Scenario Definition for Prototyping Cooperative Advanced Driver Assistance Systems

K. Massow, F. M. Thiele, K. Schrab, B. S. Bunk, I. Tschinibaew, and I. Radusch

***Abstract*—Today's Advanced Driver Assistance Systems (ADAS) adopt an autonomous approach with all instrumentation and intelligence on board of one vehicle. In order to further enhance their benefit, ADAS need to cooperate in the future. This enables, for instance, to solve hazardous situations by coordinated maneuvers for safety intervention on multiple vehicles at the same point in time. Our prototyping environment presented in previous work addresses developing such cooperative ADAS. Its underlying approach is to either bring ideas for cooperative ADAS through the prototyping stage towards plausible candidates for further development, or to discard them as quickly as possible. This is enabled by an iterative process of refining and assessment. In this paper, we focus on handling the application specific parameter space, and more precisely on the scenario related aspects. As a part of our iterative prototyping process, defining and tuning scenarios and application parameters are highly repetitive tasks which needs to be designed very efficiently. We, therefore, strive to create a scenario definition methodology, which provides best flexibility and a minimal expenditure of time on the developer side.**

I. Introduction

Advanced Driver Assistance Systems (ADAS) are integrated functions of road vehicles, designed to support the driving process. Today's ADAS are realized through an autonomous approach with all instrumentation and intelligence on board of one vehicle. However, in order to assemble more of these functions to reach fully autonomous driving in a complex road network, very expensive sensors and complex machine intelligence are required [1]. Thus, to further enhance the area of application for ADAS with reasonable implementation effort for sensors and intelligence, ADAS need to cooperate in the future [2]. Such cooperative ADAS will be enabled by communication between ADAS deployed on different vehicles and on road infrastructures. For this purpose, V2X communication [2] is used, e.g. by Cooperative Adaptive Cruise Control (CACC) [3] to share information in a vehicle platoon aiming driving efficiency.

ADAS may directly intervene into vehicle control. Consequently, design and implementation of ADAS is highly safety-critical and comprehensive evaluation methodologies [14] are of vital importance for the ADAS development process. The development of cooperative ADAS, however, requires new evaluation methods. Due to the complexity of the addressed traffic scenarios, employing real world vehicles would require a tremendous effort. Thus, especially for the early phases of prototyping, simulations will become increasingly important. In the first stage of prototyping, an idea e.g. of solving a hazardous situation by a new cooperative ADAS needs to prove its feasibility. The cooperative aspect makes the number of parameters to be considered during prototyping very large. Developers need to handle this large parameter space, which includes finding and tuning parameters in a time-consuming trial and error manner, in order to come to such a verdict about feasibility.

The prototyping environment presented in this paper is designed to support developers in the first stage of prototyping, when an idea for a new cooperative ADAS is tested for feasibility. For this purpose, our prototyping environment dedicatedly supports handling large parameter spaces inherent in cooperative ADAS. This parameter space refers to two aspects: vehicle dynamics related parameters and application specific parameters. The parameter space explodes, if each vehicle involved in a cooperative maneuver, needs to consider these parameters for each of the other vehicles involved. Thus, the overall number of parameters to be considered grows superlinear with the number of vehicles involved.

Our prototyping environment tackles the large parameter space regarding vehicle dynamics by a tradeoff between the number of vehicles to be simulated at the same time and the precision of mapping physics realistically below the limits of driving dynamics. In order to handle the exploding application specific parameters space, we propose a process to bring an idea of a new cooperative ADAS through an iterative process of refining and assessment towards a plausible candidate for further development. Aligned with this process, developers can use our prototyping environment in a trial and error manner to create, refine, and assess. In this way, the candidate cooperative ADAS can either be brought incrementally through the stage of prototyping or be discarded as quickly as possible. We designed our prototyping environment to support this process.

We initially presented our prototyping environment in [14] aiming on a comprehensive presentation and with a special focus handling the parameter space regarding vehicle dynamics. In this paper, we focus on handling the application specific parameter space, and more precisely on the scenario related aspects. As a part of our iterative prototyping process, scenario definition and application parameter tuning are highly repetitive tasks which needs to be designed very efficiently. We, therefore, strive to create a scenario definition methodology, which provides best flexibility and a minimal expenditure of time on the developer side. As a consequence, our key design decision is to assume knowledge of Kotlin [15] programming language by developers of cooperative ADAS using our prototyping environment. We enable the named flexibility by providing a lean domain-specific language (DSL) based programming interface, instead of a complex GUI.

K. Massow, F. M. Thiele, and B. S. Bunk are with the Technical University Berlin / Daimler Center for Automotive IT Innovations, Berlin, Germany (e-mail: kay.massow, max.thiele, sebastian.bunk@dcaiti.com).

K. Schrab, I. Tschinibaew, and I. Radusch are with Fraunhofer Institute FOKUS Berlin, Berlin, Germany (e-mail: karl.schrab, iskander.tschnibaew, ilja.radusch@fokus.fraunhofer.de).

The rest of this paper is organized as follows. In Section II, we derive the scope for our prototyping environment, which is the basis for its requirements presented in Section III. We give a brief overview on how all requirements are derived and put a focus on the requirements related to the scenario. We present the design of the overall architecture with a detailed view on the scenario related part in Section IV. Section V briefly discusses related work. In Section VI we give an outlook on current and future work and on our ambitions to go open source, while Section VII concludes this paper.

II. Scope

With the scope of our prototyping environment we define the class of applications to be addressed, its constraints on driving dynamics, and the goals of prototyping. As indicated by Fig. 1, ADAS can be characterized by their level of automation [4] and cooperation [2]. In order to elaborate this characterization, Fig. 1 orders different ADAS according to their specific degree of cooperation and automation [5]. The prototyping environment described in this work is more useful to prototype ADAS, which require a high degree of both dimensions equally. ADAS of this kind are arranged close to the diagonal in the figure. Thus, the scope of the prototype environment, indicated by the arrow in the figure, covers the area around the diagonal and grows with the degree dimensions from the bottom left to the top right.

In addition to perceiving the vehicle's environment and driving conditions, it might also be necessary to be able to control the vehicle at its dynamic limits, e.g. to enable accident avoidance maneuvers. Mapping driving dynamics in simulations in a physically realistic way at or beyond the limits of driving dynamics, requires complex and highly nonlinear simulation models. Such models grow in complexity and computational demand with their precision at the limits of driving dynamics. This characteristic makes vehicle dynamics simulation to be a conflicting requirement with simulating multiple vehicles at the same time. Thus, we need a tradeoff between the number of vehicles to be simulated at the same time and the precision of mapping physics realistically.

We address this tradeoff by restricting the scope of our prototyping environment to the class of applications which are meant to prevent accidents, not to mitigate them. Therefore, we can assume a certain safety margin which should always be regarded by the applications. That excludes e.g. very close drive-by maneuvers at high speeds or with very high acceleration forces. For our simulator it is thus sufficient to map a smaller range of dynamics, as its purpose is to match real vehicle dynamics up to the limits of driving dynamics only. Based upon this definition, we can now define a set of reference applications for our simulator to cover all aspects of the class of cooperative ADAS within our scope. We use these reference applications to derive the requirements of the simulator. The set of selected reference applications and the reasoning of selection is described in detail in [14]. In summary, this leads to the following three applications.

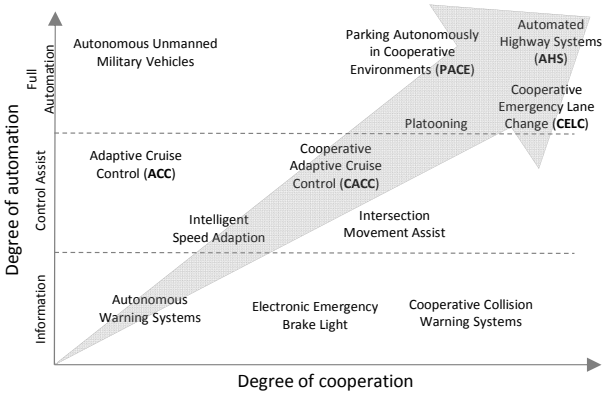

Figure 1: Usefulness of the prototyping environment for developing application, depending on their level of automation and cooperation

- CELC – Cooperative Emergency Lane Change
- CACC – Cooperative Adaptive Cruise Control
- PACE – Parking Autonomously in Cooperative Environments

III. Requirements

Controlling a vehicle by an ADAS requires consideration of vehicle dynamics including a great number of parameters related to vehicles and environment [6]. Additionally, an ADAS deployed on a vehicle needs to regard many application specific parameters, like those related to situation awareness [7]. For cooperative ADAS deployed on multiple vehicles (such as CELC, CACC, and PACE), the number of parameters to be considered by the simulator might grow superlinear. Each vehicle might also need to consider the relevant parameters of multiple other vehicles, they need to interact with. These two aspects, parameters regarding vehicles dynamics and application specific parameters, in combination with many vehicle make the parameter space explode. Towards these two aspects, we can now define the requirements for our simulator, described on the basis of the three reference applications (ELC, CACC, and PACE).

We begin by identifying the originators of the requirements. These are the cooperative ADAS *application* to be deployed in our simulator, the *developer* using our simulator, and the *scope* of our simulator. The first three main entities of our simulator defined as clusters of requirements are derived from the DVE model [8], which models the loop of *driver*, *vehicle*, and *environment* while driving that will be observed by the *application*. These three are complemented with the *communication*, which addresses the cooperation aspect, and finally the *architecture* of the simulator which needs to fit our *scope*. In Table 1 the originators are arranged at the column headers and the requirement clusters at the row headers. The table cells describe the concrete requirement of an originator to a requirement cluster. For this work, we focus on the requirements related to the scenario definition, while a detailed description of all requirements is given in [14].

1) Vehicle

Sensors and actuators are needed to enable the applications to interact with the vehicle and sense its environment. Modelling sensors can be extremely complex and computational expensive. However, within our scope the following set of fundamental sensors using simple models with few parameters should be sufficient for the majority of cooperative ADAS applications, while offering developers the possibility to hook up further custom sensors.

- Frontal sensor like radar [1], [3] as needed by CACC;
- Side sensors like a blind spot detection system [1] as needed by CELC;
- Lane detection sensor like a stereo camera [1] as needed by PACE for precise navigation.

Parametrization - of the vehicle dynamic and sensor model need to be adaptable by developers. Changing these parameters during prototyping cooperative ADAS is a highly recurrent, iterative process and needs to be designed in a way to enable very short cycles. For this purpose, at this point we identify the definition of such an *iterative prototyping process* as an additional separate requirement.

2) Environment

The environment of the vehicles in our simulator needs to contain objects that can be perceived by sensors and infrastructural elements, the vehicles can interact with. This includes static and moving objects like houses and street furniture (e.g. traffic lights) as well as obstacles like road users as e.g. needed by CELC. For the majority of research done on cooperative ADAS within our scope, the following environmental features should be sufficient:

- road infrastructure generated procedurally from simple multi-lane road segments, as well as complex street grids including mapping of traffic rules;
- properties of objects and infrastructure influencing vehicle dynamics and perception need to be parametrable, e.g. road surface or weather conditions;
- location and time bound triggers are needed, e.g. to move the obstacle in front of the vehicle very closely;
- scriptable procedures are required, e.g. like braking events, cutting in, or objects entering the road.

Generating and iteratively modifying scenarios including complex road infrastructures as described, as well as its parametrization and scripting is a time-consuming job for developers. Thus, at this point we identify the need for a *scenario definition* as an additional, separate requirement, which enables rapid prototyping in a very time effective way.

3) Driver

A driver model moving vehicles in our simulator is needed first as input for a cooperative ADAS application deployed on this vehicle, and second to generate surrounding traffic for sensor perception. In order to generate driver behavior, the basic tool fitting our scope are speed annotated routes defined by developers plus a set of basic driver behaviors and related features should be provided by our simulator:

- microscopic behaviors [9] regarding traffic, e.g. stopping in front of red traffic lights, giving way, regarding speed limits, and avoiding collisions;
- scripted maneuvers defined by developers as part of the *scenario definition*, such as speed changes and braking maneuvers that can be triggered by the environment, as required by CACC;
- parameterization of the maneuvers that need to be varied while prototyping, such as the driver reaction time and randomized behavior (e.g. swaying in the lane);
- reproducible random seeds to guarantee repeatability.

4) Communication

Similar to the approach of modeling sensors, our simulator should provide models for V2X and cellular communication [10] allowing for easy parametrization of at least the following parameters: delay, packet lost, range, and bandwidth. Again, we provide the ability to hook up further custom models.

5) Architecture

The most important requirement on the architecture of our simulator is to maximize the processing power available for simulation. This can be achieved by designing the architecture to be able to scale the simulation over multiple instances running on different machines. From the perspective of developers, the architecture should further enable:

- visualization of the running simulation;
- provide an open interface to hook up the applications instead of enforcing a certain technology to this end, such as MATLAB®/Simulink®;
- support the iterative prototyping process mentioned as separate requirement and in this context a time effective parameterization of the simulation;

6) Scenario Definition

The scenario definition must contain all relevant parameters of the simulation model (vehicle, sensors, communication), the parameter setting of the applications, as well as the definition of the environment and the definition of driver input for a simulation. Thus, the scenario should contain all information developers need to specify and vary with regard to the named iterative prototyping process.

7) Prototyping Process

Our prototyping environment should support a process to enable developers handling the application-specific part of the related parameter space. Designing our simulator should be aligned with this process which brings the idea of a cooperative ADAS through an iterative process of refining and assessment towards a plausible candidate for implementation.

Table 1: Requirements

	Application	**Developer**	**Scope**
Vehicle	Actuators, sensors	parameterize models and sensors, validation	Tradeoff between precision and computational effort
Environment	Perception by sensors	Scenario definition (time efficiently)	Simple models to address computational effort
Driver	Interaction by actuators	Define behavior	-
Communication	Short range V2X	Parameterize	Simple models to address computational effort
Architecture	Deployment	Open interfaces, iterative prototyping process, visualization, time, repeatability	Distribution to increase performance

IV. The Simulator

In this section, we present the design of the different aspects of the simulator according to the requirements defined in previous section. In the context of this paper, we focus on the key design aspects regarding the scenario definition and related aspects. A detailed description of the overall design is given in [14]. Prior to designing the simulator, we first need to define its underlying prototyping process, since all other aspects are aligned with this process.

A. Prototyping Process

We propose the following process depicted in Fig. 2 to bring an idea of a cooperative ADAS through an iterative process of refining and assessment towards a plausible candidate for further development. Aligned with this process, developers can use our simulator in a trial and error manner to create, refine, and assess their use case. In this way, new approaches can either be brought incrementally to a certain stage of maturity or be discarded as quickly as possible.

Beginning with the implementation of the cooperative ADAS prototype, developers define a set of working parameters of the application. An initial set of these parameters needs to be given by developers in the first step. The same applies to the simulation scenario and its parameters. Subsequently, the process iteratively traverses an arbitrary number of cycles including the three steps, running the simulations, assessing its results, and tuning the parameters of scenario and application. This cycle is completed once the results suggest that the cooperative ADAS application under research is realizable and effective. Otherwise, the cycle is to be terminated after a significant number of iterations without any progress on the expected results, so the idea needs to be either reconsidered or discarded.

In order to reach either of both verdicts as fast as possible, the cycle time needs to be minimized. Accordingly, the goal of the simulator design is to minimize the execution time of one cycle. The execution time of one cycle depends significantly on the preferably low complexity of the simulation models (vehicle, environment, communication) and the scalability of the simulator architecture. Tuning the parameters of application and the scenario by developers is highly depending on a pragmatic scenario description of the simulator. These aspects will be object of the following sub sections.

B. Architecture

Fig. 3 depicts a high-level view of the architecture of our simulator consisting of two parts, the *Developer Implementation* containing the components developers need to define and implement, and the *Simulation Framework*. The latter contains multiple *Simulation Instances* which can be distributed over multiple machines, and the *Simulation Main Instance*, which is coupled to the *Visualizer*. This architecture and its components are designed towards the following three underlying paradigms derived from our requirements.

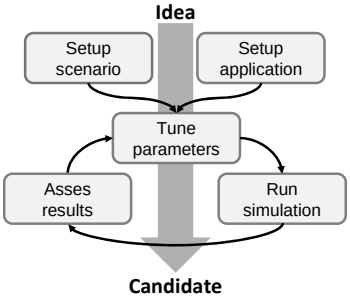

Figure 3: Iterative prototyping process

Decoupling Simulation in Time – our architecture decouples simulation time from real time. Its simulation models support both, defining fixed simulation speed, as well as running headless as fast as possible regarding a predefined fixed simulation step size.

Decoupling Simulation in Space - refers to the idea to scale out the simulation by splitting the simulation scenario area in different cohesive sub areas. This enable distributing the execution of simulation models on different *simulation instances* (see Fig. 3) i.e. on multiple machines.

Decoupling Simulation in Complexity - refers to designing our architecture to run several parts (such as sensor models) of the simulation remotely (e.g. using MATLAB®/Simulink®) and allow custom implementation of these parts by developers by providing open interfaces.

C. Scenario Key Design Aspects

Before describing concept and implementation of our scenario model, we briefly present its underlying design decisions. We address the requirements of environment, driver, and scenario specification regarding simulation performance and developer's interaction by the following key aspects:

1) Environment

All objects other than vehicles are modelled by simple geometrics (polygonal planes, cuboids, and tetrahedrons) to safe computational effort and human effort for design. Coping without textures to the greatest extent, our visualization still allows for an appealing puristic scene rendering using simple Phong shading techniques (see Fig. 4). Static objects are procedurally generated from existing map material. The generated infrastructure can be modified and complemented by developers. Event triggers and scripted procedures related to the environment e.g. time bound occurrence of road geometry changes (see e.g. constructions site in Fig. 4).

2) Driver Input

The definition of driver behavior is based on speed annotated routes along the road map links (see Fig. 4). In

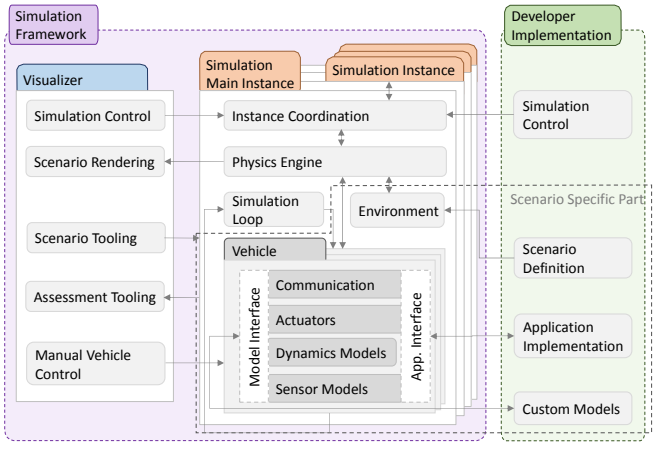

Figure 2: Architecture

addition to the target speed, developers can attach time or event triggered, scripted maneuvers to these routes to create specific situations. The driver behavior in traffic is automatically realized by an extendable driving controller hierarchy of various speed and steering controllers.

3) Scenario Definition

The description of a simulation scenario contains all scenario specific information about environment, driver input, simulation models and their parameterization. For all position related elements of a scenario (e.g. placing objects, vehicles, defining routes), our simulator provides visual tool support. All scalar elements (e.g. parameterization of vehicle models, sensors, and randomized behavior on routes like swaying or driver reaction time) are defined in a set of configuration files. The *scenario definition* bundles all that information and distributes it to all *simulation instances* by the *instance coordination* (see Fig. 4). Optionally a seed set for all randomized parameters of the simulation models is included.

Remark: *As parts of our iterative prototyping process, scenario (re)definition and application parameter tuning are highly repetitive tasks which needs to be designed very efficiently. We therefore, strive to create a scenario definition methodology, which provides best flexibility and a minimal expenditure of time on the developer side. As a consequence, our key design decision is to assume that developers of cooperative ADAS will use our prototyping environment to master the Kotlin [15] programming language. Instead of complex GUI based interfaces, we provide a lean DSL based programming interface, which enables the named flexibility.*

Examples of scenario created for our simulator can be found in [11] and [14]. Fig. 4 gives an impression of an example scenario displayed by the *visualizer* (see Fig. 3). The scenario near Ernst-Reuter-Platz in Berlin, Germany, was generated procedurally by the simulator from Open Street Map (OSM) [12]. A route defined by the developer is depicted as a blue line. In this example the route has no annotated speeds as the target speed is taken from the imported OSM map speed limit.

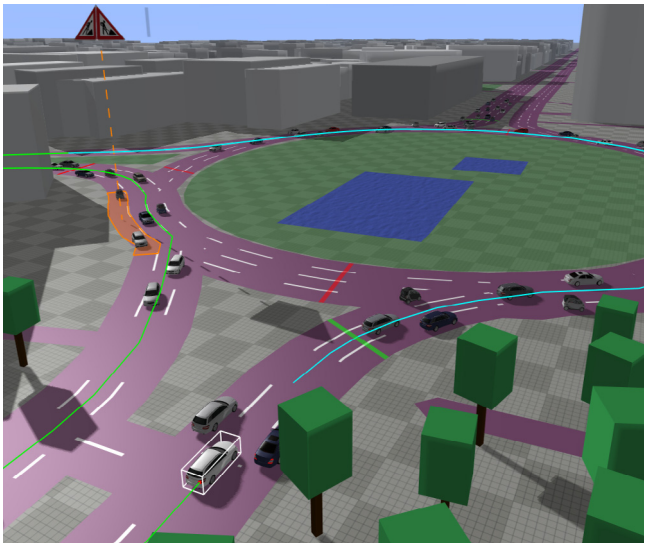

Figure 4: Simulator reference implementation – example scenario: Construction site at round about, Ernst-Reuter-Platz, Berlin, Germany

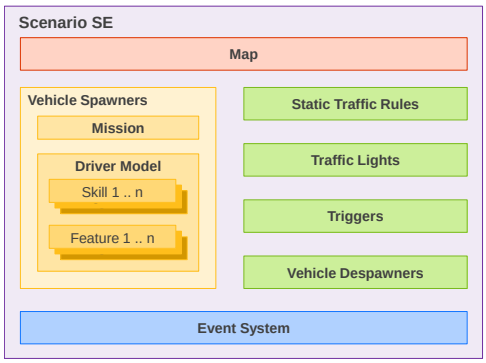

Figure 5: Scenario Model

D. Scenario Model

The scenario model is depicted in Fig. 5. It provides a more detailed view on the *Scenario Specific Part* outlined in the general architecture in Fig. 3. It contains the *HD map*, which provides the basic road geometry of each scenario and is generated procedurally from OSM or imported from HD map sources such as HD Live Map [19] or Lanelet. *Routes*, *Triggers* (such as the construction site in Fig. 4) and *Auxiliary Objects* are geo spatially referenced to the map and connected via the *Event System* to the *Vehicles* and their specific *Driver Models* in the scenario. *Vehicle Spawners* and *Vehicle Despawners* deploy and remove vehicles from the scenario at runtime, according to the scenario specification. The vehicles contain specific sensors and driver behavior realized by *Skills* and *Features*, which are described in the subsequent subsection. *Skills* also realize traffic rule related behavior.

E. Driver Model

1) Motivation of Skills and Features

In order to address handling the complexity of the large, application related parameter space, we designed a driver model composed of reusable behavior entities, inspired by the subsumption architecture [16]. Our driver model is designed by a flexible and scenario specific composition of *Skills*. A *Skill* is the implementation of a preferably reusable behavior entity, that is either atomic or a combination of other atomic *Skills*. Atomic *Skills* provide actions, which cannot be broken further down, as, e.g., lane changing (provided by the lane-change-skill). Composed *Skills* provide more complex actions,

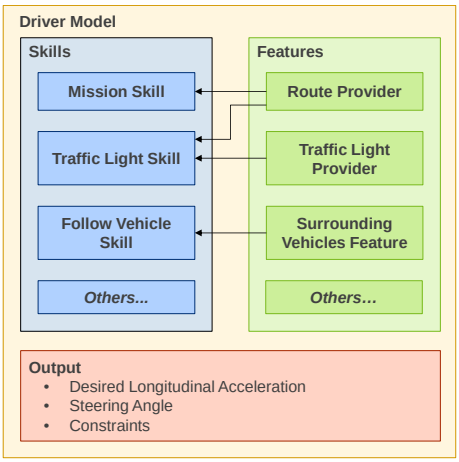

Figure 6: Driver Model – a composition of Skills and Features (illustrated components required to realize the scenario of Fig. 4)

which can be achieved by combining multiple atomic *Skills*, as e.g., the Intersection-Skill which combines the Traffic-Light-Skill and the general Traffic-Rule-Skill. Atomic *Skills* command their requested actuator control to the vehicle by desired acceleration and steering angle. For the realization of such behavior entities, *Features* are used by the *Skills* in order to interact with the environment. *Features* encapsulate functionalities required for this purpose. *Skills* and *Features* are designed fully decoupled and work in a self-orchestrated manner. Thus, at design time of a *Skill*, there is almost no need to consider other existing *Skills* or *Skills* that will be designed afterwards. In accordance with our iterative prototyping process, this enables best flexibility and reduced complexity for designing complicated, multilayered driving behaviors.

We illustrate this concept in the following, based on the example of braking at a red traffic light, depicted in Fig. 6. For the sake of conciseness, we consider a very simple orchestration of three *Skills* which determine the longitudinal acceleration of a vehicle in a self-orchestrated manner. The regular driving velocity is determined by the *Mission-Skill*, which makes a vehicle traverse its route, according to the speed limits. In case the vehicle is following a slower predecessor, the regular driving velocity is overridden by the *Follow-Vehicle-Skill*. At red traffic lights, both are overridden by the *Traffic-Light-Skill*. Braking at red traffic lights is a fundamental element of all urban driving models. It includes recognition of the traffic light, deriving a decision whether to brake and performing a brake if required. The recognition is realized by the *Red-Light-Feature*, which continuously checks the map in front of a vehicle for traffic lights. Their current phase is then perceived by the *Feature* either checking the *Auxiliary Object – Traffic Light* directly in the scenario, or by evaluating either a specific camera sensor model, or a communication device. Accordingly, the *Red-Light-Feature* reports a red light and the *Traffic-Light-Skill* calculates a desired deceleration regarding the remaining distance.

2) Priorities and Constraints:

In order to achieve the described decoupling of Skills and to enable their self-orchestrated coordination, we introduce the concept of priorities and constraints to control lateral and longitudinal actions. Priorities are used to decide which action is rendered, in case multiple skills command conflicting actions. Priorities are assigned to skills at skill design time and then remain fixed. In order to simplify the decision about what priority to assign to a certain skill, available priorities reflect the various importance levels in the driving context: Listing 1. Fixed skill-priorities are sufficient to orchestrate the various skill actions in simple situations. However, in certain circumstances, lowly prioritized skills might need to increase their priority to avoid rule violations or overcome potentially dangerous situations. This is achieved by constraints. In addition to their regular desired action skills can specify constraints with higher priorities. An example is the mission-skill: The desired speed output of that skill is of priority *DRIVING_FREE*, which is very low, so that other skills, as, e.g., the follow-vehicle-skill can override that output. However, vehicles should not exceed the speed limit. Accordingly, the current speed limit is set as a constraint with priority *RULE_SPEED_LIMIT*. Moreover, the mission-skill knows the upcoming course of the road and can compute lateral accelerations resulting from vehicle speed and road

```
public enum Priority {
    NONE(0.0),
    DRIVING_FREE(1.0),
    DRIVING_IN_TRAFFIC(1.1),
    RULE_DEFAULT(2.0),
    RULE_TRAFFIC_SIGN(2.1),
    RULE_TRAFFIC_LIGHT(2.2),
    RULE_SPEED_LIMIT(2.3),
    RULE_EMERGENCY(2.4),
    AVOID_COLLISION(3.0),
    AVOID_COLLISION_SEVERE(3.1),
    FORCED(4.0);

    private final double prio;
    Priority(double prio) {
        this.prio = prio;
    }
    public boolean isHigherThan(Priority other) {
        return prio > other.prio;
    }
    public boolean isHigherOrEqualThan(Priority other) {
        return prio >= other.prio;
    }
}
```

Listing 1: Available skill priorities

curvature. Thus, a second constraint is set with priority *AVOID_COLLISION*, which limits the speed to a value which ensures safe vehicle handling.

Fig. 7 shows the outputs of the three aforementioned skills in a traffic light situation. For simplicity, the desired speed of each skill is shown, not its actual demand of acceleration. The ego vehicle follows a vehicle while approaching a traffic light. The leading vehicle accelerates in order to pass the traffic light. The Follow-Vehicle-Skill wants to accelerate to close the growing gap to the leading vehicle. The result speed is first limited by the speed limit and then reduced to standstill by the Traffic-Light-Skill when the traffic light turned red. After turning green the vehicle continues its way with reduced speed in order to keep a pleasant speed within the roundabout.

F. Scenario Definition

In the following we present our methodology to define scenarios using a Kotlin-based scenario domain-specific language (DSL). An example for a simple scenario definition is given in Listing 2. Note that this is no pseudocode but a valid Kotlin program (without the import declarations) that will run a simulation. Scenarios are defined within the *scenario {}* block: An HD map is loaded and a traffic light is generated for a certain intersection. The traffic light generation function takes the nearest intersection for the given location and generates default traffic lights for that intersection. Alternatively, manual definition of the layout and timing of arbitrary traffic lights is also possible. Next, a single *RouteSpawner* is defined, which spawns ten vehicles on the first lane at the beginning of the route. The route is computed by a standard A* routing between

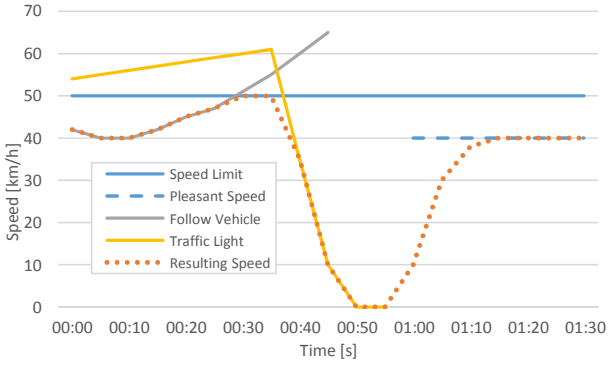

Figure 7: Priority graph – longitudinal control near traffic lights

```
fun main() {
    scenario {
        hdMap("maps/ernst-reuter-platz_large.lanelet")
        generateTrafficLight(
            AbsoluteLocation(52.510718, 13.314574))

        routeSpawner {
            route = route(
                AbsoluteLocation(52.509964, 13.314440),
                AbsoluteLocation(52.514046, 13.317432),
                AbsoluteLocation(52.516047, 13.316296))

            spawnNVehicles = 10
            spawnLocation = RouteLocation(route, 0)

            vehicleSkills {
                mission()
                followVehicle()
                trafficLight()
            }
            vehicleFeatures {
                staticRouteProvider { route }
                trafficLightProvider()
            }
        }

        despawnOn {
            endOfRoute()
        }
    }
}
```

Listing 2: An Example Scenario

the given control points, specified as geo coordinates. The behavior of vehicles spawned is defined by adding the desired set of *Skills* and *Feature*. For this example, a minimal set of three skills is required in order to follow the route (mission-skill), consider any other vehicle in front (follow-vehicle-skill) and to stop at red lights (traffic-light-skill). The *Skills* are provided with the route and traffic lights by the corresponding *Features*. Finally, the *despawnOn {}* block creates a *Despawner*, which removes vehicles that reached the end of their route. Leaving the *scenario {}* block starts the scenario.

Similar to the scenario definition, *Skills* are preferably written in Kotlin as well, although we have not yet created a DSL for their definition. Listing 3 gives the source code for a simplified but working version of the traffic-light-skill. The class implements the *Skill* interface and overrides two containing methods: *onSpawn()* is called on vehicle creation. It obtains references to the *Features* required by the *Skill*. The actual logic is defined within *takeAction()*, called on every simulation step. It uses the *trafficLightProvider* to obtain a reference to the next relevant traffic light. If one is found, the *routeProvider* is queried for the distance to the traffic light along the route. Next, the needed acceleration is computed to brake to standstill in front of the traffic light. If the traffic light is not passable and the computed brake acceleration is above a threshold of 2 m/s², the brake is requested with a priority of *RULE_TRAFFIC_LIGHT*. If the traffic light is green or the distance to it is large enough, the traffic-light-skill performs no action and lower priority actions control the vehicle.

V. Related Work

A. Subsumption architecture

Our driver model composed of reusable behavior entities, that are either atomic or a combination of other atomic entities, is inspired by the subsumption architecture [16]. It addresses operation of robots in a complex and unpredictable environment which their designers don't know completely at design time. Instead of piping complex individual tasks in a sense-plan-act manner, tasks are split into behaviors elements and arranged horizontally (and called layers). That means, each

```
class TrafficLightSkill : Skill {
    lateinit var routeProvider: RouteProviderFeature
    lateinit var trafficLightProvider: TrafficLightProviderFeature

    private val brakeAccelThreshold = 2.0

    // Called once on vehicle setup.
    override fun onSpawn(vehicle: ScenarioVehicle) {
        routeProvider = vehicle.features.require(
                RouteProviderFeature::class.java)
        trafficLightProvider = vehicle.features.require(
                TrafficLightProviderFeature::class.java)
    }

    // Called on every simulation step.
    override fun takeAction(vehicle: ScenarioVehicle, deltaT: Double,
                            resultAction: DesiredAction) {
        val nextTrafficLight = trafficLightProvider
                                    .getNextTrafficLight()
        nextTrafficLight?.let { trafficLight ->
            val distanceToTrafficLight = routeProvider
                        .routeDistanceTo(trafficLight.location)
            val accelToStandstill = AccelerationHelper
                        .computeAccelerationForStandstill(
                             vehicle.speed, distanceToTrafficLight)

            if (trafficLight.isNotPassable() &&
                        accelToStandstill > brakeAccelThreshold) {
                resultAction.brake(accelToStandstill,
                                   Priority.RULE_TRAFFIC_LIGHT)
            }
        }
    }
}
```

Listing 3: Traffic Light Skill

task is enabled to control the robot alone in a minimalistic way. The horizontal tasks contain all elements of classical sense-plan-act architectures, however realized in a simplified way so that each task implements one behavior only. In that way, each task only needs to handle one manageable problem and instead of the whole complex process of navigation including perception and planning. This concept reduces or even spares direct communications between layers, which decouples the whole operation of a robot in complexity. However, in order to arbitrate the access of the separated tasks to the actors of the robot, determining some kind of priority is required. Subsumption of low-prior behaviors by higher priorities is hard to determine, which is a drawback of the approach. For our prototyping environment, we were able to reduce the complexity of arbitration compared to navigating a robot. With our concept of priorities and constraints we were able to tackle this hurdle within the scope of our prototyping environment.

B. OpenScenario

OpenScenario [17] is an XML based, open format for describing complex simulation scenarios, adopted by the standardization organization ASAM. We initially considered OpenScenario for scenario definition in our prototyping environment. However, it misses the lightweightness and the flexibility we need to address our iterative prototyping process. Indeed, our scenario definition approach emerged from our early work on binding the OpenScenario description to our implementation. Finally, we decided to expose our API initially intended to bind OpenScenario as actual scenario definition interface. [18] compares OpenScenario with other scenario specification formats.

VI. Outlook

Our road map for evaluation and implementation is presented in the following. In this paper we presented our scenario definition methodology and the initial version of our scenario DSL. The language enables creation of simulation scenarios relevant for developing and testing cooperative

ADAS. In contrast to OpenScenario, it takes advantage of a DSL and, if required, allows the user to script their own extensions directly into the scenario description. As we have shown, the scenario interface enables to create simulations with small effort. We want to encourage the community to contribute their own use cases and ideas in order establish an open and feature-rich scenario description interface.

Our strategy for field evaluation and improvement of our scenario definition methodology is to go open source. In that sense, parts of the implementation of our prototyping environment called PHABMACS with its presented interfaces will be made available open-source as part of the new co-simulation framework Eclipse MOSAIC [13] (see Fig. 8). MOSAIC allows to couple simulators from further domains, such as road traffic, V2X communication, electric mobility, intermodal traffic, traffic management, and the like. This allows to extend the development and testing of cooperative ADAS on a broader level, e.g. by integrating the sub-microscopic simulation of individual vehicles in PHABMACS with large-scale traffic simulations with thousands of vehicles.

The scenario description interface, together with the API for the simulator will be made available as part of Eclipse MOSAIC under the Eclipse Public License 2.0. Fig. 8 depicts an excerpt of the PHABMACS implementation related to the parts which will be open source as well as parts on our road map for further development. We are currently working on complementing our current physics engine [14] with an nVidia PhysX based implementation. While our current implementation already scales out over multiple JVMs on different machines [14], our new engine enables us to better scale up on each machine. We are also working on a WebGL based visualization, which enables running PHABMACS on multiple instances in the cloud and visualize in a web browser.

VII. Conclusion

We presented an environment for rapidly prototyping cooperative ADAS based on vehicle simulation. Its underlying approach is to either bring ideas for cooperative ADAS through the prototyping stage towards plausible candidates for further development, or to discard them as quickly as possible. By designing an efficient scenario definition methodology, we address scenario definition and application parameter tuning as highly repetitive tasks an iterative prototyping process for cooperative ADAS. Our scenario definition methodology, provides best flexibility and a minimal expenditure of time on the developer side. As a consequence, our key design decision is to assume knowledge of Kotlin programming language by developers of cooperative ADAS using our prototyping environment. Instead of complex graphical or verbose XML based interfaces, we provide a lean DSL interface, which enables the named flexibility. In order to address handling the complexity of large application related parameter spaces, we designed a driver model based on composing with reusable behavior entities, inspired by the subsumption architecture [16]. Our strategy for field evaluation and improvement of our scenario definition methodology is to open source parts of the implementation of our prototyping environment called PHABMACS, as part of the new co-simulation framework Eclipse MOSAIC [13].

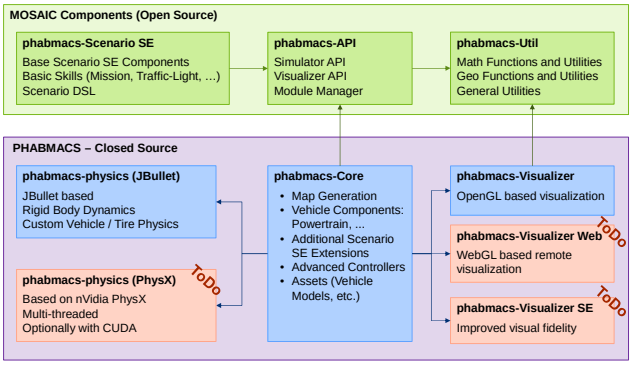

Figure 8: Eclipse Mosaic

References

[1] J. Ziegler *et al.*, "Making Bertha Drive—An Autonomous Journey on a Historic Route," in *IEEE Intelligent Transportation Systems Magazine*, vol. 6, no. 2, pp. 8-20, Summer 2014.

[2] O. Sawade and I. Radusch "A selection process for next generation cooperative driver assistance systems" *Proceedings of the 20th ITS World Congress* pp. 1-9 2013.

[3] S. E. Shladover, et al. "Cooperative Adaptive Cruise Control: Definitions and Operating Concepts." *Journal of the Transportation Research Board* 2489 (2015): 145-152.

[4] SAE International Standard J3016: Taxonomy and Definitions for Terms related to On-Road Motor Vehicle Automated Driving Systems, SAE International (2014)

[5] S. E. Shladover, "Cooperative (rather than autonomous) vehicle-highway automation systems," in *IEEE Intelligent Transportation Systems Magazine*, vol. 1, no. 1, pp. 10-19, Spring 2009.

[6] R. Rajamani, *Vehicle dynamics and control*, Springer Science & Business Media, 2011.

[7] Z. Papp, "Situational Awareness in Intelligent Vehicles," in Handbook of Intelligent Vehicles, A. Eskandarian, edt., Springer , 62-78, 2012

[8] A. Eskandarian, " Fundamentals of Driver Assistance," in *Handbook of Intelligent Vehicles*, A. Eskandarian, edt., Springer, 493-524, 2012

[9] D. Krajzewicz, et al., "Recent development and applications of SUMO-Simulation of Urban MObility." *International Journal On Advances in Systems and Measurements* 5.3&4 (2012).

[10] R. Protzmann, K. Massow and I. Radusch, "An Evaluation Environment and Methodology for Automotive Media Streaming Applications," *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, Birmingham, 2014, pp. 297-304.

[11] B. Schaeufele et al., "Forward-looking automated cooperative longitudinal control: Extending cooperative adaptive cruise control (CACC) with column-wide reach and automated network quality assessment," 2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC), Yokohama, 2017, pp. 1-6.

[12] M. Haklay, and P. Weber. "Openstreetmap: User-generated street maps." *IEEE Pervasive Computing* 7.4 (2008): 12-18.

[13] Eclipse Foundation, Eclipse MOSAIC homepage [Online], https://eclipse.org/mosaic, accessed Feb 28, 2020.

[14] K. Massow and I. Radusch, "A Rapid Prototyping Environment for Cooperative Advanced Driver Assistance Systems, "*Journal of Advanced Transportation*, vol. 2018, 2018.

[15] Kotlin programming language [online] Available: https://kotlinlang.org/. Accessed on 15 June 2020.

[16] R. Brooks, "A robust layered control system for a mobile robot," in *IEEE Journal on Robotics and Automation*, vol. 2, no. 1, 1986.

[17] M. Dupuis, "Openscenario - bringing content to the road," in *2nd OpenSCENARIO Meeting*, 06 2016.

[18] C. Pilz, G. Steinbauer, M. Schratter and D. Watzenig, "Development of a Scenario Simulation Platform to Support Autonomous Driving Verification," *2019 IEEE International Conference on Connected Vehicles and Expo (ICCVE)*, Graz, Austria, 2019, pp. 1-7.

[19] R. Herrtwich, R. (2018). The evolution of the HERE HD Live Map at Daimler. HERE Technologies. https://360.here.com/the-evolution-of-the-hd-live-map. Accessed on 15 June 2020.