A Modular Simulation Framework for Analyzing Platooning Coordination

Christian Krupitzer, Veronika Lesch
University of Würzburg
Würzburg, Germany
firstName.lastName@uni-wuerzburg.de

Michele Segata
DISI, University of Trento
Trento, Italy
msegata@disi.unitn.it

Martin Pfannemüller, Christian Becker
University of Mannheim
Mannheim, Germany
firstName.lastName@uni-mannheim.de

ABSTRACT

Recent developments by companies as Waymo, Uber, or Tesla show that autonomous driving is no science fiction anymore. Coordinated driving applications such as platooning, i.e., driving in convoys of coordinated vehicles, use the full potential of the automation. In this paper, we present a simulation framework for analyzing platooning coordination strategies based on the Plexe platooning simulation. We show the applicability of the simulation framework to support the analysis of platooning coordination strategies in a case study with three coordination strategies.

CCS CONCEPTS

• **General and reference** → **Surveys and overviews**; • **Human-centered computing** → **Ubiquitous and mobile computing**; • **Computer systems organization** → *Self-organizing autonomic computing*; *Embedded and cyber-physical systems*;

KEYWORDS

platooning coordination, coordinated driving, simulation

ACM Reference Format:
Christian Krupitzer, Veronika Lesch, Michele Segata, and Martin Pfannemüller, Christian Becker. 2019. A Modular Simulation Framework for Analyzing Platooning Coordination. In *Proceedings of ACM Workshop on Technologies, mOdels, and Protocols for Cooperative Connected Cars (TOP-Cars 2019).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.475/123_4

TOP-Cars 2019, July 2019, Catania, Italy

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

1 INTRODUCTION

The fast development of Adaptive Cruise Control (ACC) systems and self-driving vehicles in the last years enables new applications as platooning, in which vehicles use ACC and vehicular communication to drive with small inter-vehicle distances [1]. Besides an increase of driver's comfort, the small distances in platoons (i) increase the efficiency in utilization of the street and (ii) reduces the fuel consumption and, hence, the emissions due to slipstream effects as well as (iii) has social implications as it increases safety by eliminating the likeliness of accidents with the ACC system and coordination through communication.

Platooning requires two types of organization. We distinguish the platoon management from platoon coordination: where the first targets the intra-platoon level, i.e., the adjustment of the inter-vehicle distance, the latter focuses on vehicle-platoon and inter-platoon interactions.

In this paper, we target coordination of platooning, hence, the assignment of vehicles to platoons based on individual factors as well as the coordination of platoons especially in scenarios with a small amount of platoonable vehicles, i.e., the pioneering phases of platooning. We present a simulation framework that integrates the platooning simulation Plexe [10] which is based on Veins [12] (including *SUMO* and OMNeT++) with the *Platooning Coordination System (PCS)* for platooning coordination [6]. The simulation framework integrates two components for (i) a simplified configuration of the simulation and choice of the platooning coordination strategy and (ii) web interface based analysis of the simulation results. The modularity of the simulation enables to integrate another simulation environment not depending on the *PCS*. We show the applicability in a case study.

The remainder is structured as follows: Next, section 2 discusses related approaches. section 3 introduces the *PCS* for platooning coordination. section 4 explains the components of our modular simulation framework. section 5 describes the example use of the system within a case study. Lastly, section 6 summarizes the paper and mentions future work.

2 RELATED WORK

This section compares some tools for simulating platooning. We included for comparison Veins [12], Plexe [10], *VSimRTI* [9], the *iTETRIS Control System (iCS)* [7], and *Simpla* [11].

Veins [12] is a framework for vehicle simulation that provides both a realistic simulation of (i) wireless networking using OMNeT++ [13] as well as (ii) realistic vehicle physics based on the traffic simulator *SUMO*. Veins synchronizes these two simulators in a bi-directional way.

Plexe is a modification of Veins [10] that adds functionality to implement platooning. It adds additional driving models that disables security distances and special vehicle controllers for ACC and Cooperative Adaptive Cruise Control (CACC). The results of a simulation such as the velocity of each vehicle can be analyzed using *R* [10].

The *V2X Simulation Runtime Infrastructure (VSimRTI)* [9] provides an infrastructure that for coupling every discrete-event based simulator that supports the so called federate ambassador, e.g., network simulators like *OMNeT++* or *ns-3* [8] with *SUMO*. Such a setup can simulate platooning. Additionally, *VSimRTI* provides a GUI for analysis vehicles on the map and calculating statistics; however, this requires a commercial license. Alternatively, *VSimRTI* offers the *WebSocket Visualizer* for showing vehicles in a Google Map.

The *iTETRIS Control System (iCS)* is part of the *iTETRIS* project [7]. *iCS* allows to integrate *SUMO* and *ns-3*. Similar to Veins, every developer can implement a custom vehicle behavior, e.g., for simulating platooning behavior.

Simpla [11] is a TraCI-Plugin for *SUMO*. It implements a basic platooning logic and supports configurable parameters, such as the distance between vehicles of a platoon. In contrast to Veins, it does not integrate a network simulation.

We compared the presented approaches for simulation of platooning and platooning coordination using the criteria: (i) effort to setup and run a simulation; (ii) effort for testing custom platooning coordination strategies, (iii) visualization of a simulation, and (iv) possibility to compare the results of different platooning coordination strategies.

The effort to setup and run a simulation differs for the approaches. For example, *Simpla* offers a quite easy setup process while Plexe, Veins, *VSimRTI*, and *iCS* require not only *SUMO* but also additional software for the network simulation. The devlopers of Veins and Plexe offer virtual machine images to simplify this process.

Second, we analyze the possibility to implement and test custom platooning coordination strategies. In theory, every project supports this as all are open source and a developer could implement any required functionality. However, the projects can be split into two groups: Veins, *VSimRTI*, and *iCS* do not target platooning, but might be extended for it. Plexe and *Simpla* are originally meant for platooning simulation. So these two projects offer platooning functionality, e.g., the CACC controller. In contrast to Plexe, *Simpla* offers only a limited extendability because it does not implement custom controllers such as CACC to enable platooning but uses different vehicle types for *SUMO*. So the leading vehicle of a platoon has to be from another vehicle type than the following ones. Besides this configuration, it does not support (realistic) V2V or V2I communication [11].

The third characteristic targets the visualization of a simulation. Most of the approaches use the traffic simulator *SUMO* that offers a GUI for observing a simulation. But as most solutions do not run in real time or in parallel on a server without GUI, this solution does not provide the best user experience. *VSimRTI* offers different solutions for a graphical representation of a simulation; however, as it does not focus platooning, it does not display platooning-related information.

Finally, we analyzed the comparability of the results of different platooning coordination strategies. Here, none of the approaches offer a complete solution. All projects that use *SUMO* are able to provide raw log files that contain data, such as the velocity. This data can be analyzed using *R* or other tools, requiring manual effort.

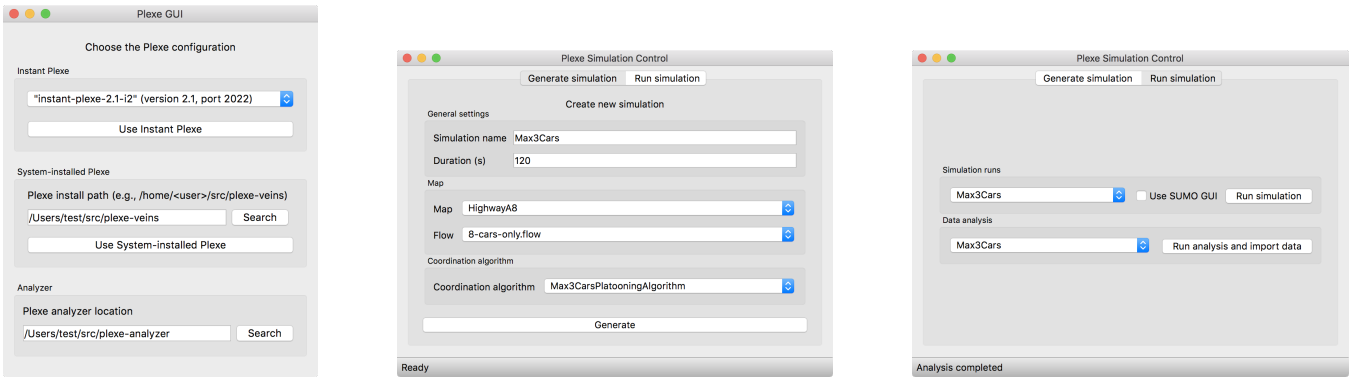
Overall, there are different tools that can be used to show how platooning works. But when it comes to the implementation of different platooning coordination strategies and especially the visual analysis and comparison of the results, the existing approaches do not offer the required functionality. With this work, we try to close this research gap.

3 PLATOONING COORDINATION SYSTEM

The *Platooning Coordination System (PCS)* coordinates the formation of platoons [6]. It receives information from drivers, e.g., their destination, searches a suitable platoon, and navigates the vehicle to the platoon. After reaching the platoon, a vehicle uses vehicle-to-vehicle communication and sensors (e.g., distance sensors) for controlling the joining process. Further, if a platoon meets another platoon, the *PCS* decides whether they should merge or overtake. The *PCS* receives constantly updates about vehicles' positions and determines when to leave a platoon or dissolve a platoon.

In [3], we used the Java-based FESAS Framework [4] to build a demonstrator for the *PCS* for coordinating self-driving Mindstorms robots[1]. Following the MAPE-K approach [2], the *PCS* is structured into the four key functionalities of monitoring the vehicles, analysing which plaotoning actions are necessary (e.g., join requests of vehicles or inter-platoon actions), planning necessary actions, and control the

[1] A video showing the platooning demonstrator can be found at: https://www.youtube.com/watch?v=Nnrbq-4Dn24



(a) choice of the subsystem **(b) simulation configuration panel** **(c) simulation runs and analysis panel**

Figure 1: Screen-shots of the GUI of the simulation tool.

execution of these adaptations. The monitor and the executor elements communicate with the vehicles (or the a platooning simulation) using a JSON-based protocol for data collecting and sending instructions, respectively. For analyzing and planning of platooning using the collected data, the *PCS* can integrate different coordination strategies to comply with individual objectives of drivers, e.g., travel as fuel efficient as possible versus travel as fast as possible while keeping the benefits of platooning. These functions are supported by a shared knowledge repository which represents a typical self-adaptive systems approach [5] and enables adaptations of platooning behavior of vehicles at any time.

4 SIMULATION FRAMEWORK

To simplify simulations with the *PCS*, we build a simulation framework integrating Plexe which contains of a configuration tool and an web interface for analysis of simulation runs. The *PCS* is well integrated into our simulation framework. Accordingly, users do not have to change the *PCS* itself to test new platooning coordination strategies. Rather, users are able to just implement the algorithms of their strategies in a common Java class. We offer a development environment which supports an API for interacting with the *PCS*, such as accessing collected information or abstracting the commands of the JSON protocol. Accordingly, users can just implement their platooning coordination strategies, load them into the simulation tool and evaluate them. In the this section, we detail the components of the simulation framework.

4.1 Simulation Tool

Using Python, we implemented the simulation tool[2]. A Graphical User Interface (GUI) permits the users to quickly setup a simulation, run it, and import the simulation outcome into the web interface for the visual analysis. Users can choose the Plexe subsystem to be used, which can either be the version of Plexe installed on the host machine or Instant-Plexe, i.e., a Linux-based Virtual Machine (VM) which comes with all the required software pre-installed (fig. 1a). In addition, it permits the user to choose the location of the web analyzer (section 4.2) to enable importing simulation results into the analysis tool.

Once the user chooses the subsystem to use, it presents to the user a window enabling the quick creation of a simulation by entering some basic information such as the duration (fig. 1b). The user can then choose the *SUMO* network map and a traffic flow configuration. Currently, the tool offers two sample maps and a couple of pre-configured flow files, but the user can integrate additional ones. Next, the user chooses the coordination strategy. The different algorithms of the coordination strategies are loaded from a *jar* file and displayed as a choice to the user. This will tell the *PCS* which coordination logic to load and to employ during the simulation. Finally, the tool creates a new Plexe simulation folder including all the required configuration files. Such files are standard OMNeT++ configuration files, as the ones included in any Veins or Plexe tutorial. The experienced user can thus also generate a basic simulation and then manually tune additional parameters (e.g., IEEE 802.11p network parameters). The final tab (fig. 1c) permits the user to run a simulation and, at the end, to extract the data from the simulation and import that into the web analyzer tool.

4.2 Analysis Web Interface

Using the PHP-based symfony framework and JavaScript, we implemented the web interface for enabling the graphical qualitative analysis of platooning coordination simulations.

[2]We will make the code publicly available upon acceptance of the paper.

It is composed of two web pages. The first one lists the simulations that have been run and permits either to analyze one, or to choose two of them for comparison. The list of simulations is retrieved from the database which, in turn, is populated by the GUI interface described in section 4.1. Once a simulation (or a pair) is chosen, the second page of the interface displays the *SUMO* map associated with the simulation and permits to re-play it. Each vehicle is displayed as a rectangle, and the color can either identify the vehicle type or the association to a platoon, i.e., vehicle belonging to the same platoon are assigned with the same color. In addition, by clicking on a vehicle, the interface shows additional information such as current speed and its time evolution, platoon id, position, distance or relative speed to the front vehicle. Additionally, the web analyzer offers a dashboard that shows the aggregated results of simulation run w.r.t. platooning compositions, velocity, environmental pollution, energy consumption, and time spent in platoons.

5 CASE STUDY

This section describes how to define coordination strategies and the analysis of their outcome. In particular, we define three toy coordination strategies:

- All-In-One: the system assigns all vehicles to a single platoon;
- Max-3-Cars: the system assign a vehicle to the closest platoon with less than 3 vehicles. If there is no such platoon, the system creates a new one; and
- Max-Distance: a vehicle is assigned to the closest platoon, provided that this is closer than 400 m.

The coordination strategies are defined as Java classes for the *PCS*. The *generateStrategy* method takes one parameter, i.e., an object (of class *Vehicle*) representing the vehicle for which the coordination strategy should be computed and must return an object (of class *DrivingStrategy*) representing the action. Currently we have three categories of strategies, i.e., *BasicDrivingStrategy*, *CreatePlatoonStrategy*, and *JoinPlatoonStrategy*. The first one simply indicates that the vehicle should drive on its own. The second one creates a new platoon with certain characteristics, e.g., a certain speed and driving on a certain lane. The last one controls the join of a platoon. In here we consider this very small set of actions as a proof-of-concept, but in future we will implement new ones, e.g., to support leaving a platoon or changing platoons.

Once the user implemented the algorithms, they are compiled and included inside a *jar* file, which is loaded by the *PCS* at run-time. Using the configuration tool (section 4.1) the user, when generating a new simulation, can choose the coordination strategy that should be employed. In here we show the pseudo-code of the three aforementioned coordination strategies for the sake of brevity.

```
public DrivingStrategy generateStrategy(Vehicle vehicle) {
  if (vehicle.getDrivingStrategy() instanceof
        BasicDrivingStrategy) {
    Platoon platoon = findBestPlatoon(vehicle);
    if (platoon == null) {
      double platoonSpeed = vehicle.getDesiredVelocity();
      int platoonLane = 0;
      return new CreatePlatoonStrategy(vehicle.getId(),
            platoonLane, platoonSpeed);
    } else {
      return new JoinPlatoonStrategy(vehicle.getId(),
            platoon.getId());
    } return null; }

private Platoon findBestPlatoon(Vehicle vehicle) {
  List<Platoon> platoons = getNearbyPlatoons(vehicle);
  if (platoons.size() == 0) {return null;}
  else{ return platoons[0];} }
```

Listing 1: All-In-One strategy.

```
public DrivingStrategy generateStrategy(Vehicle vehicle) {
  if (vehicle.getDrivingStrategy() instanceof
        BasicDrivingStrategy) {
    Platoon platoon = findBestPlatoon(vehicle);
    if (platoon == null || platoon.getVehicles().size() == 3) {
      double platoonSpeed = vehicle.getDesiredVelocity();
      int platoonLane = 0;
      return new CreatePlatoonStrategy(vehicle.getId(),
            platoonLane, platoonSpeed);
    } else {
      return new JoinPlatoonStrategy(vehicle.getId(),
            platoon.getId());
    } }

  if (vehicle.getDrivingStrategy() instanceof
        JoinPlatoonStrategy) {
    JoinPlatoonStrategy strategy = (JoinPlatoonStrategy)
          vehicle.getDrivingStrategy();
    Platoon platoon = getPlatoon(strategy.getPlatoonId());
    if (platoon.getVehicles().size() == 3) {
      double platoonSpeed = vehicle.getDesiredVelocity();
      int platoonLane = 0;
      return new CreatePlatoonStrategy(vehicle.getId(),
            platoonLane, platoonSpeed);
    } } return null; }

private Platoon findBestPlatoon(Vehicle vehicle) {
  List<Platoon> platoons = getNearbyPlatoons(vehicle);
  double minDistance = ∞;
  Platoon chosen = null;
  for (Platoon platoon : platoons) {
    double distance = calculateDistance(vehicle,
          getVehicle(platoon.getLeaderId()));
    if (distance > 0 && distance < minDistance &&
          platoon.getVehicles().size() < 3) {
      chosen = platoon;  minDistance = distance;
    } }  return chosen;  }
```

Listing 2: Max-3-Cars strategy.

```
public DrivingStrategy generateStrategy(Vehicle vehicle) {
  if (vehicle.getDrivingStrategy() instanceof
        BasicDrivingStrategy) {
    Platoon platoon = findBestPlatoon(vehicle);
    if (platoon == null) {
      double platoonSpeed = vehicle.getDesiredSpeed();
      int platoonLane = 0;
      return new CreatePlatoonStrategy(vehicle.getId(),
            platoonLane, platoonSpeed);
    } else { return new JoinPlatoonStrategy(vehicle.getId(),
          platoon.getId());
    } }  return null; }

private Platoon findBestPlatoon(Vehicle vehicle) {
  List<Platoon> platoons = getNearbyPlatoons(vehicle);
  for (Platoon platoon : platoons) {
      double distance = calculateDistance(vehicle,
            getVehicle(platoon.getLeaderId()));
      if (distance > 0 && distance < 400) {return platoon;}
    } }  return null; }
```

Listing 3: Max-Distance strategy.

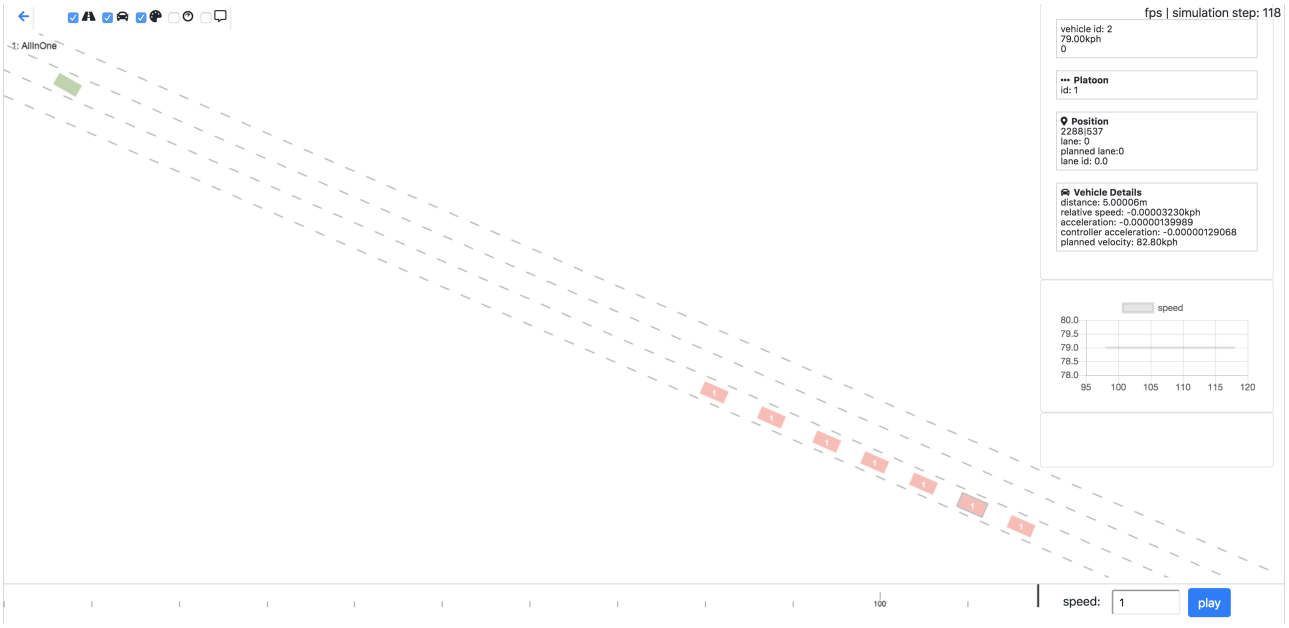


Figure 2: Screen-shot of the web analyzer showing the result of the All-In-One strategy.

Listings 1 to 3 list the three implemented coordination strategies. Each of the three coordination strategies, within the *generateStrategy* method, first check whether the vehicle is still driving on its own (*BasicDrivingStrategy*). If that is the case, the coordination strategy invokes the *findBestPlatoon* method, which actually finds the platoon given the constraints of the strategy. If no platoon is found, the vehicle becomes a new platoon; otherwise the coordination strategy commands the vehicle to join the best one. The only difference in the *generateStrategy* method is for the Max-3-Cars coordination strategy (listing 2), which evaluates if the platoon that should be joined has become larger in the meanwhile. If that is the case, the vehicle changes its coordination strategy and becomes a new platoon.

Inside the *findBestPlatoon* method, the coordination strategy exploits the functionalities provided by the *PCS*, e.g., *getNearbyPlatoons*, *calculateDistance*, and *getVehicle*. As shown in the code snippets, the *findBestPlatoon* method is a straightforward implementation of the coordination strategy.

To test the behavior of the coordination strategies and to show the analysis tool described in section 4.2 we create three simple 8-car simulations, each of which using one of the coordination strategies. Using the tool described in section 4.1 we run the simulation, extract the data, and import it into the database of the web analyzer. With respect to simulation parameters, the desired platoon speed is set to 80 km/h, while the *PCS* commands the joiners to speed up to 130 km/h to catch-up with the platoon. This speed delta is clearly unrealistic and it is used here only to make the join process quicker. As the communication parameters are not relevant in this paper, we omit them. We are just using the IEEE 802.11p PHY and MAC models provided by Veins.

Figure 2 shows a screen-shot of the web analyzer for the All-In-One coordination strategy simulation. In particular, the first seven vehicles have already formed a platoon, and the tool shows this by drawing them all with the same color. The eight vehicle is still approaching the platoon, so it displayed with a different color. At the bottom, the user can click on the time line to move back and forward, and can also choose to play the simulation automatically. Finally, by clicking on a vehicle, on the top-right corner the tool shows information about the selected car, including identifier, current speed, position, platoon identifier, etc. In addition, it is possible to show time series of certain quantities. As a first release, we are only showing the speed as an example, but we currently add information such as fuel consumption.

Figure 3 shows a screen-shot of the analyzer in comparison mode, after choosing the Max-3-Cars and the Max-Distance simulations for comparison. The two views in the browser page are linked together, so they display the same portion of the road at the same zoom level. The Max-3-Cars coordination strategy (top of fig. 3) results in the creation of three platoons, the first two with three vehicles and the last with the remaining two cars (not displayed in the figure due to the zoom level). Conversely, the lower part of fig. 3, shows the outcome of the Max-Distance strategy. The first platoon is composed of four vehicles because the three joiners were closer than the chosen 400 m at the time of joining.

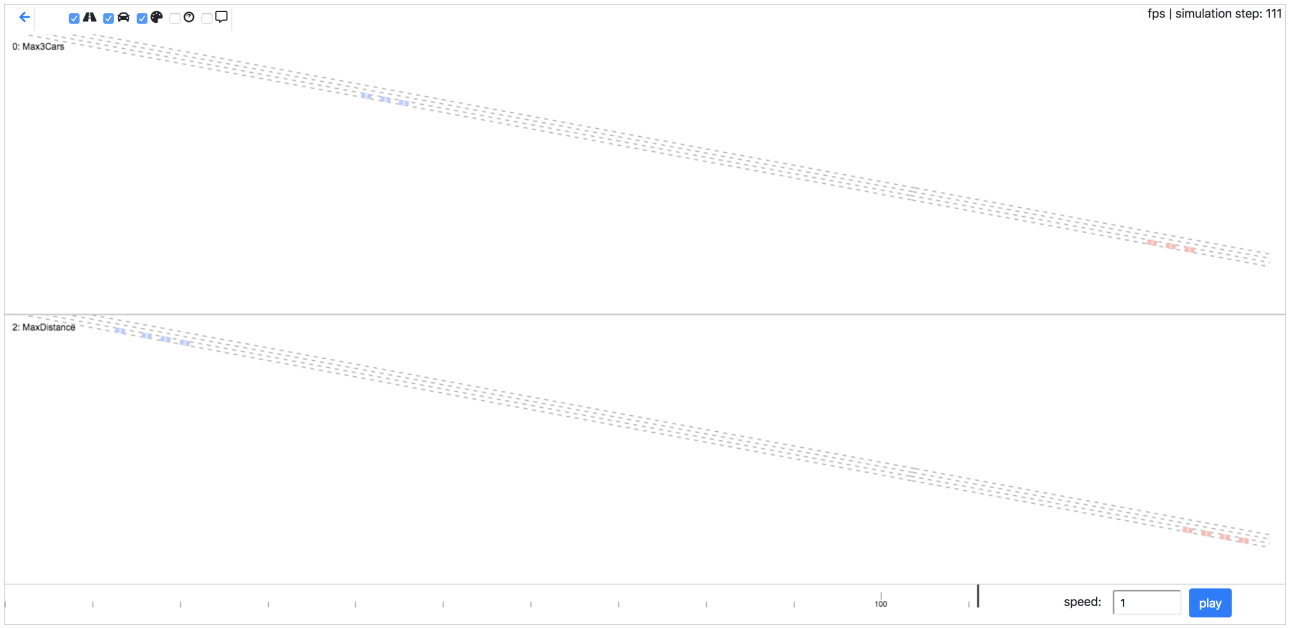


Figure 3: Screen-shot of the web analyzer used in comparison mode, showing the results of the Max-3-Cars (top) and the Max-Distance strategies.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented a tool for supporting the configuration of simulations and the visualization of the simulation and statics. We show the applicability with the Plexe simulator in combination with the *PCS* for coordinating platooning. Due to modularity of the simulation framework, the simulation itself can be substituted.

We currently extend the graphical analyzer with a dashboard to include fuel consumption analysis. Fuel consumption stats are already collected but they not yet displayed. Further, the *PCS* can now manage non-platooning interfering vehicles if they are communicating, so that the *PCS* is aware of their presence. The detection of non-communicating interfering vehicles using local sensors of autonomic vehciles and communicating this to *PCS* is part of future work.

ACKNOWLEDGMENTS

This work has been co-funded by the German Research Foundation (DFG) as part of project A4 within CRC 1053– MAKI. The authors would like to thank their students Kristina Eckert and Johannes Saal for their contribution.

REFERENCES

[1] Carl Bergenhem, Henrik Petterson, Erik Coelingh, Christofer Englung, Steven Shladover, and Sadayuki Tsungawa. 2012. Overview of Platooning Systems. In *Proc. ITSWC*.

[2] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. *IEEE Computer* 36, 1 (2003), 41–50.

[3] C Krupitzer, M Breitbach, J Saal, C Becker, M Segata, and R. Lo Cigno. 2017. RoCoSys: A framework for coordination of mobile IoT devices. In *Proc. PerComW*. 485–490.

[4] Christian Krupitzer, Felix Maximilian Roth, Christian Becker, M. Weckesser, Malte Lochau, and Andy Schürr. 2016. FESAS IDE: An Integrated Development Environment for Autonomic Computing. In *Proc. ICAC*. 15–24.

[5] Christian Krupitzer, Felix Maximilian Roth, Sebastian Vansyckel, Gregor Schiele, and Christian Becker. 2015. A survey on engineering approaches for self-adaptive systems. *PMCJ* 17 (2015), 184–206.

[6] Christian Krupitzer, Michele Segata, Martin Breitbach, Samy El-Tawab, Sven Tomforde, and Christian Becker. 2018. Towards Infrastructure-Aided Self-Organized Hybrid Platooning. In *Proc. GCIoT*.

[7] Michele Rondinone, Julen Maneros, Daniel Krajzewicz et al. 2013. ITETRIS: A Modular Simulation Platform for the Large Scale Evaluation of Cooperative ITS Applications. *Simulation Modelling Practice and Theory* 34 (2013), 99–125.

[8] George Riley and Thomas Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. 15–34.

[9] Björn Schünemann. 2011. V2X simulation runtime infrastructure VSimRTI: An assessment tool to design smart traffic management systems. *Computer Networks* 55, 14 (2011), 3189–3198.

[10] Michele Segata, Stefan Joerer, Bastian Bloessl, Christoph Sommer, Falko Dressler, and Renato Lo Cigno. 2014. PLEXE: A Platooning Extension for Veins. In *Proc. VNC*. 53–60.

[11] Simpla. 2018. Simpla Webpage, URL: http://sumo.dlr.de/wiki/Simpla, Accessed: 01.11.2018. http://sumo.dlr.de/wiki/Simpla

[12] Christoph Sommer, Reinhard German, and Falko Dressler. 2011. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE TMC* 10, 1 (2011), 3–15.

[13] Andras Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *Proc. SIMUTools*.