partially evaluate very specific parts of the simulated car such as sensors or actuators.**

I. Introduction

Autonomous vehicles are complex software and hardware systems that deal with a wide spectrum of safety-relevant functions [1]. It is therefore crucial to perform tests in order to validate these systems. However, performing real-world tests is often unfeasible, since cost-intensive infrastructures and test vehicles are required. Moreover, a malfunctioning during tests might cause dangerous situations. Therefore, tests must be performed in controlled environments. This leads to the importance of simulation frameworks. Here, new software systems for self-driving cars can be safely and efficiently tested in a predefined environment.

The perfect simulation framework should meet a broad set of requirements. First of all, it should combine the features of high-level and low-level simulators. In other words, it should be able to simulate large-scale environments but, at the same time, guarantee a high level of detail. Furthermore, it should allow the detailed simulation of real-world situations, such as a specific area of a given city. It should also support an easy integration and comparison of different sensors, actuators and controllers. In addition, simulators should support automatic unit testing to support repeatable tests and agile development. Meeting all these requirements represents the challenge that we want to tackle in this paper.

A significant number of simulation frameworks already exists. In Section III, we compare the most popular and powerful ones, such as Gazebo, PTV Vissim, SUMO, and CarSim. Each of the investigated simulation frameworks exhibits its particular benefits and drawbacks. For example, Gazebo, which is a highly versatile open-source simulation tool featuring multiple physics engines, does not support environment data imports from OpenStreetMap. Conversely, PTV Vissim can import OpenStreetMap data but does not have a physics engine and is relatively expensive. To our knowledge no simulation framework supports native execution of Component & Connector (C&C) models in realistic city environments. In fact, C&C modeling is possible in MathWorks Simulink with Animation3D. However, no realistic city environment is provided by this solution. What is more, most simulators do not explicitly address automated unit testing support.

The simulation framework introduced in this paper aims to combine all the advantages of the considered frameworks while eliminating their drawbacks. We propose a simulation framework named MontiCAR Simulator that meets the previously highlighted requirements, thereby supporting model-based software engineering (MBSE), test driven development (TDD), evolution, and execution of autonomous vehicle functions. Our framework has its own extensible physics engine and is compatible with OpenStreetMap enabling a realistic simulation of large-scale environments such as cities. At the same time, highly-detailed low-level simulations of vehicle dynamics and accidents are possible. A further feature is the continuous integration and regression testing support allowing to perform unit, integration and acceptance testing.

The remainder of this paper is structured as follows: First, we introduce a running example and requirements on the simulator in Section II in order to demonstrate the capabilities of the developed framework. Second, we compare existing simulation frameworks with our proposed solution in Section III. Third, in Section IV we present our first contribution, namely, the **modular architecture of our simulation framework allowing to exchange and adapt C&C controllers, sensors and actuators in vehicle models as well as maps and environment data in simulation models**. Our second contribution presented in Section V are **automatically executable testing concepts for C&C models such as unit or environment integration tests**. Finally, Section VI concludes this paper.

```
1  simulation Example1 {                                   Sim
2    map = AachenCity.osm;
3    startTime = 22.06.2017 13:30;
4    deltaT = 1ms
5    wheather = noRain, noSnow;
6    duration = 30s;
7    cars {                          Start Position
8      AC-SE001: 50°46'43.7"N 6°03'38.6"E ->
9                50°46'49.7"N 6°04'32.5"E,
10     M-SE003: … -> … }}                    Target Position
```

Fig. 1: Example Simulation Model.

```
1  car AC-SE001 {                                          Car
2    dimension = 4.43m,1.93m,1.25m;
3    visualModel = R8Red.json;
4    weight = 1`655 kg;
5    controller =
6      LaneKeepingController;
7    sensors {                          Controller's input port
8      SpeedSensor => velocity;
9      TiHighAccGPS => position;
10     Compass => direction;     }
11   actuators {                        Controller's output port
12     steering => SteeringFIR4; }}
```

Fig. 2: Example Car Assembling Model.

```
1  component interface Act <T>                             EMA
2  { ports in T wVal,
3          out T aVal;  }
4  component StrgFIR4(Q(0:1)^5 h)
5    implements Act<Z(-45°:45°)> {
6    ocl inv coef: sum(h) == 1;
7    implementation Math {
8      static Z(-45°:45°)^4 d =
9                    zeros(1,4);
10     aVal =wVal*h(1)+ d*h(2:end);
11     d = [wVal d(1:end-1)];
12     aVal = saturate(aVal +
13               normrnd(0, 0.01),
14               -45°, 45°);     } }
```

Act=Actuator,Strg=Steering,w=wished,a=actual,val=value

Fig. 3: Steering actuator.

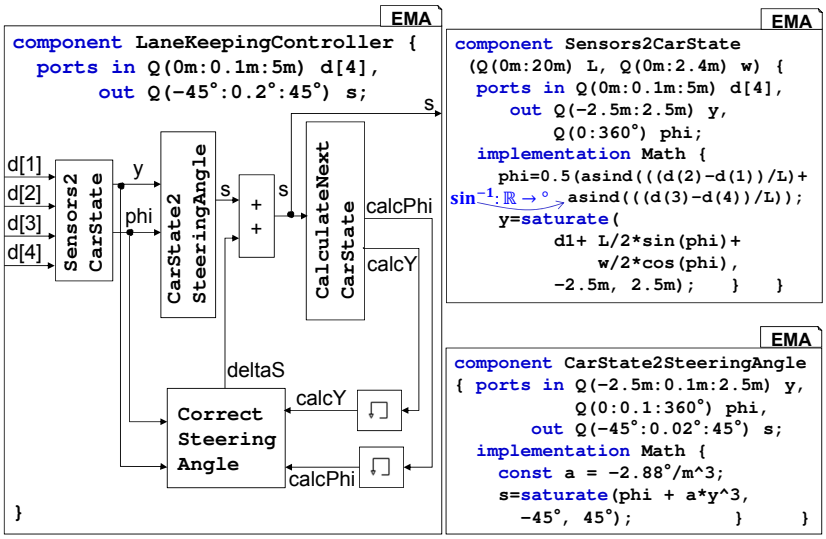

Fig. 4: Overview of C&C Controller models.

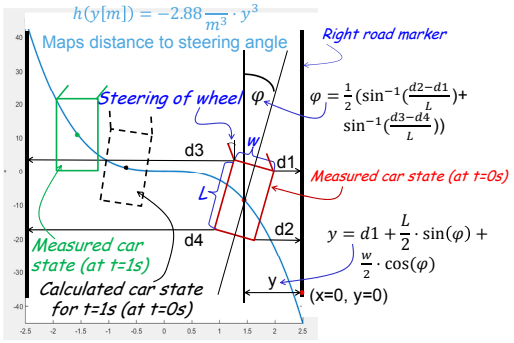

Fig. 5: Explanation for C&C `LaneKeepingController`.

II. Running Example and Requirements

Based on previous research projects with industry partners [19], [17], [3] and three labs on autonomous vehicle modeling, we identified a precise set of requirements for a simulation framework: **(R1)** Import and reuse of existing real world environment data. **(R2)** Capability to simulate large-scale everyday scenarios, e.g., different traffic densities, light and weather conditions. **(R3)** Support for realistic and extensible car models with sensors, controllers and actuators. **(R4)** Multi-platform and portable devices support. **(R5)** Automated support for continuous integration and regression testing. **(R6)** Simulator should contain a physics engine. **(R7)** 3D visualization for demonstration purposes. These requirements must be fulfilled by a versatile autonomous driving simulator in order to execute and validate previously developed C&C vehicles models.

In the remainder of this section, the usage workflow of our proposed simulation framework is demonstrated based on a small realistic example, where two cars follow a straight street; videos showing our simulation framework in action are available at: https://youtu.be/TwMtA6ZV86I, https://youtu.be/d-EEs62-rGM, and https://youtu.be/HiaHf0dd_1w.

(1) First, the map of interest is downloaded from openstreetmap.org; based on this information the simulator accurately recreates streets, buildings and traffic signs. (2) Next, a simulation model as shown in Figure 1 is created. It contains an environment description as well as simulation parameters such as location, time, resolution, weather conditions, and available vehicles. (3) New vehicle models can be created as shown in Figure 2 if needed. Here, one specific car is assembled by assigning its physical and visual properties as well as its controlling unit, sensors and actuators. For each input port of the controller, our framework delivers three default virtual sensors with different noise models. (4) The logic of the self-driving vehicle is modeled as a C&C controller mapping the sensor inputs to actuator commands. Therefore, we use the MontiCAR C&C modeling language family together with its embedded math expression language for behavior definitions [12]. Figure 4 depicts a simple lane keeping control system for straight streets (road curvature = 0 °) inspired by [13]; due to better readability, the body of the `LaneKeepingController` is represented graphically in Figure 5 instead of showing the textual syntax. The input port of the lane keeping control system `d[4]` represents a distance array to the road markers (see Figure 5 for a detailed explanation); the output port `s` represents the desired steering angle of the wheels. The controller contains a simple correction component to automatically adapt steering errors due to imperfect sensor inputs or actuator delays (see difference in *Measured car state (at t=1s)* and *Calculated car state for t=1s (at t=0s)* in Figure 5). (5) Finally, new sensors and actuator models can be created. Figure 3 shows a simple steering actuator modeled in EmbeddedMontiArc, the core language of MontiCAR. In l.12-13 zero-mean normally distributed noise with a variance of 0.01 is added to the manipulated variable

of the actuator. Thereby, we take into account its non-ideal and non-deterministic nature as well as environmental factors such as changing street surfaces.

Using appropriate quality measures such as the mean squared error (MSE) of the car's trajectory, our simulation framework allows for an easy comparison between different C&C controllers.

III. Existing Simulation Environments

Simulation frameworks represent a consistently used tool in the software development and testing process of self-driving vehicles. A wide variety of frameworks already exists on the market. In this section, the most popular ones will be compared in terms of their functionalities and capability to satisfy the requirements presented in section II. In particular, we will focus on Pelops [4], Prescan [23], Mathworks Simulink 3D Animations [16], RTMaps [7], Gazebo [14] [9], SUMO [2] [10], PTV Vissim [5], SimTram [11] [21], Vires VTD Virtual Test Drive [18], the Udacity simulator [20], OpenDaVicni [8] and CarSim [15]. Finally, their differences will be summarized with respect to the requirements.

(R1) Import and reuse of existing real world environment data: Pelops [4], a framework developed by the Forschungsgesellschaft Kraftfahrwesen mbH Aachen, does not present this function as a default feature. The same holds for Mathworks Simulink 3D Animations, RTMaps, Gazebo and the Udacity simulator. Gazebo, which is a widely used simulation framework allows the design of complex maps, but not the import of OpenStreetMap data, which can only be performed by using external modules. Conversely, Prescan, SUMO, PTV Vissim and SimTram provide data import functionality from OpenStreetMap.

(R2) Capability to simulate large-scale everyday scenarios: This is a typical feature of high-level traffic simulators which allows, for instance, to analyze different traffic densities, or different weather conditions. PTV Vissim [5], SUMO, and SimTram are typically used for micro-traffic simulations [21]. Mathworks Simulink 3D Animation, Gazebo, and CarSim offer limited support for such simulations, i.e., they are not optimized for large-scale scenarios while Pelops, RTMaps, and the Udacity simulator do not allow simulations of this kind due to their focus on a single vehicle.

(R3) Support for realistic and extensible car models with sensors, controllers and actuators: Being able to simulate different car, sensor, and controller models is a core feature of our framework. Among the considered simulation frameworks, only SUMO, PTV Vissim and SimTram do not offer this feature due to their pure high-level nature. [14] and [8] display how different models are implemented in Gazebo and OpenDaVinci, respectively. [14] explains the modeling of an all-wheel driver car with GPS and LIDAR whose dynamics is handled by the Open Dynamics Engine. [8] demonstrates how to implement a lane detection application by using the OpenDaVinci framework.

(R4) Multi-platform and portable devices support: CarSim, PTV Vissim and Prescan exclusively support Windows machines. All other frameworks support Linux, as well. Mathworks Simulink 3D Animation supports also a 3D animation web viewer based on HTML5.

TABLE I: Comparison of simulation frameworks, √: yes, P: partially, -: no

Simulator Frameworks	Import and Reuse Environment Data (R1)	Simulate Large Scale Scenarios (R2)	Support for Extendible Car Models (R3)	Multiplatform Support (R4)	Automatic Continuous Integration (R5)	Physic Engine Support (R6)	3D Visualization (R7)
MontiCAR	√	√	√	Win/Lin/ Mac/Web	√	√	√
Pelops	-	-	√	-	-	√	√
Prescan	√	√	√	Win	-	√	√
Mathworks	-	√	√	Win/Lin/ Mac/Web	-	P	√
RT Maps	-	-	√	Win/Lin	-	-	√
Gazebo	-	√	√	Win/Lin	√	√	√
SUMO	√	√	-	Win/Lin	-	-	-
PTV Vissim	√	√	-	Win	-	-	√
SimTram	√	√	-	Win/Lin	-	-	√
Vires VTD	-	√	√	Lin	-	√	√
Udacity	-	-	√	Win/Lin/Mac	-	-	√
OpenDaVicni	-	√	√	Win/Lin	√	-	√
CarSim	-	√	√	Win	√	√	√

(R5) Automated support for continuous integration and regression testing: This requirement is only fulfilled by Gazebo and the OpenDaVinci framework. Gazebo runs a variety of regression tests concerning both the simulation and the physical robot either on an own cluster or in its Hudson regression test suite. OpenDaVinci itself is developed using continuous integration on Jenkins. Furthermore, its modular structure facilitates unit testing of new self-driving vehicle algorithms.

(R6) Physics engine: While Pelops, Prescan, Gazebo and CarSim provide their own physics engines, Mathworks Simulink 3D Animation does not include out-of-the-box physics. However, it allows to import a car modeled in Simulink and to simulate it. The Udacity simulator, RTMaps, SUMO, PTV Vissim, and SimTram do not provide a real physics engine.

(R7) 3D visualization for demonstration purposes: In the context of the considered frameworks, SUMO and SimTram are the only frameworks not supporting a 3D visualization.

Table I summarizes the comparison between all the presented frameworks including ours.

IV. Framework for the Simulation of C&C models in realistic environments

An overview of the proposed MontiCAR Simulator is shown in Figure 6. The main characteristic of this simulation framework is the support for rigid body based physics simulation, computer vision, versatile sensors, control systems, actuators, car models, real environment data as well as continuous integration and regression testing. The framework is based on the discrete time paradigm, i.e., the state of the simulation

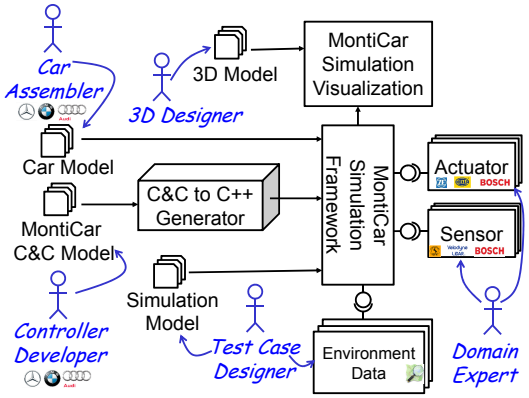

Fig. 6: Simulation Framework and its actors.

```
1  stream StearingAngleTest                    Stream
2    for Sensors2CarState
3    d = [50cm 50cm 2.5m 2.5m]                 (L=3m, w=2m) {
4     tick [50cm 1.3m 2.6m 1.8m];
5    phi = 0° tick 15.45°+/-0.05°;
6    y = - tick -;
7  }
```

Values for input port d
time step
Expected Values for output ports
15.4° ≤ exp. value ≤ 15.5°
Do not care about value

Fig. 7: Stream Test Model.

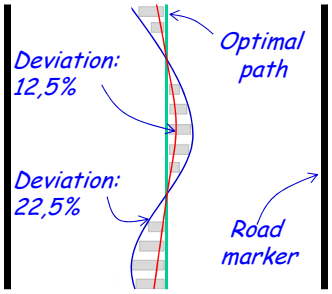

Fig. 8: Deviation of Sensor

is updated in predefined discrete time steps. Furthermore, the main simulator can be coupled with an arbitrary amount of helper simulators in order to add specific capabilities such as vehicle to vehicle communication or tire pressure simulation. Helper simulators can be both discrete time or discrete event based.

The main part contains one or multiple `car models` describing a particular car and one or multiple `MontiCar C&C models` to model control systems. These models are independently developed by the *vehicle engineer* and the *controller developer*. A *test case designer* creates one or multiple `simulation models` describing the simulation. Furthermore, additional `environment data` can be inserted. All sensors and actuators are modeled and integrated by a *domain expert*. Apart from the `MontiCar C&C Model`, each model is directly inserted into the simulation framework. The `MontiCar C&C Model` is translated to multi-threaded C++ code, which is executed to control a vehicle during the simulation.

The visualization part of the simulation framework consists of `3D models`, which are created by a *3D Designer* and used in the visualization of the simulation.

In the following, each element of the simulation framework is explained in detail.

A. Translation of C&C models to C++ code

One main requirement for the MontiCAR C++ Code generator is to be compatible with the widely used Simulink. If we transform Simulink block diagrams into MontiCAR C&C models, the MontiCAR model will produce for the same input values the same output values as the original Simulink model. This allows us to integrate Simulink models in our toolchain.

In the first step, our C++ code generator calculates the execution order of the C&C component instances. This creates a sorted execution list, equivalent to Simulink `slist` command, where only *atomic* components are considered. All other C&C components are similar to the Simulink `virtual` subsystems and are flattened so that only their atomic components are considered. The calculation of the execution order, which is mostly not unique, satisfies the two main rules: (i) If component C1 is connected with component C2, i.e there is at least one connector from any outgoing port of C1 to any ingoing port of C2, than the execution order of C2 must be greater than the execution order of C1. (ii) Components having no input ports (e.g. constant components) can have any execution order as long as they satisfy (i).

The C++ code generator now uses the execution order to invoke the calculation methods of the atomic blocks in the right order. In this context, the input values are pointers of the previously executed calculation method results so that objects do not need to be copied.

In MontiCAR, the behavior of atomic blocks is described via MontiMath, a strongly typed Matlab-like Domain Specific Language (DSL). To enable high-speed matrix operations, the body of the calculation methods invokes the highly optimized Octave [6] C++ implementation using the Intel Math Kernel Library [22].

To tackle the issue that one atomic component A can produce an output x in *km/h* and the other one B wants input values in *m/s*, the C++ code generator converts all input and output values to default SI unit values. In this example x is divided by 3.6 at end of the calculation method before being transferred to B. Thereby, the modeler benefits from SI unit support including generator errors, e.g., when trying to connect *kg* with *m/s*, as well as optimized Basic Linear Algebra Subprograms (BLAS) libraries.

B. Provide Realistic Environment

The simulation of the environment relies upon OpenStreetMap. which is a 2D model. Thus the third dimension has to be obtained from a different source or needs to be sampled as a random variable from an appropriate probability distribution.

In the generated map, for each intersection road signs and traffic lights are randomly generated, as well. Similarly, pedestrians can be included. Their behavior is defined by a set of five parameters which define their movement between two defined points P0 and P1: (1) the distance the pedestrian moves during each time frame (2) the side of the road on which the pedestrian is (3) the direction, either from P0 to P1 or the opposite (4) whether the pedestrian will cross the road or not (5) the instantaneous position of the pedestrian. The weather conditions can be either fixed, constantly or randomly changing.

Vehicles have access to the simulated environment via sensors implementing the `Sensor` interface; available

sensor classes are `WeatherSensor`, `LocationSensor`, `CameraSensor`, `CompassSensor`, `SpeedSensor`, `SteeringAngleSensor`, `DistanceToRightSensor`, etc.

C. Aggregation of C++ Code and Environment in Simulator / PhysicsEngine

The actuator, as shown in Figure 3, can either be written in Java or in MontiMath. Since the `Actuator` interface for MontiMath has one input and one output port of the same type, the Simulator engine generates one large C&C model, which has the controller and all MontiMath actuators as child components. According to the car assembly model, the controller output ports are connected with the actuator inputs. All input ports and unused output ports of the controller are directly connected to the large C&C model, also all output ports of the actuators are connected to the corresponding output ports of the large C&C model. Due to the `Actuator` interface, the large C&C model has always the same interface as the controller. The large C&C model is now translated to C++ Code in order to take advantage of the BLAS operations. Note that the simulator, the sensors and actuators are written in Java.

The generic Java `Sensor` interface contains one method that has two pointers as parameters, one to the complete world object and one to the current car object. This method returns one calculated value. Based on the external simulated environment and objects, the sensor can calculate an output. The generic Java `Actuator` interface contains also one method with a `HashMap` containing all input and output values of the C&C controller, e.g., the engine actuator for the acceleration does not only depend on the target acceleration but also on the current speed. In contrast to the generated C++ Code, where all units are compatible by construction and thus ignored, the Java code for sensors and actuators works with the JScience library. In this way, unit incompatible computations result in Java compile errors.

The simulator (1) executes the sensors models, than (2) it transforms all JScience values to standard SI values and finally it (3) invokes the generated C-Code by JNI. Afterwards, the unitless C++ Code result values of the large C&C model are (4) translated back to JScience values. Subsequently, the (5) Java actuators are executed and, eventually, the (6) physics engine updates all objects (vehicles, pedestrians) in the world. When this process is completed, everything starts from (1) again.

D. Fluent Visualisation of Simulated Data

For systems or acceptance tests, it is helpful to visualize the simulated data. This way, it is possible to figure out missing unit or integration tests, such as "the car should not to leave the road", "the car should not wobble too much", "the car should not park too close in parking slot, otherwise the door cannot open".

Another important reason that justifies the visualization is the test case designer's motivation, i.e., they produce models of better quality when the executed result can be seen. A visualization allows to demonstrate their models' features, to actually see the performance of their automated vehicles, and to better compare different models in order to identify the best one.

The visualisation is realized as follows. The server sends the states of the objects (cars and pedestrians) to the ThreeJS library via web sockets for rendering the world. In order to have a fluent visualization, the JavaScript visualization script interpolates the car states, so that it can fluently drive between two discrete state updates preventing the car from hopping from one place to another. Technically, the visualization consists of two parts, an `init` to create the world and a `loop` part to move the objects.

a) Init.: The simulator delivers all data required to build up the entire map at the beginning, which contains (i) the *terrain* as bounds and height-map data, (ii) the *street* as 4d-array (x,y,z position and street width) of nodes, (iii) the *street signs*, delivered as a json file for the geometry, a png image file as well as a 3d-position point in the map and a 3d vector telling in what direction the sign points, and (iv) the *traffic lights*, handled similarly to the street signs. To make the scenes more realistic, the visualization supports a day and night cycle and three different light sources: ambient light (that globally illuminates all objects in the scene), directional light (that gets emitted in a specific direction), hemisphere light (positioned directly above the scene).

b) Loop.: In the loop, the visualization sets the position of the cars and the pedestrians. In order to have a fluent visualization at 60 fps, which is independent from the steps per seconds that can be calculated by the simulator, the visualization has its own simple interpolation algorithms for positions and directions of all moved objects. This way the simulation also looks fluent even if a position must be retransmitted due to data loss.

Since the simulator is only interested on the friction coefficient of the road, it does not model rain drops to spare computational power on the server. Thus, the JavaScript web client contains a rain controller that creates a specified amount of rain or snow particles (which may have independent sizes) randomly in the world and moves the existing particles around, e.g., by just decrementing the y coordinate.

V. C&C Unit and Integration Testing

Apart from the simulation, the MontiCore Simulator addresses testing purposes in order to test sensors, controllers, actuators, etc. Hence, in the remainder of this section, two testing methods are presented. First, a unit testing method without environment is presented. Second, a C&C integration test method is explained to test controllers with the environment.

A. C&C Unit Testing without Environment

The aim of C&C unit testing, analogue to JUnit testing, is to validate the correctness of single C&C components. We assume that the C&C steering controller (`Sensors2CarState` component (s. l. 2)) should be tested. For this controller, Figure 7 shows a test case for testing

the correct calculation of the steering angle (`phi`) of the `Sensors2CarState` component. In this example, the input values for which the component is tested are defined in ll.3-4. Note that the `tick` represents one time step, e.g., going from $t = 0s$ to $t = 0.1s$. The expected output values for `phi`, i.e., the calculated angle, are defined in l.5. as `15.45°+/-0.05°` meaning that any value between 15.4 °and 15.5 °is accepted. Finally, the expected output values for `y`, i.e., relative y-position of the car are defined in l.6. Since these values are irrelevant for this test, both time steps are defined by a slash.

If a unit test is successful, the result is a list of components used in this test case. Based on this information, a test coverage is computed. If a unit test fails, then the list of failing components is presented. Note that the inner-defined C&C components are regarded, as well.

B. C&C Integration Testing with Environment

Unit testing controllers, sensors, or actuators can also be done in an environment to simulate real world scenarios. For example, consider the example in Figure 8. In this example, the car should follow the straight line autonomously. However, the path computed by the controller is not ideal and also shown in the figure.

To test different controllers, the deviation from the optimal path can be computed for each controller. Based on this deviation, the most optimal controller can be identifed. In this testing scenario, the deviations are totalized and compared. Starting from an ideal position on the line, the environment is initialized (`void init(World w)`). After each simulation time stemp (`void simulateTick(World w)`), the deviation from the ideal position is added to the previous deviation. When the simulation is finished (`void finish(World w)`), the total deviation is computed as $s = \left(\frac{\text{deviation}[m]}{\text{driven distance}[m]}\right)$ and it is checked whether the simulation was successful (`boolean success()`).

VI. Conclusion

Simulating self-driving vehicles is essential but still a highly complex task. Hence, in this paper, we presented a simulation framework that aims to address extension and adaptation concerns, real-world environments as well as real-world sensors and actuators. In particular, it provides an integration of high-level and low-level simulation, which allows to analyze large-scale scenarios as well as to address low-level details, e.g., the steering angle of the wheels. Furthermore, the framework supports integration and unit testing with and without the environment, respectively. This can be used to test sensors, actuators, and controllers. Based on a small scale example, we have demonstrated the applicability of the framework.

Acknowledgements This research was supported by a Grant from the GIF, the German-Israeli Foundation for Scientific Research and Development, and by the Grant SPP1835 from DFG, the German Research Foundation.

References

[1] ISO 26262: Road Vehicles : Functional Safety. ISO (2011)

[2] Behrisch, M., Bieker, L., Erdmann, J., Krajzewicz, D.: Sumo – simulation of urban mobility an overview. Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation (2011)

[3] Bertram, V., Manhart, P., Plotnikov, D., Rumpe, B., Schulze, C., Wenckstern, M.v.: Infrastructure to Use OCL for Runtime Structural Compatibility Checks of Simulink Models. In: Modellierung 2016 Conference. LNI, vol. 254, pp. 109–116. Bonner Köllen Verlag (March 2016), http://www.se-rwth.de/publications/Infrastructure-to-Use-OCL-for-Runtime-Structural-Compatibility-Checks-of-Simulink-Models.pdf

[4] Forschungsgesellschaft Kraftfahrwesen mbH, A.: Pelops, white paper http://www.pelops.de

[5] Gabriel Gome, Adolf May, R.H.: Congested freeway microsimulation model using vissim. Transportation Research Record: Journal of the Transportation Research Board (2004)

[6] Hansen, J.S.: GNU Octave: Beginner's Guide: Become a Proficient Octave User by Learning this High-level Scientific Numerical Tool from the Ground Up. Packt Publishing Ltd (2011)

[7] I. Abuhadrous, F. Nashashibi, C.L.: Multi-sensor data fusion for land vehicle localization using /sup rt/maps. Intelligent Vehicles Symposium. Proceedings (2003)

[8] Ibtissam Karouach, S.I.: Lane detection and following approach in self-driving miniature vehicle. Bachelor of Science Thesis in Software Engineering and Management, University of Gothenburg (2016)

[9] Koenig, N., Howard, A.: Design and use paradigms for gazebo, an open-source multi-robot simulator. In: Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on. vol. 3, pp. 2149–2154. IEEE (2004)

[10] Krajzewicz, D., Erdmann, J., Behrisch, M., Bieker, L.: Recent development and applications of sumo-simulation of urban mobility. International Journal On Advances in Systems and Measurements 5(3&4), 128–138 (2012)

[11] Kumar Abhilash, Sarkar Partha Pratim, S.T.K.: Studying and simulating mix traffic using simtram. Traffic Engineering & Control (2016)

[12] Kusmenko, E., Roth, A., Rumpe, B., von Wenckstern, M.: Modeling architectures of cyber-physical systems. In: 13th European Conference on Modelling Foundations and Applications. Springer (2017)

[13] Oliver Toro, Tamas Becsi, S.A.: Design of a lane keeping algorithm of autonomous vehicle. Periodica Polytechnica Transportation Engineering (2015)

[14] Özgüner, Ü., Redmill, K., Biddlestone, S., Hsieh, M.F., Yazici, A., Charles, T.: Simulation and testing environments for the darpa urban challenge. IEEE International Conference on Vehicular Electronics and Safety (2008)

[15] Patil, K.S., Jagtap, V., Jadhav, S., Bhosale, A., Kedar, B.: Generating a 3d simulationof a car accident from a written description in natural language: the carsim system. Proceedings of the workshop on Temporal and spatial information processing (2001)

[16] Patil, K.S., Jagtap, V., Jadhav, S., Bhosale, A., Kedar, B.: Performance evaluation of active suspension for passenger cars using matlab. IOSR Journal of Mechanical and Civil Engineering (2013)

[17] Richenhagen, J., Rumpe, B., Schloßer, A., Schulze, C., Thissen, K., von Wenckstern, M.: Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In: International Systems and Software Product Line Conference (SPLC '16). pp. 174–183. ACM (2016)

[18] Roth, E., Dirndorfer, T.J., Knoll, A., v. Neumann-Cosel, K., Ganslmeier, T., Kern, A., Fischer, M.O.: Analysis and validation of perception sensor models in an integrated vehicle and environment simulation. Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation (2011)

[19] Rumpe, B., Schulze, C., Wenckstern, M.v., Ringert, J.O., Manhart, P.: Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In: Software Product Line Conference (SPLC'15). pp. 141–150. ACM (2015), http://www.se-rwth.de/publications/Behavioral-Compatibility-of-Simulink-Models-for-Product-Line-Maintenance-and-Evolution.pdf

[20] Udacity: Self-driving car simulator. GitHub (2016)

[21] Viral Patel, Manish Chaturvedi, S.S.: Comparison of sumo and simtram for indian traffic scenario representation. Transportation Research Procedia (2016)

[22] Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library. In: High-Performance Computing on the Intel® Xeon Phi™, pp. 167–188. Springer (2014)

[23] Zhizhou Wu, Jie Yang, L.H.: Study on the collision avoidance strategy at unsignalized intersection based on prescan simulation. 13th COTA International Conference of Transportation Professional (2013)