Plexe Version 1.1 Documentation

Michele Segata (segata@ccs-labs.org)

August 21, 2014

Contents

1 Introduction

Plexe (PLatooning EXtension for vEins) has been developed to enable the study of platooning systems both from a networking and a road traffic perspective. The purpose of this framework is to provide a tool to researchers which enables a detailed simulation of wireless communication among the vehicles (in particular IEEE 802.11p-based communication), together with realistic mobility. To this aim, Plexe has been built upon Veins [4][1] which couples the OMNeT++[2] network simulator with the SUMO[3] road traffic simulator. The aim of this article is to document the simulator, in particular describing

1. Where to download the simulator and how to compile it;
2. How to run the basic example scenario shipped with the simulator;
3. How the original Veins has been extended. This last step is particularly important in order to develop new scenarios and/or network protocols to be studied and evaluated;
4. How to extend the simulator to implement a new, user-defined, controller.

In here we assume that the reader is familiar with the simulation framework Veins, and thus with the basic concepts on network simulation in OMNeT++. It is thus assumed that the user has already installed OMNeT++. If OMNeT++ is not installed, please refer to Section 7.1 or Section 9. Throughout this documentation, the OMNeT++ version used is 4.4.1. Moreover, we also assume that all source tarballs or git folders are placed in the home directory, under the `src` folder (∼/`src`). The simulator requires build tools and libraries to be built (e.g., `g++`). If you are building the simulator in a freshly installed system, you might want to have a look at Section 9, but be sure to read the documentation before starting to use it.

2 Downloading and Building

Plexe source code can be either downloaded in a `tar.bz2` archive, or via `git` through the public repository. To obtain the archive please visit the download page[4]. In there you will find two files, `plexe-veins-1.1.tar.bz2` and `plexe-sumo-1.1.tar.bz2`, containing a modified version of Veins and SUMO respectively. Download, place, and extract them in source folder by typing in your terminal

```
cd
cd src
tar xjf plexe-veins-1.1.tar.bz2
tar xjf plexe-sumo-1.1.tar.bz2
```

To download the source code via `git` instead, clone the `github` repositories with

```
cd
cd src
git clone https://github.com/michele.segata/plexe-veins.git
git clone https://github.com/michele.segata/plexe-sumo.git
```

and checkout the `plexe-1.1` branches.

[1]http://veins.car2x.org
[2]http://www.omnetpp.org
[3]http://www.sumo-sim.org
[4]http://plexe.car2x.org/download

The next step is to compile the two sub-parts of the simulator.

2.1 Building SUMO

The procedure is similar for both Linux and Mac OS systems, but with some small differences in the commands. SUMO depends on some third party libraries which can be installed on a Linux machine with

```
sudo apt-get install libgdal-dev libproj-dev \
     libxerces-c-dev libfox-1.6-dev libtool \
     autoconf
```

After installing the dependencies, SUMO can be configured with

```
cd ~/src/plexe-sumo
make -f Makefile.cvs
./configure
```

On Mac OS X, third party libraries can be installed via MacPorts[5] by typing

```
sudo port install xercesc proj gdal fox
```

The configuration command is slightly different on Mac OS X, as you need to tell the script that the libraries are located in the MacPorts folder (usually `/opt/local`):

```
export CPPFLAGS="$CPPFLAGS -I/opt/local/include"
export LDFLAGS="$LDFLAGS -L/opt/local/lib \
       -framework GLUT -framework OpenGL"
make -f Makefile.cvs
./configure --with-fox=/opt/local \
            --with-proj-gdal=/opt/local \
            --with-xerces=/opt/local
```

On both systems, SUMO can then be built by simply typing `make` on the command line. The final step is to add the SUMO `bin` directory to your `PATH`. Add to your `.bash_rc` (or `.profile` in OS X)

```
export PATH=$PATH:$HOME/src/plexe-sumo/bin
```

Now you should be able to run SUMO by typing either `sumo` or `sumo-gui` for the command line and the GUI version respectively.

2.2 Building Plexe Veins

Building Plexe Veins is the simplest step. Just type the following on the command line

```
cd ~/src/plexe-veins
make -f makemakefiles MODE=release
make MODE=release
```

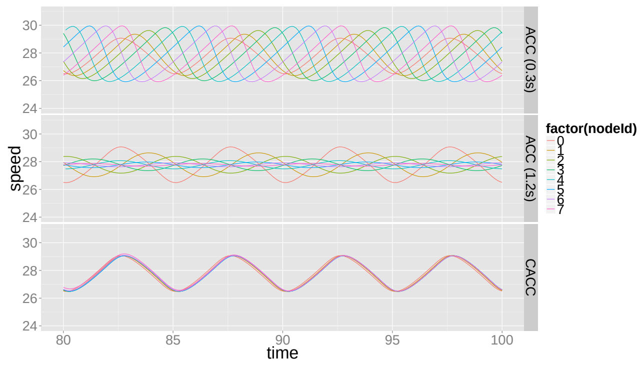
3 Running the Example Experiments

Plexe includes a sample scenario which is located in

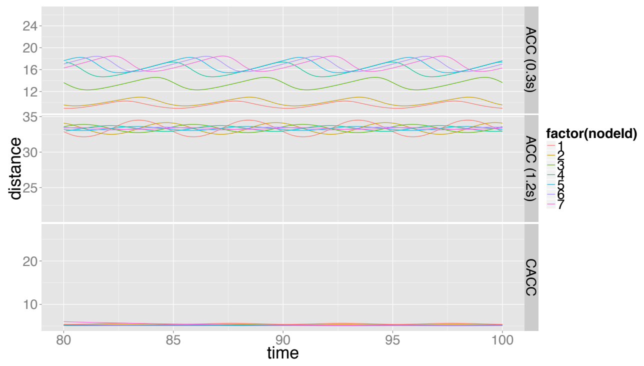
```
cd ~/src/plexe-veins/examples/sinPlatoon
```

The example reproduces a platoon of eight cars traveling for two minutes on a stretch of a freeway. The leader is driving with an average speed of 100 km/h, oscillating

[5]http://www.macports.org



(a) Speed in time



(b) Distance in time

Figure 1: Plots of speeds and distances as function of time for different controllers configurations obtained with the simulator.

in a sinusoidal fashion for demonstration purposes. There are three sub-scenarios, each of which uses a different automated controller setup.

In the first one, all cars are controlled by an Adaptive Cruise Control (ACC) [3, Chapter 6] which headway time T has been set to 0.3 s. In this setup the platoon is string-unstable, because the stability condition $T \geq 2\tau$ [3, Equation (6.26)], where τ is the actuation lag in seconds, does not hold. τ is by default set to 0.5 s. In the second sub-scenario, cars are still controlled by an ACC, but this time in a string-stable manner. The ACC headway time T is indeed set to 1.2 s. Finally, in the third scenario cars are controlled by a Cooperative Adaptive Cruise Control (CACC) [3, Chapter 7]. Each car is receiving speed and acceleration of the first car and of the car immediately in front through wireless communication, in particular using IEEE 802.11p. Cars are maintaining a fixed inter-vehicle gap of 5 m, independently of the current speed.

Before running the experiments, the `sumo-launchd` script must be started. This script waits for Veins simulations to start and automatically launches SUMO for you. To start it, go on your terminal and type

```
cd ~/src/plexe-veins
./sumo-launchd.py -c sumo-gui
```

The `-c` switch tells the script which SUMO command to launch, so use `sumo-gui` for the graphical version, and `sumo` for the command line version. There are some other options, for example to daemonize the script. Use `-h` to access the help.

To now start the simulation, open a new terminal and type

```
cd ~/src/plexe-veins/examples/sinPlatoon
./run -u Cmdenv -f omnetpp.ini -c Sinusoidal -r <run>
```

For the `run` command, `-u Cmdenv` tells to use the OMNeT++ command line environment, so the graphical user interface will not be shown, `-f omnetpp.ini` indicates which OMNeT++ configuration file should be used, `-c Sinusoidal` the simulation configuration within the `omnetpp.ini` file, and `-r <run>` is the number of the simulation to be run. The run number can be either 0, 1, or 2, and each value refers to one of the three sub-scenarios previously described.

The SUMO graphical user interface should pop up, and the simulation should be paused at time 0.00 s. You should see a stretch of an empty freeway. Zoom in at the beginning of the highway on the left side, and then press the play button in SUMO. Cars will now be inserted in the highway one after the other, and they will start to make a platoon. If the simulation is too fast, change the delay time, which will add a delay between each simulation step. You can also right-click the first car and choose `Start tracking` to have SUMO continuously track the leader.

All three simulations run for 120 s (simulation time). The actual running time should be around one minute, depending on the performance of your machine. When all three simulations are completed, you will find the output files with some statistics in the `results` folder. The provided example also includes an `R` script to plot distances and speeds of the cars for the three scenarios. To use it, you will need to have the `R` statistical framework installed together with the `omnetpp` and `ggplot2` packages. If you do not have `R` with all needed packages installed, please see Section 7.2. Otherwise, just type the following to generate two plots

```
cd ~/src/plexe-veins/examples/sinPlatoon/analysis
Rscript plot.R
```

When the script is completed, you should find `speed.pdf` and `distance.pdf` in the `analysis` folder (Figures 1a and 1b). Figure 1a shows how the speeds of the vehicles oscillate in time. For the first scenario, the string-instability is evident. In the second scenario, the oscillations are attenuated along the platoon, due to the string-stability conditions and the large gaps. In the scenario with CACC, every car in the platoon perfectly reproduces what the leader is doing.

Another way to compare the systems is to look at the distances between the cars in time (Figure 1b). In the first scenario, distances do not stabilize; they actually get bigger and bigger. In the second scenario the platoon is string-stable, and this can be clearly seen by looking at how the distance oscillation is attenuated moving towards the tail of the platoon. Such distance, however, is in the order of 33 m, i.e., time headway multiplied by the platoon speed ($T \cdot v$). When considering CACC instead, all cars maintain a fixed distance of 5 m and oscillations are barely noticeable.

4 Implemented Controllers

The standard Plexe version implements three controllers, namely a Cruise Control (CC), an ACC, and a CACC. All controllers are taken from the book by Rajamani [3], but it is possible to extend the simulator and include different models (see Section 6).

The CC [3, Chapter 5], if the automated controllers are activated, drives the car when no vehicle in front is detected by the radar (i.e., for a distance higher than 250 m). The control law is defined as

$$\ddot{x}_{\text{des}} = -k_p \left(\dot{x} - \dot{x}_{\text{des}}\right) - \eta \tag{1}$$

where $\ddot{x}_{\text{des}}$ is the acceleration to be applied, $\dot{x}$ is the current speed, $\dot{x}_{\text{des}}$ is the desired speed, x the current position, and x_{des} the position of a fictitious vehicle in front travelling at the desired speed. k_p and is the gain of the proportional part of the controller, while η is a random disturbance taking into account imprecisions of the actuator and of the speed measure (default set to 0). By default, k_p is set to 1. Notice that in reality, the output of the controller is not the acceleration to be applied $\ddot{x}_{\text{des}}$, but an input u which should then be passed to vehicle's driveline, and then be "transformed" into an actual acceleration. Since this is a simulated system, we directly map the input u to $\ddot{x}_{\text{des}}$.

The CC does consider possible obstacles in front, so if the driver does not deactivate it when approaching a slower vehicle in front, the car would collide with the one in front. The ACC makes use of a radar to detect vehicles in front and automatically slow down the car whenever needed. The control law of the ACC is defined as [3, Chapter 6]

$$\ddot{x}_{i\_des} = -\frac{1}{T}\left(\dot{\varepsilon}_i + \lambda\delta_i\right) \tag{2}$$

$$\delta_i = x_i - x_{i-1} + l_{i-1} + T\dot{x}_i \tag{3}$$

$$\dot{\varepsilon}_i = \dot{x}_i - \dot{x}_{i-1} \tag{4}$$

where T is the time headway in seconds, $\dot{\varepsilon}_i$ is the relative speed between own car and the vehicle in front, and δ_i is the distance error, i.e., the difference between the actual distance ($x_i - x_{i-1} + l_{i-1}$) and the desired distance ($T\dot{x}_i$). As you can notice, the desired distance is dependent on speed. In particular it grows proportionally with speed, and for stability reasons the time headway T cannot be arbitrarily small, and needs to be greater than 1 s (please see [3] for further details). λ is a design parameter which must be strictly greater than 0, and it is set to 0.1 by default.

When the ACC is selected, the interaction between CC and ACC is implemented as

$$\ddot{x}_{\text{des}} = \min(\ddot{x}_{\text{CC}}, \ddot{x}_{\text{ACC}}) \tag{5}$$

Basically, if the CC is mandating to accelerate to reach the desired speed, but the ACC is mandating to slow down because of a vehicle in front, the car follows the instructions of the ACC. Conversely, if the ACC is mandating to accelerate to follow the car in front, but the car has reached its desired speed, the CC will make the car to "detach" from the one in front. Notice that this might not be the best strategy to implement. A more appropriate way to switch between CC and ACC is to use transitional controller derived from range – range rate diagrams [3, Section 6.7.2]. This can be easily implemented in the simulator, but for the sake of simplicity we choose to use this straightforward switching mechanism.

The CACC controller we consider [3, Chapter 7] uses wireless communication to improve performance. In particular, each vehicle feeds acceleration and speed of leader and car directly in front in the controller in order to perform close car following. This communication pattern is not the only possible one, see for example the work by Ploeg et al. [2]. We choose this one because it implements a constant spacing policy, i.e., it is able to maintain a fixed distance which is independent from platoon's speed. The control law of the i-th vehicle in the platoon is defined as

$$\ddot{x}_{i\_des} = \alpha_1\ddot{x}_{i-1} + \alpha_2\ddot{x}_0 + \alpha_3\dot{\varepsilon}_i + \alpha_4\left(\dot{x}_i - \dot{x}_0\right) + \alpha_5\varepsilon_i \tag{6}$$

where

$$\varepsilon_i = x_i - x_{i-1} + l_{i-1} + \text{gap}_{\text{des}} \tag{7}$$

$$\dot{\varepsilon}_i = \dot{x}_i - \dot{x}_{i-1}. \tag{8}$$

Here, $\ddot{x}_0$ and $\dot{x}_0$ are the acceleration and speed of the leader respectively, while $\ddot{x}_{i-1}$ is the acceleration of the front vehicle. Notice that now the distance error term ε_i includes a desired distance gap_{des} which is constant, and it is expressed in meters (5 m by default).

The α_i parameters in Equation (6) are defined as

$$\alpha_1 = 1 - C_1; \quad \alpha_2 = C_1; \quad \alpha_5 = -\omega_n^2 \tag{9}$$

$$\alpha_3 = -\left(2\xi - C_1\left(\xi + \sqrt{\xi^2 - 1}\right)\right)\omega_n \tag{10}$$

$$\alpha_4 = -C_1\left(\xi + \sqrt{\xi^2 - 1}\right)\omega_n. \tag{11}$$

C_1 is a weighting factor between the accelerations of the leader and the preceding vehicle, which we set to 0.5, ξ is the damping ratio, set to 1, and ω_n is the bandwidth of the controller, set to 0.2 Hz as in [1]. The interaction of the CACC with the CC is performed depending on the distance. If a vehicle is farther than 20 m from the front one, the policy is the same as for ACC: $\ddot{x}_{\text{des}} = \min(\ddot{x}_{\text{CC}}, \ddot{x}_{\text{CACC}})$, otherwise $\ddot{x}_{\text{des}} = \ddot{x}_{\text{CACC}}$. In this way it is possible to have two different maximum accelerations, $a_{\max,\text{CC}}$ for the CC (limited for comfort reasons) and the absolute maximum and minimum $a_{\max}$ and $a_{\min}$ representing the vehicle's limit.

The desired acceleration computed by each of the controllers cannot be applied immediately, as there will be actuation lags connected to driveline dynamics. In the simulator, the actuation is modeled through a first order lag (first order low-pass filter), which means that the actual acceleration applied to the car is computed as

$$\ddot{x}_i[n] = \beta \cdot \ddot{x}_{i\_des}[n] + (1 - \beta) \cdot \ddot{x}_i[n-1] \tag{12}$$

$$\beta = \frac{\Delta_t}{\tau + \Delta_t}. \tag{13}$$

The acceleration at simulation step n is computed based on the desired acceleration (computed by the controller) and the acceleration in the previous simulation step. Here, τ is the time constant, i.e., the actuation lag which is set to 0.5 s by default, while Δ_t is the simulation step size in seconds.

5 Changes to the Original Simulator

This section will describe how the original simulators (both Veins and SUMO) have been extended in order to obtain Plexe. This will help you in understanding the structure of the simulator, and in extending it to fit your purposes.

5.1 Changes to SUMO

The main changes has been made to SUMO, in particular by introducing a new car following model which implements the controllers described in Section 4. This can be found in the files `MSCFModel_CC.{h,cpp}` in `plexe-sumo/src/microsim/cfmodels/`. In here we will describe the main concepts behind the implementation, and not the meaning of each line of code. The code itself

Listing 1: Example

```
SUMOReal
MSCFModel_CC::followSpeed(const MSVehicle* const veh, SUMOReal speed, SUMOReal gap2pred, SUMOReal predSpeed,
                          SUMOReal predMaxDecel) const {

    [...]

    if (vars->activeController != MSCFModel_CC::DRIVER)
        return _v(veh, gap2pred, speed, predSpeed, desiredSpeed(veh), MSCFModel_CC::FOLLOW_SPEED);
    else
        return myHumanDriver->followSpeed(veh, speed, gap2pred, predSpeed, predMaxDecel);
}
```

Listing 2: ACC source code

```
SUMOReal
MSCFModel_CC::_acc(const MSVehicle *veh, SUMOReal egoSpeed, SUMOReal predSpeed, SUMOReal gap2pred,
                   SUMOReal headwayTime) const {

    //Eq. 6.18 of the Rajamani book
    return fmin(myAccel, fmax(-myDecel, -1.0 / headwayTime *
               (egoSpeed - predSpeed + myLambda * (-gap2pred + headwayTime * egoSpeed))));

}
```

is pretty well documented, so understanding the concepts should suffice for basic usage and extension.

The main idea of the new car following model is to include both a human behavioral model and the automated controllers. This way the cars in the simulator can both mimic a human, for example for entering or leaving the freeway, and use automated systems when requested. So far the employed human model is Krauss, which is the default one in SUMO. If the car is driven by a human, then the function of the Krauss mobility model are invoked. Otherwise, the methods defined in `MSCFModel_CC.cpp` are used. As an example, in Listing 1 the `followSpeed` function checks if the user has activated an automated controller. If so, the `MSCFModel_CC::_v` function is invoked, otherwise the model invokes the `followSpeed` method of the human behavioral model.

The mobility model implements the controllers described in Section 4, and permits to configure and enable them through the TraCI interface. They are implemented in the `_cc()`, `_acc()`, and `_cacc()` methods, and are invoked by `_v()` depending on which one is enabled. They compute the acceleration to be applied at the next simulation step, clipped between a maximum possible acceleration and deceleration, without considering actuation lags. Listing 2 shows the source code for ACC acceleration.

Within the `_v()` method, the acceleration ($\ddot{x}_{\text{des}}$ in Section 4) is filtered by the `_actuator()` function which needs the acceleration computed at the previous time step. This is stored into the `MSCFModel_CC::VehicleVariables` class. We cannot use a class variable because SUMO instantiates only one mobility model per vehicle type, and the `VehicleVariables` class is meant for keeping track of values which might be different from vehicle to vehicle. Given the filtered acceleration, the model finally computes the speed the car will have at the next time step.

The final step of `_v()` is to store computed information within the `VehicleVariables` class for other usages. Before storing data, the model checks whether data should actually be stored or not. The `_v()` method is indeed invoked more than once per timestep and car. As an example, the lane changing logic invokes this method to understand if the car could get a gain by changing lane. In this case, the function computes the speed the car could reach by changing lane, but the value is used only to understand if the situation on the other lane is better and we must not change the status of the vehicle. The second condition checks who invoked `_v()`. SUMO uses `followSpeed()` to compute the speed to apply to stay behind a vehicle, `freeSpeed()` to compute the speed if the road in front is free, and `stopSpeed()` to compute the speed to use when approaching a non moving obstacle such as a red traffic light. The latter case is simply ignored (see the comments in `stopSpeed()` for a really detailed explanation). The `followSpeed()` and `freeSpeed()` methods might both be called within the same timestep. For this reason, we need to keep track of speed values computed by both methods, and then return the appropriate one to SUMO when needed (i.e., in the `moveHelper()`) method. To know which speed value to return to SUMO in `moveHelper()`, we save the timestep `followSpeed()` has been invoked at. Since SUMO does not invoke `followSpeed()` if no vehicle is in front, we can use this information to decide the correct speed value to return.

As briefly mentioned, another fundamental component of the model is the `VehicleVariables` class. It contains all values that may differ from vehicle to vehicle. As an example, if you consider the ACC described in Section 4, we need to store desired speed, or time headway. Some values are controller related parameters, some others are needed to ensure proper operation, like `controllerFollowSpeed` and `controllerFreeSpeed`, and some are used to provide information to Veins when requested. As an example, radar related information is stored in `radarFrontDistance` and `radarFrontSpeed`.

The final crucial parts of the model are the `get` and `setGenericInformation()` methods. These are the main access points to the model via the TraCI interface. In principle, to send or to retrieve data from a model via TraCI, you would need to change several SUMO and Veins core files, which include `TraCIConstants.h` (on both Veins and SUMO), `TraCIServerAPI_Vehicle.cpp` (on SUMO), and `TraCIScenarioManager.{h,cc}` and `TraCIMobility.h` (on Veins). In `TraCIConstants.h` you need to define a new constant which identifies your new TraCI command, while the other files need to be modified to handle it. The result for a model like `MSCFModel_CC`, which has a huge amount of interactions with Veins, is that the amount of constants defined in the file grows indefinitely. Moreover, it is not desirable to change core files when unneeded.

For such reasons, we define these message passing mechanisms which are able to deal with generic data structures using pointers. The constants are defined within `CC_Const.h` (on both SUMO and Veins). When adding new getters or setters, it is sufficient to add a new constant in `CC_Const.h` and handle it in either `getGenericInformation()` or `setGenericInformation()`.

5.2 Changes to Veins

Veins changes are minor. The main change is in the TraCI functionalities, which have been extended in order to be able to send to and received data from the SUMO model. Then we provide a base set of files which actually handle the simulation, from a network and vehicle dynamics point of view. The files are located in `src/modules/application/platooning`. In there you find:

- `CC_Const.h`: this is a copy of the file described in the previous subsection;
- `UnicastProtocol.{ned,h,cc}`: since Veins IEEE 802.11p model still does not implement unicast communications, this can be used for acknowledged communication when needed. Notice that this is a kind of application-layer unicast, it does not respect MAC layer timings;
- `messages`: in this folder are defined all the `.msg` files which might be needed;
- `protocols`: in here are implemented communication protocols, i.e., the ones that are responsible for beacon dissemination to provide data to the CACC. The folder contains `BaseProtocol` which takes care of some of the duties, such as loading parameters from the `omnetpp.ini` file, logging output data, and so on. This class must be extended to actually implement the beaconing logic. This pattern permits to have only the protocol logic within classes that extend `BaseProtocol`. For example `SimplePlatooningBeaconing` implements static beaconing, and the resulting code is straightforward;
- `apps`: this folder is meant to include files which implement application layer logic. Notice that when referring to application layer we mean the logic that tells vehicles what they should do. For example, if you want to tell a car to move to a specific lane and use the ACC with a desired speed of 130 km/h, then this must be done at the application layer. The design pattern is the same as in the protocol layer. There is a `BaseApp` which simply extracts data out of packets coming from the protocol layer, and updates CACC data via TraCI if such data is coming either from the leader or from the car in front. Then `SimplePlatooningApp` implements the actual logic, i.e., it tells the vehicles to stay on the rightmost lane, to use the controller requested by the user, and makes the leader accelerate and decelerate in a sinusoidal fashion.

Notice that all these files are just examples. You can build your own protocols and application layer dynamics depending on your needs making use of the TraCI functionalities provided by PLEXE.

6 Extending the Simulator

In this section we briefly describe the steps that are required in order to modify the simulator, in particular by adding a new automated controller. We first define the controller that we are going to implement. This is just a fictional controller, there are no guarantees on stability, convergence time, etc. We assume to take into account data from front vehicle only. The control law is defined as

$$\ddot{x}_{i\_\text{des}} = k_d\left(x_{i-1} - x_i - l_{i-1} - 25\,\text{m}\right) + k_s\left(\dot{x}_{i-1} - \dot{x}_i\right) \quad (14)$$

The controller in Equation (14) aims at maintaining an inter-vehicle distance of 25 m and the same speed of the vehicle in front, and has two design gains k_d and k_s.

To implement the controller, we start from SUMO. What we want to do here is to:

1. Implement and make available to the user Equation (14) as a new controller;
2. Be able to specify k_d and k_s via TraCI.

First, we modify `CC_Const.h` by adding our controller to the `ACTIVE_CONTROLLER` enum, and by adding two constants for setting the gains. We call our new controller as `MYCC` (Listing 3). We then need to modify `MSCFModel_CC.h::VehicleVariables` and include the new parameters (Listing 4), and edit `setGenericInformation` to handle their configuration via TraCI (Listing 5).

The final step for SUMO is to implement the control law. We thus define the function `_mycc()` (together with its prototype in `MSCFModel_CC.h`) and we invoke it in `_v()` (Listing 6).

We now need to change Veins to make use of the new controller. We modify the provided example located in `examples/sinPlatoon`. First, we copy `CC_Const.h` from the SUMO source code and place it in `src/modules/application/platooning`. We then edit `BaseApp.ned` and `SimplePlatooningApp.ned` to add the controller parameters (Listing 7). To get the parameters, we modify `BaseApp.{h,cc}` adding class variables, loading them using the OMNeT++ `par()` function, and passing them to SUMO via TraCI in the `initialize()` method (Listing 8).

The final step before running the simulation is to change the `omnetpp.ini` file and `SimplePlatooningApp.cc` to make the simulation use the new controller (Listings 9 and 10). Now the simulation is ready to be started. We have 4 runs, 2 with ACC, one with CACC, and a new one using MYCC. Start the simulation with

```
./run -u Cmdenv -f omnetpp.ini -c Sinusoidal -r 3
```

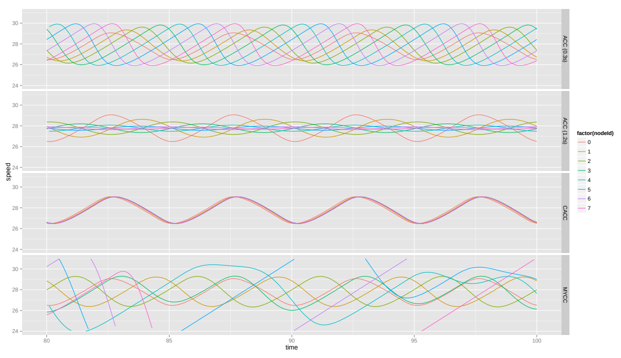
and watch the new controller in action.

To compare the new controller with the ones provided by the simulator, we change `plot.R` located in `examples/sinPlatoon/analysis` (Listing 11). Figure 2 shows the resulting plot, highlighting the instability of MYCC.

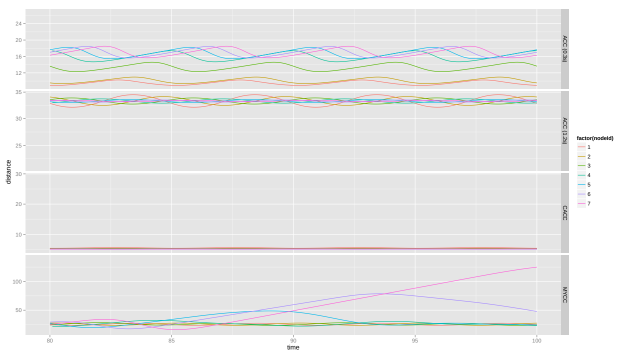
7 Installing Needed Software

7.1 Install OMNeT++

OMNeT++ is the core part of the network simulation framework provided by Veins. In here, we consider version 4.4.1. To build OMNeT you will first need to install some libraries. On a Linux system, type



(a) Speed in time



(b) Distance in time

Figure 2: Plots of speeds and distances as function of time for different controllers configurations obtained with the simulator, including MYCC.

```
sudo apt-get install bison flex tk8.5-dev
```

while on Mac OS X you can install them with

```
sudo port install bison flex tk
```

Then you will need to add the OMNeT++ `bin` folder to your path, by adding to your `.bash_rc` or `.profile`

```
export PATH=$PATH:$HOME/src/omnetpp-4.4.1/bin
```

After downloading OMNeT (source plus IDE) from the official website in your home folder, extract and compile it with

```
cd
tar xzf omnetpp-4.4.1-src.tgz
cd omnetpp-4.4.1
./configure
make
```

7.2 Install R

R[6] is a powerful statistical framework which can be used to parse, process, and plot data obtained through OMNeT++ simulations. The plots in Figures 1a and 1b have been obtained with a few lines of code. To install R, open a terminal and type

```
sudo apt-get install r-base
```

on a Linux machine or

```
sudo port install R
```

on Mac OS X. You will then need to install OMNeT++ package for R, `ggplot2`, and `reshape`. To install `ggplot2` and `reshape`, open the `R` console and simply type

```
install.packages(c('ggplot2', 'reshape'))
```

Once installation is complete, exit the console by hitting CTRL+D twice. Download the OMNeT++ R package[7] and then install it by typing

```
R CMD INSTALL /path/to/download/omnetpp_0.2-1.tar.gz
```

on your terminal. Your `R` environment should now be correctly set up.

References

[1] Pedro Fernandes. *Platooning of IVC-Enabled Autonomous Vehicles: Information and Positioning Management Algorithms, for High Traffic Capacity and Urban Mobility Improvement.* Phd thesis, University of Coimbra, Portugal, April 2013.

[2] J. Ploeg, B.T.M. 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)*, pages 260–265, Washington, DC, October 2011. IEEE.

[3] R. Rajamani. *Vehicle Dynamics and Control*. Springer, second edition, 2012.

[4] Christoph Sommer, Reinhard German, and Falko Dressler. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, 2011.

[6]`http://www.r-project.org`

[7]`https://github.com/downloads/omnetpp/omnetpp-resultfiles/omnetpp_0.2-1.tar.gz`

8 Known Issues

Q: *SUMO linking fails with undefined reference to symbol glGetBooleanv, pthread_mutex_setattr, or gluTessProperty*

A: This issue has been found when compiling SUMO on Ubuntu 14.04. To solve the problem, use the following configuration command before compiling:

```
LIBS="-pthread -lGLU -lGL" ./configure
```

Q: *OMNeT++ compilation fails with error "abspath.cc error: 'getcwd' was not declared in this scope"*

A: There is a missing include directive in abspath.cc. Fix this by adding the directive in abspath.cc

```
#include <unistd.h>
```

Q: *ggplot2 is not available for a particular R version*

A: Installation might fail if you own an old version of R. For example, if you are using Ubuntu 12.04 LTS, the R version you will be installing from apt-get will be 2.14.1, and some required dependencies are not available anymore. To overcome the problem, you can install the latest version of R with

```
sudo add-apt-repository ppa:marutter/rdev
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install r-base
```

Q: *OMNeT++ compilation fails with an include or a linking error on Mac OS X for* `libtk`

A: This can happen if you installed required libraries using MacPorts. To let the compiler and the linker know about the locations of MacPorts installed libs, configure OMNeT++ in the following way

```
CFLAGS='-I/opt/X11/include -I/opt/local/include'\
LDFLAGS='-L/opt/X11/lib -L/opt/local/lib'\
./configure
```

9 For the Impatient

9.1 Linux

This section contains a list of commands that you can copy-paste on your terminal to download, configure, and build the simulator on a brand new system, for example a fresh Ubuntu installation. Before using the commands, be sure to change your `PATH` in your `.bashrc` file, and then source it:

```
export PATH=$PATH:~/src/omnetpp-4.4.1/bin
export PATH=$PATH:~/src/plexe-sumo/bin
export TCL_LIBRARY=/usr/share/tcltk/tcl8.5
```

```
cd
source .bashrc
sudo apt-get install build-essential bison flex zlib1g-dev tk8.5-dev openjdk-6-jre autoconf libtool libproj-dev libgdal-dev \
                     libfox-1.6-dev libxerces-c-dev r-base

mkdir -p src
wget http://omnetpp.org/download/release/omnetpp-4.4.1-src.tgz
tar xf omnetpp-4.4.1-src.tgz
cd omnetpp-4.4.1
./configure
make

cd ..
wget http://plexe.car2x.org/download/plexe-veins.tar.bz2
tar xf plexe-veins.tar.bz2
cd plexe-veins
make -f makemakefiles MODE=release
make MODE=release

cd ..
wget http://plexe.car2x.org/download/plexe-sumo.tar.bz2
tar xf plexe-sumo.tar.bz2
cd plexe-sumo
make -f Makefile.cvs
LIBS="-pthread -lGLU -lGL" ./configure
make

cd
wget https://github.com/downloads/omnetpp/omnetpp-resultfiles/omnetpp_0.2-1.tar.gz
mkdir -p R
echo ".libPaths(c(.libPaths(), '$HOME/R'))" > $HOME/.Rprofile
R -e "install.packages('ggplot2', lib='$HOME/R', repos='http://mirrors.softliste.de/cran/')"
R -e "install.packages('reshape', lib='$HOME/R', repos='http://mirrors.softliste.de/cran/')"
R CMD INSTALL omnetpp_0.2-1.tar.gz --lib=$HOME/R
```

9.2 Mac OS X

On a Mac OS X system, you will need to install XCode (to get build tools like `gcc`) and MacPorts[8] to install required libraries and software. After installing these programs, edit your `.profile` file in you home directory and add

```
export PATH=$PATH:~/src/omnetpp-4.4.1/bin
export PATH=$PATH:~/src/plexe-sumo/bin
```

```
cd
source .profile
sudo port install bison flex zlib tk autoconf libtool proj gdal fox xercesc R wget

mkdir -p src
wget http://omnetpp.org/download/release/omnetpp-4.4.1-src.tgz
tar xf omnetpp-4.4.1-src.tgz
cd omnetpp-4.4.1
./configure
make

cd ..
wget http://plexe.car2x.org/download/plexe-veins.tar.bz2
tar xf plexe-veins.tar.bz2
cd plexe-veins
make -f makemakefiles MODE=release
make MODE=release

cd ..
wget http://plexe.car2x.org/download/plexe-sumo.tar.bz2
tar xf plexe-sumo.tar.bz2
cd plexe-sumo
export CPPFLAGS="$CPPFLAGS -I/opt/local/include"
export LDFLAGS="$LDFLAGS -L/opt/local/lib -framework GLUT -framework OpenGL"
make -f Makefile.cvs
./configure --with-fox=/opt/local --with-proj-gdal=/opt/local --with-xerces=/opt/local
make

cd
wget https://github.com/downloads/omnetpp/omnetpp-resultfiles/omnetpp_0.2-1.tar.gz
mkdir -p R
echo ".libPaths(c(.libPaths(), '$HOME/R'))" > $HOME/.Rprofile
R -e "install.packages('ggplot2', lib='$HOME/R', repos='http://mirrors.softliste.de/cran/')"
R -e "install.packages('reshape', lib='$HOME/R', repos='http://mirrors.softliste.de/cran/')"
R CMD INSTALL omnetpp_0.2-1.tar.gz --lib=$HOME/R
```

[8] `http://www.macports.org`

A Listings

This section includes all the listings showing which part of the simulator needs to be modified in order to implement a new controller, as done in Section 6.

Listing 3: Changes to CC_Const.h

```
enum ACTIVE_CONTROLLER
{DRIVER = 0, ACC = 1, CACC = 2, FAKED_CACC = 3, MYCC = 4}

[...]

#define CC_SET_CACC_C1     0x06    //C1
#define CC_SET_ENGINE_TAU  0x07    //engine time constant

#define CC_SET_MYCC_KD     0x08    //k_d for new controller
#define CC_SET_MYCC_KS     0x09    //k_s for new controller
```

Listing 4: Changes to VehicleVariables

```
class VehicleVariables : public MSCFModel::VehicleVariables {
public:
    VehicleVariables() : egoDataLastUpdate(0), egoSpeed(0), egoAcceleration(0), egoPreviousSpeed(0),
        [...]
        caccAlpha3(-1), caccAlpha4(-1), caccAlpha5(-1), engineAlpha(-1), engineOneMinusAlpha(-1),
        myccKd(1), myccKs(1) {

    [...]

    /// @brief controller related parameters
    double caccXi;
    double caccOmegaN;
    double caccC1;
    double caccAlpha1, caccAlpha2, caccAlpha3, caccAlpha4, caccAlpha5;
    double engineTau, engineAlpha, engineOneMinusAlpha;
    /// @brief parameters for MYCC
    double myccKd, myccKs;
};
```

Listing 5: Handling passage of parameters

```
[...]
case CC_SET_ENGINE_TAU: {
    vars->engineTau = *(double*)content;
    recomputeParameters(veh);
    break;
}
case CC_SET_MYCC_KD: {
    vars->myccKd = *(double*)content;
    break;
}
case CC_SET_MYCC_KS: {
    vars->myccKs = *(double*)content;
    break;
}
default: {
    break;
}
```

Listing 6: Implementation of _mycc

```
SUMOReal
MSCFModel_CC::_v([...]) {
    [...]
    case Plexe::MYCC:

        ccAcceleration = _cc(veh, egoSpeed, vars->ccDesiredSpeed);
        caccAcceleration = _mycc(veh, egoSpeed, vars->frontSpeed, gap2pred);
        controllerAcceleration = fmin(ccAcceleration, caccAcceleration);
        break;

    case Plexe::DRIVER:

        std::cerr << "Switching to normal driver behavior still not implemented in MSCFModel_CC\n";
        assert(false);
        break;
   [...]
}

[...]

SUMOReal
MSCFModel_CC::_mycc(const MSVehicle *veh, SUMOReal egoSpeed, SUMOReal predSpeed, SUMOReal gap2pred) const {
    VehicleVariables* vars = (VehicleVariables*)veh->getCarFollowVariables();
    return fmin(myAccel, fmax(-myDecel, vars->myccKd * (gap2pred - 25) + vars->myccKs * (predSpeed - egoSpeed)));
}
```

Listing 7: Changes to ned files

```
[...]
double caccC1;
double engineTau @unit("s");
double myccKd;
double myccKs;
```

Listing 8: Loading parameters from omnetpp.ini

```
void BaseApp::initialize(int stage) {

    BaseApplLayer::initialize(stage);

    if (stage == 0) {

        //init class variables
        [...]
        engineTau = par("engineTau").doubleValue();
        myccKd = par("myccKd").doubleValue();
        myccKs = par("myccKs").doubleValue();

        [...]
        traci->commandSetGenericInformation(traci->getExternalId(), CC_SET_ENGINE_TAU, &engineTau, sizeof(double));
        traci->commandSetGenericInformation(traci->getExternalId(), CC_SET_MYCC_KD, &myccKd, sizeof(double));
        traci->commandSetGenericInformation(traci->getExternalId(), CC_SET_MYCC_KS, &myccKs, sizeof(double));

    }

}
```

Listing 9: Using new controller in SimplePlatooningApp.cc

```
void SimplePlatooningApp::initialize(int stage) {

    BaseApp::initialize(stage);

    if (stage == 1) {

        [...]
        if (strcmp(strController, "ACC") == 0) {
            controller = Plexe::ACC;
        }
        else if (strcmp(strController, "CACC") == 0) {
            controller = Plexe::CACC;
        }
        else {
            controller = Plexe::MYCC;
        }
        [...]
```

Listing 10: Changes to omnetpp.ini

```
#use ACC or CACC
**.dummycontroller=${controller=0,0,1,2}
*.node[*].appl.controller = ${sController = "ACC", "ACC", "CACC", "MYCC" ! controller}
#ACC time headway. note that time headway for CACC is ignored
**.dummyheadway=${headway=0.3,1.2,0.3,0.3!controller}s
*.node[*].appl.accHeadway = ${headway}s
[...]
#make the leader accelerate and decelerate with a sinusoidal trend. set to 0 for constant speed
*.node[*].appl.leaderOscillationFrequency = 0.2Hz
#set parameters for MYCC
*.node[*].appl.myccKd = 0.7
*.node[*].appl.myccKs = 1
[...]
```

Listing 11: Changes to plot.R

```
accCloseData <- prepare.vector('../results/Sinusoidal_0_0.3_0.vec')
accFarData <- prepare.vector('../results/Sinusoidal_0_1.2_0.vec')
caccData <- prepare.vector('../results/Sinusoidal_1_0.3_0.vec')
ccData <- prepare.vector('../results/Sinusoidal_2_0.3_0.vec')

#add a column to distinguish them before merging
accCloseData$controller <- "ACC (0.3s)"
accFarData$controller <- "ACC (1.2s)"
caccData$controller <- "CACC"
ccData$controller <- "MYCC"

#merge all data together
allData <- rbind(accCloseData, accFarData, caccData, ccData)
```