Platooning in SUMO: An Open Source Implementation

Michele Segata
Dept. of Information Engineering and Computer Science, University of Trento, Italy
msegata@disi.unitn.it

***Abstract*—Platooning is an application that enhances autonomous driving by means of cooperation. Vehicles, by using wireless communication, exchange data such as position, speed, and acceleration to reduce their inter-vehicle gap and drive in groups, called indeed platoons. The benefits span from improved infrastructure usage to reduced pollution, and from increased safety to a better travel experience. The analysis of a platooning system is complex as it encompasses several research fields, including communication, control theory, traffic engineering, and vehicle dynamics. In addition, even small field operational tests can be extremely expensive in terms of money and time, let alone large scale scenarios. Research studies in platooning can thus highly benefit from simulative analyses, as they overcome the limitations of real world test beds. To this purpose, we developed PLEXE, an extension to the Veins vehicular networking framework that enables the simulation of platooning systems, both from a network and a vehicle dynamics perspective. In this paper we detail the changes PLEXE makes to SUMO to simulate platooning control systems, including realistic engine models for the simulation of vehicle dynamics. In addition, we provide two sample use cases to show its potential.**

I. Introduction and Background

Autonomous driving has definitely become the hottest topic in automotive in recent years. Self-driving cars can possibly change the way we conceive vehicles today. Besides improving our safety, these vehicles pave the road towards new visionary applications. Autonomous cars do not only relieve the human from driving duties but, for example, we can have the car pick us up in a different place from the one where we left it. Or think about car sharing: usually a user that needs a vehicle needs to fetch the car where the previous user left it. Autonomous vehicles could get rid of this problem.

While self-driving cars can improve safety by avoiding typical human errors such as distractions or drunken driving, they share the same limitation of human beings, i.e., the limited field of view. Autonomous vehicles exploit sensors to collect data and take their driving decisions, but sensors cannot see behind obstacles, just like the human eye. The way to improve the efficiency and the benefits of autonomous vehicles is thus through cooperation: vehicles should share information by means of wireless communication, overcoming the limits of sensors. As an example, take an intersection collision avoidance application. The exchanged information is used to understand whether two vehicles out of each others sight are in collision course [1].

Another example application is platooning [2]–[4]. With platooning, vehicles share information such as position, speed, and acceleration to form group of cars closely following each other in an autonomous way. The benefits of such an application obviously include safety, but driving at a close distance also reduces the air drag and thus fuel consumption.

The design of platooning systems poses several challenges due to its interdisciplinary nature. The analysis this application is non-trivial, as it requires vehicles, control systems, communication devices, and infrastructure. Simulation-based techniques can thus be highly helpful, but to trust the results the models should be realistic enough. For this reason we developed PLEXE [5].

PLEXE extends the Veins vehicular networking framework [6] to model platooning systems. The base Veins framework couples the OMNeT++ network simulator with SUMO, providing the research community with a tool that can realistically generate vehicle traces and precisely simulate wireless communication. PLEXE takes a step further, implementing platooning control algorithms and detailed engine models within SUMO, enabling platooning studies both from a communication and a vehicle dynamics perspective. PLEXE is Open Source and free to download[1].

The goals of PLEXE are

- Not being a new simulator: PLEXE extends two well-known network and mobility simulators, so users do not need to learn a new tool;
- Being versatile: PLEXE can be used to perform joint network and control analyses;
- Extensibility: users should easily be able to implement additional models within the simulator; and
- Openness: PLEXE must be free to download and modify.

In this paper we describe the changes we made to SUMO to realize PLEXE, we show two example use cases, and highlight the challenges we encountered.

II. Platooning Primer

The fundamental component of platooning is the Cooperative Adaptive Cruise Control (CACC). The CACC is an extended version of a standard Adaptive Cruise Control (ACC) that exploits wireless communication to reduce the inter-vehicle gap. The ACC, instead, relies only on a radar or a lidar. The difference in performance is non-marginal and highlights the gap between selfish, sensor-based vehicles and cooperative vehicles.

[1] http://plexe.car2x.org

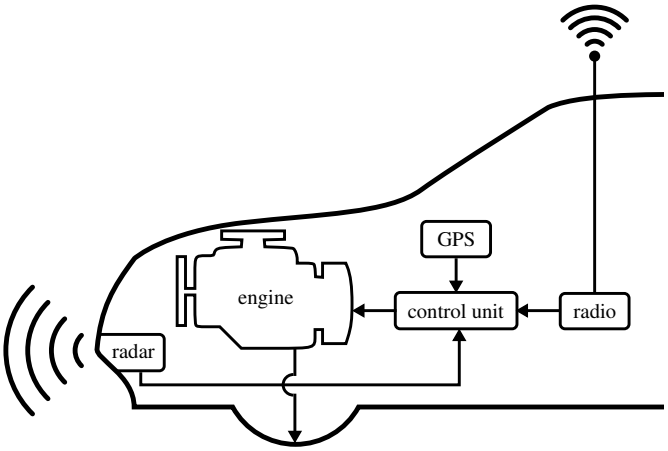


Figure 1: Sketch of the control system simulated in PLEXE.

As an example, the inter-vehicle gap for an ACC is a constant time headway, i.e., the actual distance is computed as the product of the time headway and the current speed, so the distance increases proportionally with speed. For control stability reasons an ACC must keep a time headway to the front vehicle larger than roughly 1 s to 2 s, which is the same safe distance that must be kept by human drivers [7]. By exploiting wireless communication, the CACC developed in [8] is perfectly stable with a time headway of 0.6 s. Each vehicle sends its own "desired" acceleration to the vehicle behind, increasing the reactivity of the CACC. The design in [9], by using data received both from the front and the leading vehicle is stable under a constant spacing policy, i.e., where no time headway is required. In the demonstration, the vehicles drive at a fixed distance of 6 m. For a sensor-based vehicle only, obtaining the information of the leading vehicle is hardly possible.

The principle of operation of a CACC is conceptually simple. The CACC longitudinally controls the vehicle by computing a desired acceleration u that should be applied to maintain a certain inter-vehicle gap. To take its decision the control system obtains data from sensors and via wireless communication: this data can include distance to the front vehicle, position, speed, or acceleration of other vehicles. The acceleration computed by the control system is then given to the lower components for actuation. The engine (or the braking system) requires some time to actuate the command: this time is referred to as actuation lag. Figure 1 shows a sketch of the simulated system.

We can model a cruise control as a function $u = C(\boldsymbol{x}, \boldsymbol{r})$, where u is the desired acceleration, $\boldsymbol{x}$ is a set of variables representing the state of the vehicle, and $\boldsymbol{r}$ a set of reference variables. The simplest example is a standard cruise control, i.e., a device that simply maintains a desired speed set by the driver. A very simple control system achieving this goal can be defined as

$$u = C\left(\boldsymbol{x} = \{\dot{x}\}, \boldsymbol{r} = \{\dot{x}_{\text{des}}\}\right) = k_p\left(\dot{x}_{\text{des}} - \dot{x}\right). \qquad (1)$$

The desired acceleration u is computed as the difference between the desired speed $\dot{x}_{\text{des}}$ (reference value) and the actual speed $\dot{x}$ (state of the vehicle), i.e., the speed error. The speed error is multiplied by a gain k_p, which determines how "reactive" the function is to errors. More sophisticated controllers like an ACC or a CACC follow the same principle, but the sets $\boldsymbol{x}$ and $\boldsymbol{r}$ include additional variables, like acceleration, speed, or position of the own and other vehicles in the platoon.

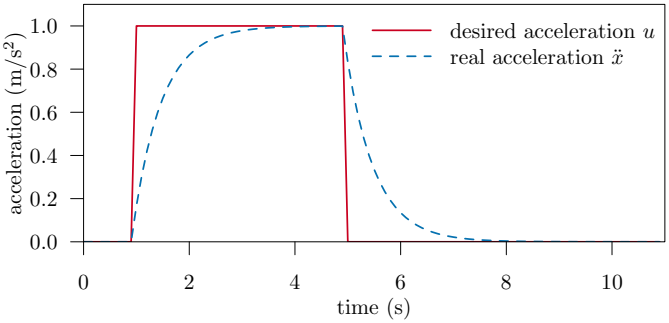


Figure 2: Engine actuation lag using a first order lag.

The desired acceleration u is then passed to engine control for actuation. In control theory nomenclature the equation representing the real system is called the plant. The plant is a function $\ddot{x} = P(u, \boldsymbol{x})$ that computes the real acceleration $\ddot{x}$ given the desired acceleration u and the current vehicle state $\boldsymbol{x}$. One simple but commonly used model is a first order lag [7], [8], which computes the real acceleration by applying a first order low-pass filter to the desired acceleration. Figure 2 shows the behavior of such a model.

In a discrete simulator, this is implemented as

$$\ddot{x}_k = P(u, \boldsymbol{x} = \{\ddot{x}_{k-1}\}) = \alpha u + (1 - \alpha)\ddot{x}_{k-1}, \qquad (2)$$

where

$$\alpha = \frac{\Delta_t}{\tau + \Delta_t}. \qquad (3)$$

Equation (2) computes the acceleration at time step k using the desired acceleration computed by the controller and the acceleration at time step $k-1$. In Equation (3), Δ_t represents the sampling time of the simulator while τ is the engine time constant in seconds, i.e., the delay of the engine. In [7], τ is assumed to be 0.5 s.

This is a simple example, but the engine dynamics can be made more and more realistic by considering further vehicle informations. In PLEXE, we implement the first order lag model, as well as a realistic engine model. For brevity, in this paper we do not describe the whole model. The interested reader can refer to [4, Section 2.3] for further details.

The goal of PLEXE is to implement cruise control systems and engine dynamics within SUMO. The next section describes how this has been achieved.

III. IMPLEMENTING PLATOONING IN SUMO

A. PLEXE *Structure*

This section briefly explains the structure of PLEXE to get a general view on how the simulator works. PLEXE is built upon Veins, a vehicular networking framework that couples

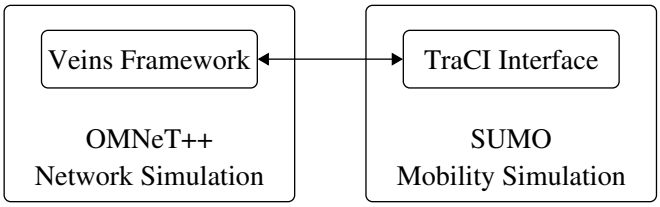


Figure 3: Structure of the Veins vehicular networking framework.

OMNeT++ and SUMO. OMNeT++ takes care of simulating communication between network nodes, while SUMO generate realistic vehicular traffic traces for moving the nodes within the network simulator. Veins, through the TraCI interface, bidirectionally couples the simulators: each vehicle in SUMO has a corresponding network node in OMNeT++, and the node in OMNeT++ moves accordingly. In addition, a node in OMNeT++ can change the behavior of a vehicle in SUMO, for example, by changing the planned route. Veins also offers realistic channel, physical, and medium access layer models for vehicular communication (IEEE 802.11p) [10], as well as higher layer application primitives. Figure 3 shows the structure of Veins.

PLEXE is built on top of Veins, and in particular it extends both simulators to enable the study of platooning systems. The main changes are applied to SUMO to implement CACCs and engine models. In addition we extend the TraCI interface exploited by Veins for the bidirectional coupling to make the SUMO platooning models accessible within the OMNeT++ modules. The next section describes the implementation details.

B. Longitudinal Dynamics: A new Car-following Model

The core of the extension is a new car-following model that implements the CACCs, as well as embedding a Krauss car-following model that can "drive" the vehicle when automated controllers are disabled. This enables complex studies where drivers can switch on and off automated driving depending on some conditions. For example, we can imagine a driver to enter the highway, approach a platoon, and only then switch to autonomous driving.

The car-following model is supported by an enriched `VehicleVariables` class. The class handling this data is the `CC_VehicleVariables` class, which stores controller parameters, vehicle state, configuration of the platoon, and user choices.

The specialized car-following model is implemented within the `MSCFModel_CC` class, where "CC" stands for cruise control. The model implements one ACC taken from [7] and three CACCs [8], [9], [11]. This is done by keeping in mind that researchers might want to implement and test additional control algorithms, so adding new ones is straightforward.

Algorithm 1 shows the pseudo-code for the new car-following model. In general, SUMO invokes one of the methods that ask the model for the speed to use in the simulation step, for example FOLLOWSPEED. If the active "driver" is the human-behavioral model, then we simply invoke the same method within the `MSCFModel_Krauss` class. If, instead, the user enables an autonomous controller, the model invokes the method computing the speed to be used in the next simulation step. Within this method, the model computes the desired acceleration for the chosen controller and then passes this value to the engine model, which considers the actuation lag and computes the real acceleration of the vehicle.

Finally, when SUMO invokes MOVEHELPER to obtain the speed of the vehicle for the current time step, we return the speed value depending on the invoked method. In addition, we update the state of the vehicle.

As SUMO is not designed to simulate driving systems down to the level of detail reached by PLEXE, supporting the engine model requires some "hacks". The main problem is that SUMO car-following models are "speed-oriented", i.e., to compute the next speed value each model considers own current speed, front vehicle speed, and distance to the front vehicle.

To compute the next speed value, the engine model requires the current acceleration as well. To account for this requirement, the current acceleration is stored within the `CC_VehicleVariables` class, so the engine model can access it to compute the real acceleration and store the new value. This is, however, not enough as SUMO invokes the model multiple times for each time step. The model should only store the real acceleration value for the actually chosen speed. For example, imagine that SUMO invokes the FREESPEED and the FOLLOWSPEED methods. The methods will compute their own speed value, each of which is associated with a specific acceleration. If SUMO, for example, uses the new speed computed by FOLLOWSPEED, then the model should store that specific acceleration value.

This required to track the method invoking the model to store temporary values, and then store only the used ones inside the MOVEHELPER method. Some invocations do not even require storing temporary acceleration values, as they will never be used. One example is the lane-change model that calls the car-following model to understand whether it is worth changing lane. In this case the model needs to know that it is being invoked by the lane changing logic.

The currently available engine models include a first order lag and a realistic engine model. PLEXE defines a `GenericEngineModel` class from which all engine models should inherit. The class mainly defines a GETREALACCELERATION method which, given the current vehicle speed, the current acceleration, and the desired acceleration, computes the acceleration after actuation. The two engine models are implemented into the `FirstOrderLagModel` and the `RealisticEngineModel` classes.

C. Accessing the Model via TraCI

Users can access the model to enable cruise controllers, set parameters, change and query vehicle status, and so on. This is realized by extending the TraCI interface with a set of APIs. We do not list all the API methods here for the sake of brevity. Further information can be retrieved on the official website.

Algorithm 1 Pseudo code for the `MSCFModel_CC` class.

```
1: function FOLLOWSPEED(vehicle)
2:     if vehicle.controller == HUMAN then
3:         return Krauss.followSpeed(vehicle)
4:     else
5:         return nextSpeed(vehicle, FOLLOW_SPEED)
6:     end if
7: end function
8: function FREESPEED(vehicle)
9:     if vehicle.controller == HUMAN then
10:        return Krauss.freeSpeed(vehicle)
11:    else
12:        return nextSpeed(vehicle, FREE_SPEED)
13:    end if
14: end function
15: function NEXTSPEED(vehicle, invoking)
16:    k = currentTimestep
17:    x = vehicle.state
18:    r = vehicle.reference
19:    cacc = vehicle.controller
20:    u = C(cacc, x, r)
21:    engine = vehicle.engineModel
22:    ẍ = P(engine, u, x)
23:    ẋ = ẋ_{k-1} + ẍ · Δ_t
24:    if invoking == FOLLOW_SPEED then
25:        vehicle.followSpeedTime = k
26:        vehicle.followSpeed = {u, ẋ}
27:    else
28:        vehicle.freeSpeed = {u, ẋ}
29:    end if
30:    return ẋ
31: end function
32: function MOVEHELPER(vehicle)
33:    k = currentTimestep
34:    if vehicle.controller == HUMAN then
35:        ẋ_k = Krauss.moveHelper(vehicle)
36:        u_k = 0
37:    else
38:        if vehicle.followSpeedTime == k then
39:            {u_k, ẋ_k} = vehicle.followSpeed
40:        else
41:            {u_k, ẋ_k} = vehicle.freeSpeed
42:        end if
43:    end if
44:    ẍ_k = (ẋ_k − ẋ_{k-1})/Δ_t
45:    vehicle.state = {u_k, ẍ_k, ẋ_k}
46:    return ẋ_k
47: end function
```

```
void setGenericInformation(
  const MSVehicle* veh,
  const struct CCDataHeader &header,
  const void *content
);

int getGenericInformation(
  const MSVehicle* veh,
  const struct CCDataHeader request,
  const void *reqParams,
  void *content
);

struct CCDataHeader {
  int type;
  int size;
};
```

Listing 1: Access methods to the `MSCFModel_CC` class.

Besides the standard APIs, we define new TraCI methods to access and easily extend the model. The class defines two generic access points, i.e., the SETGENERICINFORMATION and the GETGENERICINFORMATION functions. The idea is avoiding the need to change SUMO core files when extending the simulator. For example, when adding a new CACC we want to be able to pass its parameters via TraCI, but we only want to modify the `MSCFModel_CC` class.

Listing 1 shows the prototypes of the two methods used to send data to and retrieve data from the model. TraCI invokes both methods passing the instance of the vehicle for which the action is requested. The `CCDataHeader` structure includes information about the set or get operation. The type indicates the operation, for example the actual parameter the user wants to set, while size indicates number of bytes inside the content (for the setter) or inside the parameters of the request (for the getter). In the case of the setter, the content can be anything, from a basic type such as an integer to a complex data structure.

As an example, if we want to set an integer parameter, size will be set to `sizeof(int)`, while the value can be retrieved by casting the content to an integer.

Concerning the getter, `reqParams` can be used to parameterize the request. The `content` is where the method stores the response which, as for the setter, can be of any kind. The method should return the size of the response in bytes.

These two methods provide the users with a straightforward way of extending the communication with the model using the TraCI interface.

D. Lane Change Model

In addition to the car-following model for longitudinal dynamics, PLEXE extends SUMO by adding a lane changer. The aim of the extension is to enable users to tell platooning vehicles on which lane to drive, regardless of what the lane change model dictates. The logic is defined within the `MSCACCLaneChanger` class, which extends the basic `MSLaneChanger` provided by SUMO. The `MSEdge` class instantiates the new `MSCACCLaneChanger` instead

```
void setFixedLane(
  int laneIndex
);

void setLaneChangeAction(
  enum PLATOONING_LANE_CHANGE_ACTION action
);

enum PLATOONING_LANE_CHANGE_ACTION {
  DRIVER_CHOICE,
  STAY_IN_CURRENT_LANE,
  MOVE_TO_FIXED_LANE
};
```

Listing 2: Lane changing methods.

of `MSLaneChanger`. The new lane changer overrides the methods of the base class, disabling standard behavior when required by the user.

By invoking the TraCI methods in Listing 2 it is possible to tell the vehicle to perform a certain action. With SETFIXEDLANE the vehicle will move to the required lane as soon as this can safely be done, meaning that the vehicle will avoid collisions with other vehicles. With SETLANECHANGEACTION, instead, we specify what the vehicle should do with respect to the lane change logic. By using `DRIVER_CHOICE`, the vehicle behaves using standard SUMO lane change logic, i.e., as if a human driver would be in control. With `STAY_IN_CURRENT_LANE` the vehicle is forced to drive on the current lane, while with `MOVE_TO_FIXED_LANE` the vehicle tries to change to the lane specified via SETFIXEDLANE.

IV. USE CASES

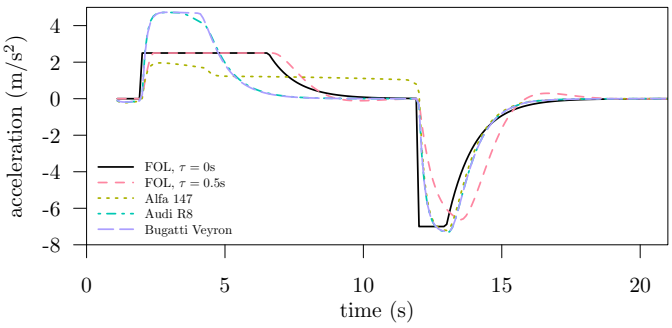
In this section we show some usage examples to highlight the potential of the simulator. In the first use case, we test different parameterizations of the engine models in an accelerate and brake scenario. In the second use case, we briefly show the differences between two control algorithms implemented in PLEXE for a 4-vehicle platoon.

A. Engine Models

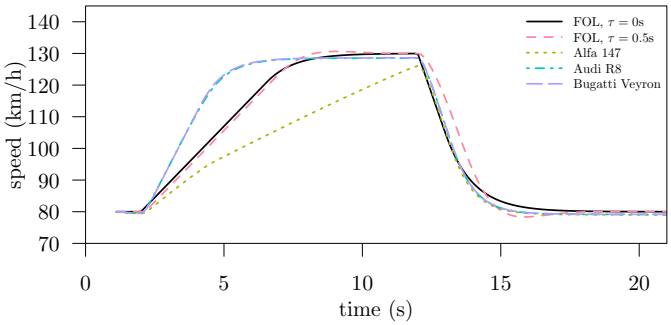
We use a standard cruise control of the form

$$u = \min\left(5, \max\left(-7, \dot{x}_{\text{des}} - \dot{x}\right)\right), \tag{4}$$

i.e., a simple controller that maintains a desired speed. The controller never asks the engine (or the braking system) to accelerate stronger than $5\,\text{m/s}^2$ and to decelerated harder than $7\,\text{m/s}^2$. These limits are not realistic, but we use them for demonstration purposes only. In the scenario we use a single vehicle that enters the road at a speed of $80\,\text{km/h}$. After one second, the desired speed $\dot{x}_{\text{des}}$ is increased to $130\,\text{km/h}$ and then reset to $80\,\text{km/h}$ after additional ten seconds. We test the behavior of the vehicle in five different configurations. The first two use the first order lag model with an engine time constant τ of $0\,\text{s}$ and $0.5\,\text{s}$. When using this model, we have an acceleration limit of $2.5\,\text{m/s}^2$. The remaining three use the realistic engine model, parameterized to reproduce the engine



(a) acceleration



(b) speed

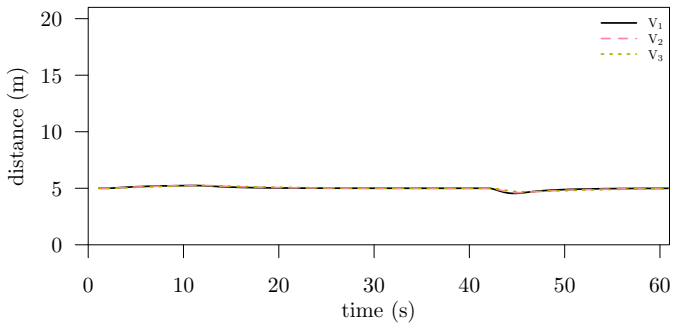
Figure 4: Vehicle dynamics for the engine models test.

of an Alfa Romeo 147, an Audi R8, and a Bugatti Veyron. The engine parameters for the vehicles are available in the PLEXE repository and are not listed here for brevity.

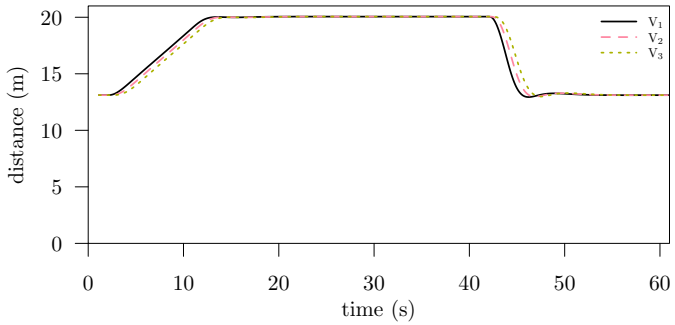
Figure 4 shows the results for the engine test scenario in terms of real acceleration $\ddot{x}$ and speed $\dot{x}$ as function of time. The first noticeable difference is in the behavior of the first order lag model for different values of τ. For $\tau = 0\,\text{s}$ the reaction of the vehicle to commands is instantaneous. The vehicle immediately performs what the controller requires to do, which is clearly unrealistic. By using a time delay $\tau = 0.5\,\text{s}$ the reaction is smoother, and we observe the delay in the actuation, both during the acceleration and in the deceleration phase.

For the realistic engine models each vehicle has its own characteristics. Clearly, as for the first order lag model with $\tau = 0.5\,\text{s}$, the engine (and braking) reaction is not immediate. Depending on the vehicle we can observe different performance. First of all, we can observe than only the two high-end cars are capable of delivering the required maximum acceleration of $5\,\text{m/s}^2$. The Alfa Romeo, depending on the current speed, reaches a maximum acceleration of $2\,\text{m/s}^2$, which drops down to roughly $1\,\text{m/s}^2$ at $5\,\text{s}$ due to gear shifting. In addition, the maximum acceleration slowly decreases over time due to air drag. In the end, the Alfa Romeo is not capable of reaching the target speed before the braking action starts.

This example shows how we can realistically simulate different vehicle behaviors, which is fundamental in the study of platooning systems. Indeed, one interesting question is how a control system performs when the platoon is made by a set of inhomogeneous vehicles. When such a level of modeling

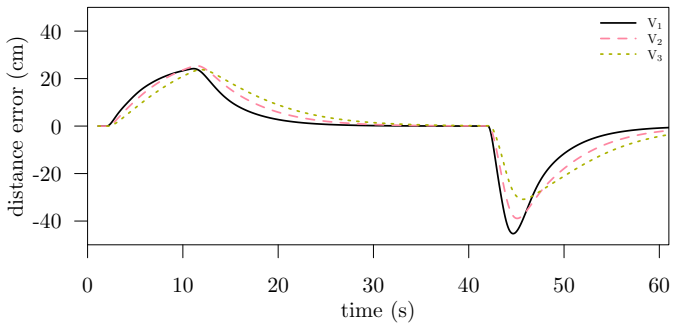


(a) PATH's CACC

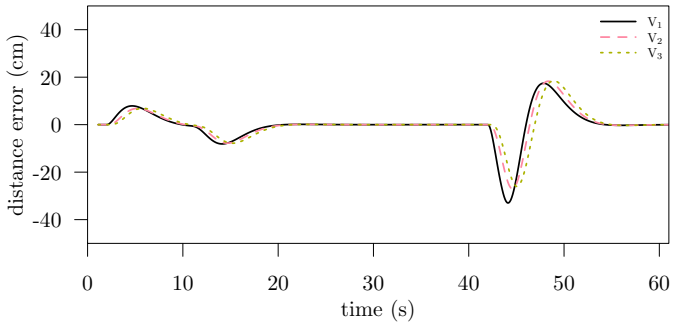


(b) Ploeg's CACC

Figure 5: Distance comparison between two CACC algorithms.



(a) PATH's CACC



(b) Ploeg's CACC

Figure 6: Distance error comparison between two CACC algorithms.

detail is not required, the user can switch to the first order lag model, which, although more simplistic, provides a trade off between a completely unrealistic and a realistic behavior.

B. Control System Analysis

In this analysis we compare the PATH's CACC [9] and the Ploeg's CACC [8]. We test a 4-vehicle platoon traveling on a highway. The leader (referred to as V_0) is driven by a standard cruise control as in Section IV-A, while the three followers (referred to as V_1, V_2, V_3) are controlled by one of the two CACCs. The scenario is similar to the one in Section IV-A, with the only difference that the cruise control desired acceleration u is limited to $1.5\,\mathrm{m/s^2}$. In addition, after increasing the desired speed to $130\,\mathrm{km/h}$, the leader maintains this speed for a longer period of time before decreasing it back to $80\,\mathrm{km/h}$.

The two controllers considered have two different spacing policies. The PATH's CACC uses a constant spacing policy, meaning that the inter-vehicle gap is independent of the speed. In the simulation we set the spacing $d_{\text{des}} = 5\,\mathrm{m}$. The Ploeg's CACC uses a constant headway policy, i.e., the inter-vehicle gap is $d_{\text{des}} = h{\cdot}\dot{x}$, for a given headway time h. We set $h = 0.5\,\mathrm{s}$, which, for the two chosen cruising speeds of $80\,\mathrm{km/h}$ and $130\,\mathrm{km/h}$, translates into an inter-vehicle distance between $13\,\mathrm{km/h}$ to $20\,\mathrm{km/h}$[2].

In both cases the engine is modeled using a first order lag with $\tau = 0.5\,\mathrm{s}$. Finally, the vehicles exchange data by sending wireless broadcast messages ten times a second (the packet transmission rate is $10\,\mathrm{Hz}$). Clearly, the message exchange rate has an impact on the performance of the controller and, if necessary, PLEXE can be employed to analyze such an impact. Here we only focus on the SUMO modeling, so we disregard communication aspects.

Figure 5 shows the performance of the two CACCs in terms of inter-vehicle distance as function of time. In the graphs, V_i indicates the distance between vehicle V_i and vehicle V_{i-1}. The graphs clearly show the difference in terms of spacing policies. While the PATH's CACC maintains a constant gap around the target of $5\,\mathrm{m}$, Ploeg's CACC adapts the distance to the cruising speed.

It is interesting to compare the controllers in terms of distance error, i.e., how much the actual distance deviates from the target distance. Figure 6 plots the distance error in centimeters computed as $d - d_{\text{des}}$. When the error is positive, the vehicle is farther to its front vehicle than it should be. When the error is negative, the vehicles is instead too close. The graphs shows that the PATH's CACC induces a larger distance error, meaning that the constant headway spacing policy improves tracking performance. However, Ploeg's CACC induces some overshooting, while the PATH's CACC smoothly converges to the target distance.

The aim of this section is not decide which control system performs better, but to highlight the potential of the simulator. Using PLEXE it is straightforward to perform comparative analysis of platooning control systems. In addition, users can easily extend the simulator to implement new control strategies

[2] The implementation considers a stand-still distance of $2\,\mathrm{m}$, to avoid the distance to become null in case of a complete stop.

and compare them with the state of the art controllers already available in PLEXE. The official website includes a detailed tutorial showing how this can be done in a few simple steps.

V. CONCLUSION

This paper describes in detail how SUMO has been extended within the PLEXE framework to implement platooning control systems and detailed engine models. As platooning is becoming more and more a hot topic in the scope of automated driving and vehicular networking, there is the need of a realistic simulator to evaluate this innovative driving solution. PLEXE achieves this goal by extending SUMO in an non-invasive way, and permits the user to further modify it to tune the simulator to specific purposes. The paper described the implementation in detail to help users understanding how the extension works. Furthermore, we have seen its potential for two sample use cases, i.e., to observe different vehicle behaviors depending on the chosen engine model and to compare platooning control strategies. The SUMO extension in PLEXE provides a valid, open, and free tool for the analysis of future driving technologies.

REFERENCES

[1] S. Joerer, M. Segata, B. Bloessl, R. Lo Cigno, C. Sommer, and F. Dressler, "A Vehicular Networking Perspective on Estimating Vehicle Collision Probability at Intersections," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 4, pp. 1802–1812, May 2014.

[2] S. Shladover, "PATH at 20 – History and Major Milestones," in *IEEE Intelligent Transportation Systems Conference (ITSC 2006)*, Toronto, Canada, Sep. 2006, pp. 22–29.

[3] C. Bergenhem, Q. Huang, A. Benmimoun, and T. Robinson, "Challenges of Platooning on Public Motorways," in *17th World Congress on Intelligent Transport Systems (ITS 2010)*, Busan, Korea, Oct. 2010.

[4] M. Segata, "Safe and Efficient Communication Protocols for Platooning Control," PhD Thesis (Dissertation), University of Innsbruck, February 2016.

[5] M. Segata, S. Joerer, B. Bloessl, C. Sommer, F. Dressler, and R. Lo Cigno, "PLEXE: A Platooning Extension for Veins," in *6th IEEE Vehicular Networking Conference (VNC 2014)*. Paderborn, Germany: IEEE, Dec. 2014, pp. 53–60.

[6] C. Sommer, R. German, and F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing*, vol. 10, no. 1, pp. 3–15, Jan. 2011.

[7] R. Rajamani, *Vehicle Dynamics and Control*, 2nd ed. Springer, 2012.

[8] J. Ploeg, B. Scheepers, E. van Nunen, N. van de Wouw, and H. Nijmeijer, "Design and Experimental Evaluation of Cooperative Adaptive Cruise Control," in *IEEE International Conference on Intelligent Transportation Systems (ITSC 2011)*. Washington, DC: IEEE, Oct. 2011, pp. 260–265.

[9] R. Rajamani, H.-S. Tan, B. K. Law, and W.-B. Zhang, "Demonstration of Integrated Longitudinal and Lateral Control for the Operation of Automated Vehicles in Platoons," *IEEE Transactions on Control Systems Technology*, vol. 8, no. 4, pp. 695–708, Jul. 2000.

[10] D. Eckhoff and C. Sommer, "A Multi-Channel IEEE 1609.4 and 802.11p EDCA Model for the Veins Framework," in *5th ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2012), Poster Session*. Desenzano, Italy: ACM, March 2012.

[11] S. Santini, A. Salvi, A. S. Valente, A. Pescapè, M. Segata, and R. Lo Cigno, "A Consensus-based Approach for Platooning with Inter-Vehicular Communications," in *34th IEEE Conference on Computer Communications (INFOCOM 2015)*. Hong Kong, China: IEEE, Apr. 2015, pp. 1158–1166.