Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Control Engineering

Simulating the impact of prioritization of emergency vehicles at traffic light controlled junctions on the other traffic

Master's thesis

VÍT OBRUSNÍK

Supervisor: Ing. Zdeněk Hurák, Ph.D.

Prague, May 2019

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Obrusník Vít**	Personal ID number:	**434659**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Control Engineering**		
Study program:	**Cybernetics and Robotics**		
Branch of study:	**Cybernetics and Robotics**		

II. Master’s thesis details

Master’s thesis title in English:

Simulating the impact of prioritization of emergency vehicles at traffic light controlled junctions on the other traffic

Master’s thesis title in Czech:

Simulace dopadu preference vozidel integrovaného záchranného systému na světelných křižovatkách na další dopravu

Guidelines:

The key focus of this project is on a microscopic simulation of traffic in an urban area comprising a few adjacent traffic light controlled junctions and an emergency vehicle driving through them. Two modes of an emergency vehicle driving through the crossroads should be considered: first, the emergency vehicle is just blue-lighting and sirening its way through the crossroads while ignoring the traffic lights altogether; second, the emergency vehicle uses a radio vehicle-to-infrastructure (V2I) communication system to inform the traffic light controller about its approaching and to request a priority for passing. Simulations based on the data from real traffic are to be used to conclude if the worries of some practitioners that while the latter mode might save a few seconds for the emergency vehicle, the other traffic takes much longer to recover after the emergency vehicle is gone, are justified. The proposed simulators are SUMO for the traffic and OMNeT++ for the communication. The empirical data from the induction loops and traffic lights for a given area will be provided.

Bibliography / sources:

[1] H. Noori, “Modeling the impact of VANET-enabled traffic lights control on the response time of emergency vehicles in realistic large-scale urban area,” in 2013 IEEE International Conference on Communications Workshops (ICC), 2013, pp. 526–531.
[2] T. Bellemans, B. De Schutter, and B. De Moor, “Models for traffic control,” Journal A, vol. 43, no. 3–4, pp. 13–22, 2002.
[3] C. Sommer and F. Dressler, Vehicular Networking, 1 edition. Cambridge, United Kingdom: Cambridge University Press, 2015.

Name and workplace of master’s thesis supervisor:

doc. Ing. Zdeněk Hurák, Ph.D., Department of Control Engineering, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **15.02.2019** Deadline for master's thesis submission: **24.05.2019**

Assignment valid until:
by the end of summer semester 2019/2020

doc. Ing. Zdeněk Hurák, Ph.D.	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor’s signature	Head of department’s signature	Dean’s signature

Abstract

In this thesis, a microscopic traffic simulation scenario is created containing four adjacent traffic light controlled junctions. The main task is to simulate prioritization techniques for emergency vehicles going through them. Three modes are modelled: first, the emergency vehicle has no preference on traffic lights and needs to rely on sirens and bluelight, second, the emergency vehicle signals preemption request after crossing certain distance from the junction and third, the emergency vehicle beacons its position and traffic light controller decides about the time to start preference, based on actual traffic situation. Outputs from simulations are then used to compare the benefits for the emergency vehicle and the impact on the other traffic. The area of the simulation scenario is a part of Brno city (the Czech Republic). Induction loop measurements were provided for traffic demand modelling. Static traffic light signal plans for the area were used.

Abstrakt

V této práci je vytvořen scénář mikroskopické simulace dopravy obsahující čtyři po sobě jdoucí křižovatky řízené semafory. Hlavní náplní je simulovat metody preference pro vozidla záchranných služeb projíždějících křižovatkami. Simulovány jsou tři módy: první, záchranné vozidlo nedostává žádnou preferenci na semaforech a spoléhá se jen na sirény a majáky, druhý, po projetí určité vzdálenosti od křižovatky se záchranné vozidlo přihlásí o preferenci, třetí, záchranné vozidlo signalizuje svou pozici a řadič křižovatky rozhodne o čase spuštění preference v závislosti na aktuálním stavu dopravy. Z výstupních dat simulací je vyhodnocena výhodnost jednotlivých módu pro záchranné vozidlo a je analyzován dopad na okolní dopravu. Simulovaná oblast je část města Brno (Česká republika). Pro namodelování hustoty dopravy byla poskytnuta data z měření indukčních smyček. V simulaci byly použity statické signální plány semaforů pro danou oblast.

Declaration

I declare that I wrote the presented thesis on my own and that I cited all the used information sources in compliance with the Methodical instructions about the ethical principles for writing an academic thesis.

Prague, May 2019

Vít Obrusník

Acknowledgements

I am very grateful to my supervisor, Zdeněk Hurák, for letting me work in his research group on amazing projects. I am very proud of this experience and I have learned a lot of things on the way. I also thank my colleague, Loi Do, because he informed me about the opportunity and without him, I would probably not be there. This work would not be possible without Ivo Herman, our project partner. Therefore I thank him dearly for all the long calls we had. I also thank Ondřej Miklóš from BKOM for his contribution. I must thank Jan Přikryl for his time that he spent discussing the project with me. It really helped me a lot. I would like to express my gratitude also to people who spend their time developing open source software SUMO and Veins, which I used throughout my thesis.

Many thanks belong to my family for their support during all these years. The biggest thank you belongs to my girlfriend, Daria, she is just great

Contents

Acronyms

API Application programming interface.

ATR Automatic Traffic Recorder.

CSV comma-separated values.

CT cycle time.

DSRC Dedicated short-range communication.

EV emergency vehicle.

FCD Floating car data.

GPS The Global Positioning System.

GUI Graphical user interface.

IDE Integrated development environment.

IDM Intelligent driver model.

LTE Long-Term Evolution.

NED Network description language.

OBU On board unit.

OMNeT++ Object-oriented modular discrete event network simulation framework.

OSM Open Street Map.

RSU Roadside Unit.

SUMO Simulation of Urban MObility software package.

TCP Transmission control protocol.

TLC traffic light controller.

TLJ traffic-light controlled junction.

TraCI Traffic Control Interface.

V2I	vehicle to infrastructure.
V2V	vehicle to vehicle.
V2X	vehicle to anything.
VANET	Vehicular ad-hoc network.
Veins	Vehicles in network simulator.
XML	Extensible Markup Language.

Chapter 1

Introduction

In this thesis, I deal with simulating the traffic on *traffic-light controlled junctions* (TLJs). In particular, I focus on scenarios when *emergency vehicles* (EVs) (ambulance, fire brigade or police) are rushing through the TLJs to the accident site. These scenarios can be commonly witnessed in all major cities on daily basis: first you hear the sirens, then you see the EV equipped with a blue light device, coming fast from nowhere, approaching a junction, other vehicles trying to clear the way.

Clearly, it is necessary that EVs gets to the accident site as soon as possible. Some research was done to show the correlation between mortality rate and emergency services response time [1, 2], authors in [3] claim that 75% of deaths caused by car accidents happen in the first hour after the accident, sometimes referred to as *golden hour*. In the Czech Republic and in many other countries, EVs have permission to go through a red light or cross and overtake through double line. EVs drivers often need to exploit this permissions. However, there are severe drawbacks:

- Passing the TLJ through the red lights is dangerous. EV drivers have much higher chance to participate in an accident [4].
- EVs drivers going through the red lights are responsible for casualties if they cause an accident.

The study [5] shows that the most accidents of EVs happens at controlled and uncontrolled intersections. The most frequent reason is crossing the junction at red traffic lights (32%). See the evaluation of 189 reported accidents of EVs in Germany throughout the years of 2009 and of 2015 in Fig. 1.1.

One way to mitigate the risks stated above and improve the response time of EVs is to give EVs preference on TLJs (note: I will use the terms *preference*, *prioritization* and *preemption* interchangeably throughout the text). The basic concept is to switch the traffic lights to green in a direction from which the EV is approaching. The switch should take place soon enough so other vehicles can empty the junction before the arrival of the emergency vehicle. This allows the EV to go through the junction faster and safer while also mitigating the risks stated above.

This topic is motivated by real world problems of traffic engineers. A few companies from industry introduced the problems and actively participated in the research, especially by providing the data and the domain knowledge. The work on this thesis was done in the research group *Advanced Algorithms for Control and Communications (AA4CC)*, Department of Control Engineering, Faculty of Electrical Engineering at Czech Technical University in Prague.

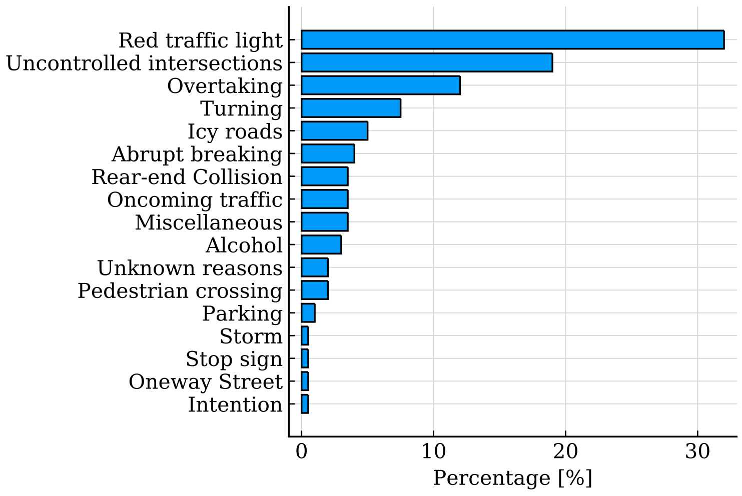
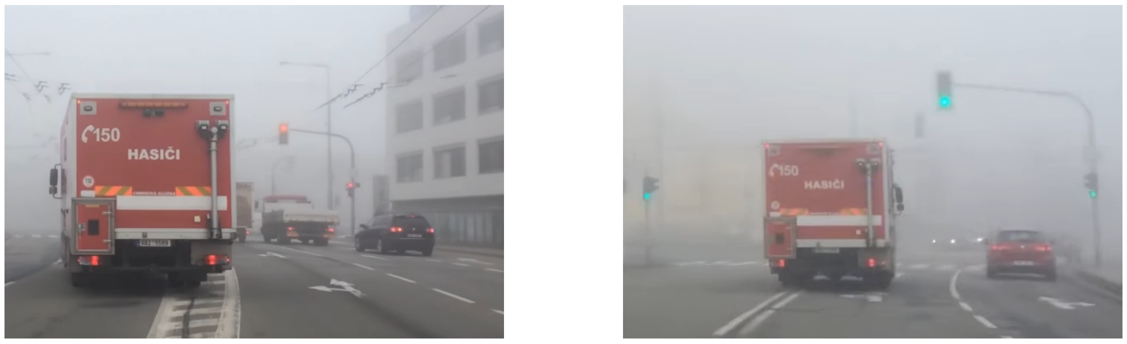


Figure 1.1: *Reasons of accidents of EVs. Data taken from [5, 6].*

1.1 Problem definition

The prioritization of EVs is used in some cities in the Czech Republic. Prioritization comes with its own unique challenges:

- It is argued by some practitioners (based on their empirical observations) that while the preference might save a few seconds for the EV, the other traffic takes longer to recover after the EV is gone. Taking into consideration that the number of ambulances driving through the city at a given moment is typically very high, the worries of creating unnecessary traffic jams are seriously blocking further adoption of *vehicle to anything (V2X)* communication enabled priority granting adaptive control schemes.
- Current method of prioritization requires a virtual area around a TLJ to be defined. This area serves as a border for the EV. When the border is crossed, the preference is activated. This is far from ideal because the preemption phase might be switched on too late or too soon. Also, it takes experienced traffic engineer and significant amount of time to define the border.



(a) *TLJ without preference* (b) *TLJ with preference*

Figure 1.2: *(a) Driver needs to take care about the vehicles from the opposite direction and also pedestrians crossing the street. (b) Preference is a big help for the EV driver. Taken from* `http://www.firebrno.cz/preference-vozidel-hzs-zelena-vlna-v-brne`

1.2 Aim of this work

The aim of this work is to tackle both problems described in 1.1. Thus, to validate the concerns of traffic engineers about the impact on the other traffic, and to propose a better solution. In order to do this, I created a simulation scenario of a part of Brno city (Czech Republic) with four adjacent TLJs. The traffic demand is configured based on the measurements from the induction loops. Traffic lights operate in static signal plan.

To compare different modes of preference, one needs to have a program which simulates the preference of EVs by today means. The development of such program is described in detail. Then I proposed and simulated novel preference method which eliminates the need of defining the virtual boundary around the TLJ. Of course, some parameters for consideration by traffic engineers stay.

To sum up the above, I have considered three modes of an EV driving through the TLJ:

1. the EV is just blue-lighting and sirening its way through the junction while ignoring the traffic lights altogether
2. current method of prioritization, which is based on *Long-Term Evolution (LTE)* communication, *The Global Positioning System (GPS)*, and an area defined by traffic engineers around a junction
3. newly proposed method of prioritization is modelled, where the emergency vehicle uses a radio vehicle to infrastructure communication system and GPS to inform the *traffic light controller (TLC)* about its approaching and to signal a request for a priority, TLC then decides about the time when to start a priority phase.

After simulating these modes I will study the trip times and waiting times of vehicles in the simulations. The results are then compared and discussed.

1.3 Structure of the thesis

This thesis is structured to 6 chapters. In Chapter 2, I give a brief introduction to traffic modelling and then I discuss the tools that are used for simulations. In Chapter 3, a process of creating and validating the simulation scenario is described. Chapter 4 finally deals with preference modes for EVs. Analysis of the outputs is discussed in Chapter 5 and Chapter 6 concludes the work.

Chapter 2

Traffic modelling

When studying a phenomena, researchers tend to look for mathematical models that provide reasonable description of the problem at hand. These models should be detailed enough and in the same time should be feasible to analyse or simulate. Studying the traffic dynamics is not different. In this chapter, I give a brief introduction to modelling the traffic. By traffic, I mean vehicles (usually cars) going through the network of roads. We could also switch vehicles to other means of transport (e.g. pedestrians or trains) and switch roads to other edge types (e.g. sidewalks or rails), and we would still be modelling the traffic.

In the first two parts of the chapter, I describe several traffic modelling approaches. To solve the problems described in 1.1, it is necessary to have a good simulation tool and to understand very well how the tool works. Thus, I describe the chosen simulator in detail in the third part. How to model vehicles in network is the topic of the last section.

2.1 Models classification

It is important to have some inside knowledge when dealing with mathematical models in simulations. There is a variety of ways to classify traffic models:

- Mathematical structure (Partial Differential Equations, Coupled Ordinary Differential Equations, Cellular automatons and more)
- Randomness (deterministic vs. stochastic)
- Scale of the independent variables (continuous vs. discrete)
- Conceptional foundation (heuristic vs. first principles)
- Level of details (Macroscopic, Mesoscopic, Microscopic, Submicroscopic)
- and many more

See [7] for a detailed description, [8] for an overview of development of traffic models in the last century and [9] for a closer look at some of them with an example. Classification by the level of details (or one can say level of aggregation) is the most important with respect to my work so it makes sense to elaborate a bit more on it.

2.1.1 Macroscopic

In principle, macroscopic models describe traffic flows similarly to the way fluid dynamics describes fluids going through pipes. If we are not interested in particular participants of traffic but we are interested in behaviour of traffic streams on bigger scale (e.g. the whole city), then macroscopic models are good tools. Macroscopic models are good match for traffic (control) engineers because they are usually interested in variables such as density $\rho(x,t)$ and flow $q(x,t)$ of vehicles in space and time. Detailed elaboration on macroscopic models is given in [9].

2.1.2 Mesoscopic

Mesoscopic models are something between macroscopic and microscopic models. That means, they are better suited for high level scenarios, yet still keeps good level of details about vehicles. These models are rarely used.

2.1.3 Microscopic

As opposite to macroscopic models, microscopic models treat every single participant of traffic stream. As expected, if a large number of traffic participants is modelled, the resulting behaviour resemble the macroscopic behaviour. However, for fewer vehicles, we can observe more realistic results than if we model few vehicles with macroscopic models. This level of details comes with much more demanding computational cost. Due to the nature of the problems 1.1, microscopic simulation is the best choice. Some researchers implement microscopic models as cellular automaton [10]. Such implementations are better suited for freeway simulations.

2.1.4 Submicroscopic

Models which go into even more details and model the interaction of wheels with the road or properties of the engine, are called submicroscopic.

2.2 Car-following models

Car-following models are the core of microscopic simulations. These models describe the behaviour of a single vehicle α in terms of dynamic variables: position $x_\alpha(t)$, speed $v_\alpha(t)$ and acceleration $\dot{v}_\alpha(t)$. Interactions with other vehicles and road segments are taken into account. In continuous-time models, the driver's response is governed by a set of coupled ordinary differential equations:

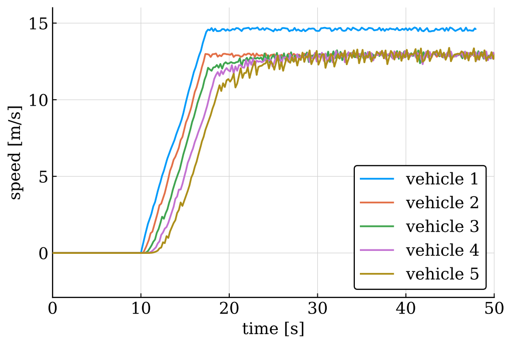
$$\dot{x}_\alpha(t) = v_\alpha(t), \tag{2.1}$$

$$\dot{v}_\alpha(t) = a_{mic}(s_\alpha, v_\alpha, v_l), \tag{2.2}$$

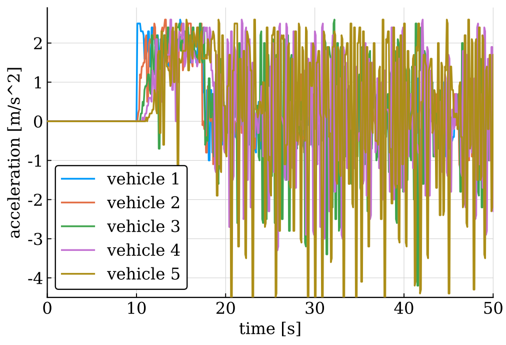
where s_α is the distance-gap between vehicles (bumper-to-bumper) and v_l is the speed of the leading vehicle. The term $a_{mic}(s_\alpha, v_\alpha, v_l)$ is an acceleration model that further specifies driving strategies of car-following models. In my work I used two types shown below, others are usually used for specific purposes. I tried to simulate a queue of 5 vehicles starting off with both of the models.

Krauss

Krauss model is an extension of older Gibbs-model [11, 12]. It is characteristic for noisy acceleration which results in slightly jerky movement of vehicles.



(a) *Speed characteristic of Krauss model*

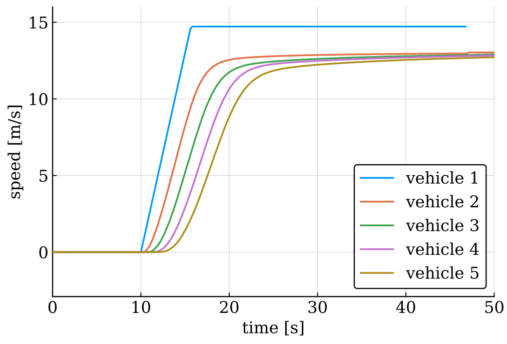


(b) *Acceleration characteristic of Krauss model*

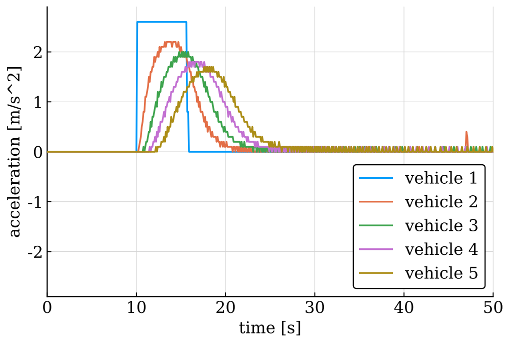
Figure 2.1: *Five simulated vehicles in a queue governed by Krauss model starts off at time 10.*

Intelligent Driving Model

Intelligent driver model (IDM) exhibits smoother movement behaviour than Krauss model. This is the model that I will be using later in the work. Newer modifications of IDM also exist.



(a) *Speed characteristic of IDM*



(b) *Acceleration characteristic of IDM*

Figure 2.2: *Five simulated vehicles in a queue governed by IDM starts off at time 10.*

2.3 SUMO

Simulation of Urban MObility software package (SUMO) [13, 14] is a free and open source microscopic, multi-modal traffic simulation suite available since 2002 developed by DLR - Institute of Transportation Systems. SUMO is programmed in C++ and Python programming languages. Since 2019, SUMO is licensed under Eclipse[1] and is a part of OpenMobility project[2]. The vision of OpenMobility is: "Advancing Simulation Environments for Transport Applications".

Because this text might be used as a reference for some of my peers, I will include some practical tips and links to study materials. After reading this section, you should have a rough understanding of how SUMO works. Invaluable resource for learning is the Wiki[3], where you can find the installation manual, tutorials and documentation.

2.3.1 Components

The microscopic dynamics of vehicles are determined by the interplay of several models (from Erdmann, 2014, [15]):

1. Car-following model: determines the speed of vehicle in relation to the vehicle ahead of it.
2. Intersection model: determines the behaviour of vehicles at different types of intersections in regards to the right-of-way rules, gap acceptance and avoiding junction blockage.
3. Lane-changing model: determines lane choice on multi-lane roads and speed adjustments related to lane changing.

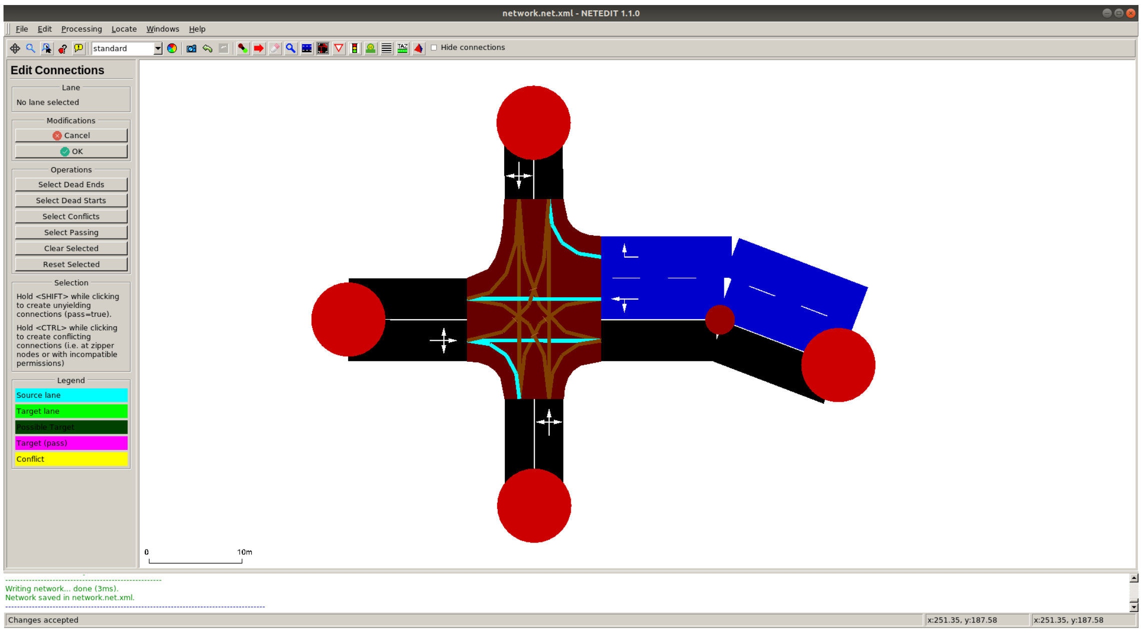
The behaviour is also influenced by randomization, SUMO uses well known Mersene Twister pseudorandom number generator [16], the seed can be set in *config* file (Section 2.3.3). You can use different seeds to slightly variate the output of simulation runs.

Two mandatory inputs must be provided to run a simulation: a road *network* and traffic demand *routes*. From now on, when writing about SUMO specific files or concepts, I will write the terms in *italics* to distinguish them from other meanings. All of the inputs are written as files in *Extensible Markup Language (XML)* format.

Network

A *network* file can be built using *Graphical user interface (GUI)* program *NETEDIT* (Fig. 2.3) or using command line tools *NETGENERATE* or *NETCONVERT*. *Network* is basically a graph as known from graph theory. It is defined by nodes (*nodes* or *junctions*) and edges (*edges*). *Edges* are divided into *lanes* that are connected to other *lanes* via *connections*. The shape of an *edge* can be arbitrarily changed by *geometry points*.

[1]Eclipse Public License - v 2.0: `https://www.eclipse.org/legal/epl-v20.html`
[2]OpenMobility: `https://openmobility.eclipse.org/`
[3]SUMO Wiki: `https://sumo.dlr.de/wiki/Simulation_of_Urban_MObility_-_Wiki`



No lane selected

Hold <SHIFT> while clicking to create unyielding connections (pass=true).

Hold <CTRL> while clicking to create conflicting connections (i.e. at zipper nodes or with incompatible permissions)

Source lane
Target lane
Possible Target
Target (pass)
Conflict

```
Writing network... done (3ms).
Network saved in network.net.xml.
```

Changes accepted   x:251.35, y:187.58   x:251.35, y:187.58

Figure 2.3: *Example of creating trivial network in NETEDIT GUI program. The network has five nodes and eight edges. One of the edges is selected (in blue) and its shape is changed with geometry point (smaller red circle). The selected edge is divided into two lanes, other edges have only one lane. The connections between lanes are visible inside the middle junction.*

Routes

A *route* file consists of *vehicle type* or *vType* (what kind of vehicle will be departed), *vehicle* (when will the vehicle depart) and *vehicle route* (where will the vehicle go). Of course, the definitions can be split into more files. *Vehicle route* is a path in the graph (*network*) that is defined by a sequence of consecutive *edges*. Instead of single *vehicles*, SUMO allows to define repeated vehicles, this is called *flow*. Here is an example of simple *rou* file with one *vehicle* and *flow*:

```
<routes>
      <!-- vehicle types -->
      <vType id="typePassenger" vClass="passenger" carFollowModel="IDM"↘
           minGap="2.5"/>
      <!--vehicle routes-->
      <route id="left-right" edges="left0A0 A0right0"/>
      <!--vehicles -->
      <vehicle id="v1" type="typePassenger" route="left-right" depart="↘
          0"/>
      <!--flows-->
      <flow id="f1" type="typePassenger" begin="1" end="200" period="1"↘
           route="left-right"/>
</routes>
```

In the example I have chosen my *vehicle type* to behave according to *Intelligent driving model* (2.2). If no `carFollowModel` is specified, the default one is chosen, which is *Krauss* (2.2). Other parameters can be found on Wiki page[4]

Additional files

Apart from mandatory inputs, there are various other entities that can be specified as *additional* files. From SUMO Wiki:

- infrastructure related things: traffic light programs, induction loops and bus stops
- additional visualization: points of interests (POIs) and polygons (i.e. rivers and houses)
- dynamic simulation control structures: variable speed signs and rerouters
- demand related entities: vehicle types and routes

Here is an example of *additional* file that defines traffic light logic (*tll*) for the junction from Fig. 2.3. This can be written be manually or *NETEDIT* GUI can be used to generate the file. Note that the number of characters in *state* corresponds to the number of *connections* at the *junction*.

```
<additional>
   <tlLogic id="A0" type="static" programID="0" offset="0">
      <phase duration="42" state="GGgrrrGGgrrr"/>
      <phase duration="3" state="yyyrrryyyrrr"/>
      <phase duration="42" state="rrrGGgrrrGGg"/>
      <phase duration="3" state="rrryyyrrryyy"/>
   </tlLogic>
</additional>
```

2.3.2 Detectors and devices

Detectors play an important role in my work. They can simulate induction loop detectors. Such detectors are called *E1 detectors*[5] and they can be created in *NETEDIT* or manually written into *additional* XML file. In the toolset of SUMO, you can find a script *generateTLSE1Detectors.py* for generating *E1 detectors* for each *junction* in the supplied *network* file. Another type is *E2 detector* which is used to model computer vision system that observes traffic situation on a larger scale. Here is an example of definition of two types of *detectors* on the same *lane*

[4]Definition of *Vehicles*, *Vehicle types*, and *Routes*: https://sumo.dlr.de/wiki/Definition_of_Vehicles,_Vehicle_Types,_and_Routes

[5]E1 Induction Loops Detectors: https://sumo.dlr.de/wiki/Simulation/Output/Induction_Loops_Detectors_(E1)

```
<additional>
   <e1Detector id="det_e1" lane="e1_0" pos="30.00" freq="600.00" file="↘
      output/det.xml"/>
   <laneAreaDetector id="det_e2" lane="e1_0" freq="600" length="10" ↘
      file="output/det_e2.xml"/>
</additional>
```

Device is a specification for *vehicle type* that it is supplied with particular piece of equipment. I used two types throughout my work:

1. *Device.fcd* which enables *vehicles* to collect *Floating car data (FCD)* data.
2. *Device.bluelight* which is used to model EVs.

2.3.3 Running the simulation

The number of files is growing rapidly when building bigger simulation scenario so there is a concept of *config* file where you can specify all the other files at once. The filename usually ends with *.sumocfg*. Similar *config* files work for other parts of the SUMO suite that require bigger number of inputs.

SUMO is mostly used from command line. The minimal command may look like this, it runs the simulation without visual output (suitable for batch simulation runs). To start with GUI, replace `sumo` with `sumo-gui`:

```
sumo -n network.net.xml -r routes.rou.xml --additional-files file.add.xml
sumo -c config.sumocfg
```

Both commands do the same thing assuming that `network.net.xml`, `routes.rou.xml` and `file.add.xml` are specified in `config.sumocfg`. It is useful to know the order of loading files to ensure the correct resolution of references. From SUMO Wiki:

1. the *network* is read
2. the *additional* files are read (completely from top to bottom) in the order in which they are given in the option
3. the *route* files are opened and the first n steps are read
4. each n time steps, the *routes* for the next n time steps are read

SUMO runs numeric simulation in discrete time steps. The length of a single time step can be specified in *config*. Interesting feature is *sublane* model that increases a lateral resolution of *lanes*, so *vehicles* drive in parallel or can move more to the sides inside their *lane*. This is useful for modelling two-wheeled vehicles or modelling the emergency corridor. To activate *sublane* model, lateral resolution must be set in *config* file.

Finally, here is a listing of usable *config* file:

```
<configuration>
   <input>
      <net-file value="network.net.xml"/>
      <route-files value="routes.rou.xml"/>
      <additional-files value="detectors.det.xml,\
       traffic_lights.tll.xml"/>
   </input>
   <output>
      <tripinfo value="output/tripinfo.xml"/>
      <!-- other outputs -->
   </output>
   <time>
      <step-length value="0.1"/>
      <!-- start at 7:00 a.m. and end at 7:10 a.m. -->
      <begin value="25200"/>
      <end value="25800"/>
   </time>
   <processing>
      <!-- sublane model -->
      <lateral-resolution value="0.4"/>
   </processing>
</configuration>
```

2.3.4 TraCI

There is a way how to directly interfere with a running simulation and change its state called *Traffic Control Interface (TraCI)* [17]. TraCI uses *Transmission control protocol (TCP)* based client/server architecture where SUMO acts as a server. TraCI client implementations exist in programming languages C++, .NET, Matlab or Java. However, the main library which is tested daily and supports the whole *Application programming interface (API)* coverage is in Python. TraCI can be used to model V2X applications and also to couple SUMO with other (often network) simulators. It is necessary to define the environment variable on your system that points to the installation directory of SUMO: `SUMO_HOME` in order to use TraCI.

API commands are logically grouped into (currently) 15 domains: *simulation*, *GUI*, *POI*, *polygon*, *vehicle*, *vehicletype*, *edge*, *lane*, *traffic light*, *induction loop*, *junction*, *multientry exit*, *areal*, *route* and *person*. Available methods of each domain can be found in the documentation of Python library[6]. Simulations are slower when running with TraCI. By how much slower is dependent on the amount and type of additional code that is executed in every step. This is a Python code snippet to get an idea about how simulation running with TraCI works:

[6]TraCI documentation: https://sumo.dlr.de/pydoc/

```
while traci.simulation.getMinExpectedNumber() > 0:
      traci.simulationStep()
      # user code performed after every simulation step, e.g.:
      speed = traci.vehicle.getSpeed("v1")
      pos = traci.vehicle.getPosition("v1")
      num_det = traci.inductionloop.getLastStepVehicleNumber("det_e1")
      # do something with the data
traci.close()
```

If you need to perform code on every simulation step, then a better and more efficient option is to implement a listener: subclass of `traci.StepListener`. The listener needs to override `step()` function. Moreover, if you need some values of particular TraCI domain, then there is a possibility to subscribe to those values. I combine these two concepts in the following code snippet to implement a listener that gets position and speed of a *vehicle* and counts from *E1 detector* in every step:

```
import traci
import traci.constants as tc

class SimulationListener(traci.StepListener):
   def __init__(veh_id, det_id):
      self.veh_id = veh_id
      self.det_id = det_id
      traci.vehicle.subscribe(self.veh_id, (tc.VAR_SPEED,
        tc.VAR_POSITION))
      traci.inductionloop.subscribe(self.det_id, (
        tc.LAST_STEP_VEHICLE_NUMBER))

   def step(self, t=0):
      veh_data = traci.vehicle.getSubscriptionResults(self.veh_id)
      il_data = traci.inductionloop.getSubscriptionResults(self.det_id)
      # do something with the data
      return True

listener = SimulationListener("v1", "det_e1")
traci.addStepListener(listener)
```

2.3.5 Outputs

SUMO is able to log a huge variety of different outputs and write them into XML files. By default, they are all disabled and must be activated by options in *config* file or in *additional* files. I describe only the outputs that I will use later. The whole list of possibilities can be found on SUMO Wiki[7].

- Fcd output: contains FCD data (position, speed and heading) for each time step and for all *vehicle types* with *device.fcd* specified in their definitions. A tool called *plot_trajectories.py* can visualize FCD outputs or save them *comma-separated values (CSV)* files for other processing.
- Tripinfo output: contains information about *trips* of all the *vehicles* defined in *rou* file. Among other entries, time of departure, time of arrival, total waiting time and total stop time can be found here. By default, tripinfo output logs only finished *trips* but can be set to log also unfinished *trips*.
- Edge data: logs various information, e.g. density, speed, waiting time for all *edges* in the *network* in every time step.
- Traffic light output: is used to observe the behaviour of traffic lights during a simulation run, especially times of switches of phases or programs definitions. Very useful for evaluating performance of dynamic traffic lights applications.
- Summary output: contains aggregated information about all the *vehicles* in the simulation, e.g. number of running *vehicles*, number of inserted *vehicles*, number of waiting to be inserted etc.
- Detectors output: logs number of *vehicles* passed over a detector, their speed etc.

Here is an example of *tripinfo* output:

```
<tripinfos>
   <tripinfo id="v1" depart="0.0" departLane="e1_0" departPos="5.10" ↘
       departPosLat="0.00" departSpeed="0.00" departDelay="-0.03" ↘
       arrival="25.4" arrivalLane="e2_0" arrivalPos="51.80" ↘
       arrivalPosLat="0.00" arrivalSpeed="18.61" duration="25.4" ↘
       routeLength="153.39" waitingTime="0.00" waitingCount="0" ↘
       stopTime="0.00" timeLoss="0.95" rerouteNo="0" devices="" vType="↘
       typePassenger" speedFactor="1.12" vaporized=""/>

   <!-- other entries -->
</tripinfos>
```

2.3.6 Other tools

SUMO comes with a lot of Python based tools that can be found in `SUMO_HOME/tools`. They can help a user with wide range of tasks, e.g:

[7]SUMO outputs: https://sumo.dlr.de/wiki/Simulation/Output

- configuring *tll* (traffic lights logic)
- generating *detector* definitions
- adjusting *network*
- visualisation of outputs

Another useful library is Sumolib[8]. This library is designed to process static data of SUMO. I found it extremely convenient to parse outputs using *sumolib*. The following code snippet shows how to parse *tripinfo* output using *sumolib*. Do not let the first seven lines confuse you, this is a standard beginning of any SUMO tool to ensure that `SUMO_HOME` is defined:

```
import os, sys
if 'SUMO_HOME' in os.environ:
   tools = os.path.join(os.environ['SUMO_HOME'], 'tools')
   sys.path.append(tools)
else:
   sys.exit("please declare environment variable 'SUMO_HOME'")
import sumolib

for trip in sumolib.xml.parse('output/tripinfo.xml', 'tripinfo'):
   print(trip.duration)
   if trip.vType == "typePassenger":
         print(trip.waitingTime)
```

2.4 Vehicles in network simulation

Radio communication systems in which vehicles are involved can be described as:

- *vehicle to vehicle (V2V)*

  Communication of vehicles between each other. Vehicles can share data about their environment, e.g. jams, accidents. They can inform other vehicles going in the opposite direction about the problems they might expect on the road.

- *vehicle to infrastructure (V2I)*

  Communication between vehicles and infrastructure along the roads. This can be communication of vehicles with TLCs or *Roadside Units* (RSUs), which are built specifically for this purpose. RSUs can serve as proxies when vehicles are too far apart and V2V is infeasible.

- V2X

  The abbreviation captures both of the two stated above.

[8]Tools/Sumolib: https://sumo.dlr.de/wiki/Tools/Sumolib

Note that the terms *car to car (C2C)* and *car to infrastructure (C2I)* are often used in literature and among some practitioners. It has the same meaning and I will use the V-version only.

An in-depth source about the topic is [18]. The type of communication protocol used, depends on the application. If the time of delivery of messages is not critically important for the application, any cellular network can be used, e.g. LTE. On the other hand if the delay of a message might affect safety of passengers on board, then *Dedicated short-range communication (DSRC)* must be used. An example of DSRC is IEEE 802.11p standard. This protocol is designed for *Vehicular ad-hoc network (VANET)*. To simulate VANET, I used an open source library called *Vehicles in network simulator (Veins)*, originally published in [19].

Veins[9] couples two simulators. The first one is SUMO, which is discussed in this chapter. The second one is *Object-oriented modular discrete event network simulation framework (OMNeT++)*. OMNeT++ is heavily used in the industry and in academia for simulations of communication networks. To use Veins, it is necessary to become familiar with OMNeT++. There are really good tutorials online[10], I would recommend to go through 'tic-toc' tutorial first. OMNeT++ has its own Eclipse-based *Integrated development environment (IDE)*. Veins source files are imported into OMNeT++ IDE as a separated project. The behaviour of user-defined modules is implemented in C++ programming language. I will not go into details here (those are covered in the tutorials). But briefly, the minimum work that need to be done is to implement two methods:

- `initialize()`

  Run only once to initiate class fields and to send first messages.

- `handleMessage(cMessage *msg)`

  The behaviour of the module after receiving a message from other modules or self-message.

To build complex networks of modules, OMNeT++ has its own high-level language called *Network description language (NED)*. Simulation is started from project-unique file called `omnetpp.ini`. This 'main' file serves for defining parameters of modules, settings of repeated simulation runs and various additional settings.

Veins initializes every new *vehicle* from SUMO as a Car module in OMNeT++. TraCI client C++ implementation is a part of Veins so you can influence the *vehicles* in SUMO. Veins comes with `sumo_launchd.py` program that uses TraCI internally. The program starts a daemon, that listens for a request from OMNeT++ to launch SUMO and then, when it's running, to update the state of both simulators after each step. You can replace `sumo` with `sumo-gui` to run with GUI.

```
./sumo-launchd.py -vv -c sumo
```

Starting to simulate your own scenario might be a little bit tricky at first. The easiest approach is to copy the example provided and modify it to your needs. Let's say we want to create

[9]Veins homepage: `http://veins.car2x.org/`
[10]OMNeT++ tutorials: `https://docs.omnetpp.org/tutorials/tictoc/`

a simulation called `MyScenario`. The first thing we need to do is to create `my_scenario.launchd.xml` file where we define the names of all SUMO related files for our scenario. Those files must be placed in the same directory as the Veins project itself. It might look like this:

```
<launch>
   <copy file="network.net.xml"/>
   <copy file="routes.rou.xml"/>
   <copy file="traffic_lights.tll.xml"/>
   <!-- other files of SUMO scenario -->

   <copy file="config.sumocfg" type="config"/>
</launch>
```

Another necessary step is to create a NED file that defines the components of your scenario. Let's say we want to have one RSU in `MyScenario`:

```
import org.car2x.veins.nodes.Scenario;
import org.car2x.veins.nodes.RSU;
network MyScenario extends Scenario
{
   submodules:
      rsu[1]: RSU {
         @display("p=150,40;i=veins/sign/yellowdiamond");
      }
}
```

Now you need to implement application layer of the submodules. To do this I would recommend to copy `MyVeinsApp` module and change whatever you need. The module has well documented source files. The parameters for the submodules are specified in the `omnetpp.ini` file. Note, that even though we specified no Car module in the NED file, they are initiated dynamically. You can define their own layer to model VANET application. It might look like this:

```
*.rsu[*].applType = "MyVeinsAppl"
*.rsu[*].appl.headerLength = 80 bit
*.rsu[*].appl.sendBeacons = false
*.rsu[*].appl.beaconInterval = 1s
*.rsu[*].appl.beaconUserPriority = 7
# other settings
*.manager.moduleType = "org.car2x.veins.nodes.Car"
.manager.moduleName = "node"
```

```
*.node[*].applType = "MyCarVeinsAppl"
*.node[*].appl.sendBeacons = true
```

Veins creates better abstractions for simulating VANET. You do not have to implement 'pure' `handleMessage(cMessage *msg)`. You choose your application specific callbacks instead. For example, from the following snippet from `MyVeinsAppl`, I used only the first one and I kept others empty.

```
void MyVeinsApp::onBSM(DemoSafetyMessage* bsm)
{
   // Your application has received a beacon message
   // from another car or RSU
   // code for handling the message goes here
}

void MyVeinsApp::onWSM(BaseFrame1609_4* wsm)
{
   // Your application has received a data message
   // from another car or RSU
   // code for handling the message goes here
}

void MyVeinsApp::onWSA(DemoServiceAdvertisment* wsa)
{
   // Your application has received a service advertisement
   // from another car or RSU
   // code for handling the message goes here
}
```

To sum this up, I recommend these steps to run your own simulation in Veins:

1. Copy the example from Veins source code.
2. Change `.launchd.xml` file with the names of your SUMO files.
3. Copy the SUMO files to the directory of Veins project.
4. Implement application layers for Cars or RSU modules of Veins. Start from `MyVeinsApp`.
5. Attach those application layer to modules in `omnetpp.ini` file.
6. Start `sumo_launchd.py` daemon.
7. Start the Veins simulation from within the OMNeT++ IDE.

Chapter 3

Scenario

In this chapter, I give a detailed description of the process of creating the scenario. I started creating the scenario in SUMO version 0.32 but at that time versions 1.0 and subsequently 1.1 were released. The final simulation is built in SUMO version 1.1 and OMNeT++ 5.4.1, which are coupled within the Veins framework version 5.a1. At the time of writing, SUMO version 1.2 is released but is not yet compatible with latest version of Veins.

3.1 Area

All the simulations described later in the thesis take place in Brno city (the Czech Republic) around Mendlovo náměstí. The choice is not arbitrary, the area contains four major TLJs and prioritization is currently used in all four of them. I will refer to junctions by code-names used by traffic engineering companies operating in Brno. The codes and respective junctions are given in Table 3.1 and their positions are visualised in Fig. 3.1.

3.2 Network

Basic *network* was imported from *Open Street Map (OSM)* [20] using *osmWebWizard.py*[1] script which can be found in the tools available with SUMO package. After starting the script, a new page is opened in an internet browser with OSM and several options in the menu. A user can select an area from the map to import directly to SUMO, pick what kinds of traffic to generate (options include cars, trucks, pedestrians etc.), then set the duration of the simulation and click on 'Generate Scenario' button. The tool then generates all the files needed to start the simulation.

I decided to generate only the *network* without *vehicles* because guessed *routes* are usually randomized and do not reflect reality. Generating the traffic demand is described in 3.3. As

Junction code	Streets intersecting
1.01	Úvoz, Pekařská
1.03	Mendlovo náměstí, Křížová
1.02	Křížová, Václavská
2.06	Křížová, Poříčí, Vídeňská

Table 3.1: *Substituting the junction names*

[1]How to import OSM map to SUMO: `https://sumo.dlr.de/wiki/Networks/Import/OpenStreetMap`

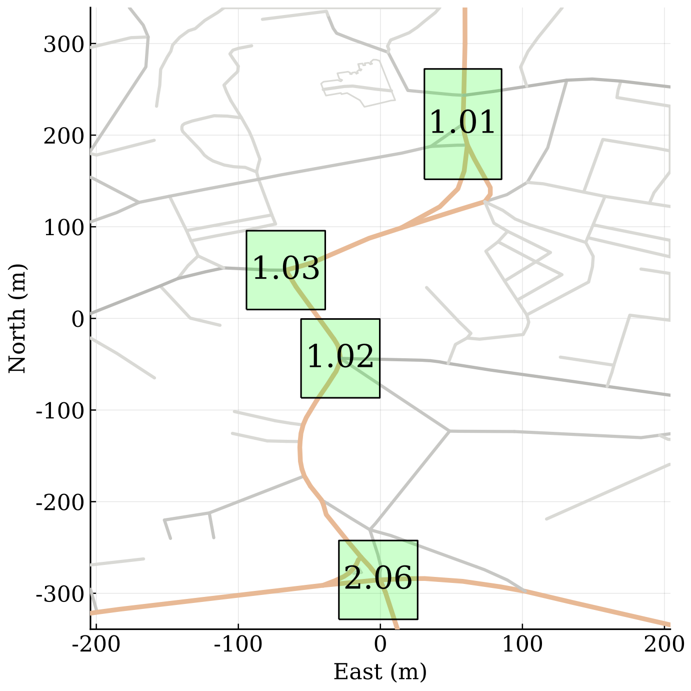


Figure 3.1: *Junctions positions and their respective codes*

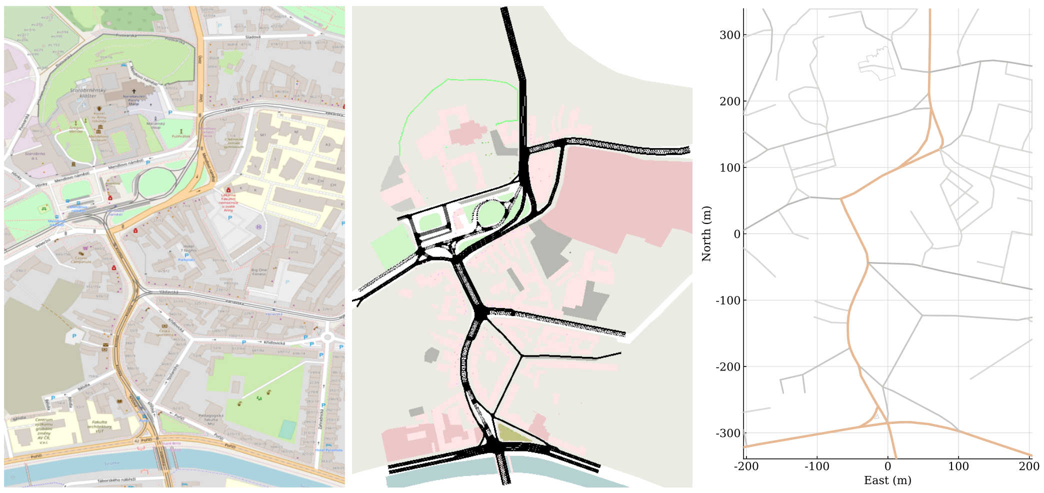
for *network* generation, the script does pretty good job by itself but additional corrections are often necessary:

- *Junctions* of generated SUMO *network* might consist of many little *junctions*. This can be solved by merging the *junction* points together in *NETEDIT* and resetting the *connections* of *lanes*.
- Tram rails are created as extra *edges* that are placed on top of other *edges*. This results in unrealistic behaviour where vehicles and trams can meet at the same position without collision. I solved it by removing extra *edges* and then adding extra *lanes* specifically for trams to remaining *edges*.
- Pedestrian sidewalks are also imported. I decided not to simulate pedestrians in the scenario so I removed all the *edges* for pedestrians.

Nice perk is that *osmWebWizard.py* contains option to extract shapes and buildings information into XML file which can be loaded to SUMO-GUI as *additional* file. Those files contains definitions of polygons. This can serve as colourful decoration (as seen in Fig. 3.2b), but more importantly, buildings are used as obstacles for signal shadowing models of V2X communication in Veins. I will come back to it later in Section 3.6.

3.3 Traffic demand

Devices for counting passing vehicles are deployed in the area, this equipment is called *Automatic Traffic Recorder (ATR)*. Either induction loops built permanently into the roads or computer vision systems implement ATR. I obtained datasets which cover the whole week from



(a) *OpenStreetMap* (b) *SUMO-GUI* (c) *Julia plot*

Figure 3.2: *Simulated area as viewed in: (a) OpenStreetMap, (b) SUMO-GUI and (c) plot made with Julia package which will be used for further visualisations*

14 November to 20 November 2018. Measurements are aggregated in intervals 10 minutes long. Unfortunately, measurements from junction 1.02 were not provided.

Traffic counts from ATR do not contain the whole information to model the demand, because we still do not have the *routes*. Fortunately, SUMO comes with several tools to estimate the traffic demand from observation points[2]. Some of them are described in [13]. Necessary first step is to place *E1 detectors* in the *network*, preferably on the exact same spot as they are placed in reality. I briefly explain two modules I used.

3.3.1 *DFROUTER*

This module assumes that an area is completely covered by detectors. All sources and sinks of the *network* must be measured, *DFROUTER* does not provide satisfactory results otherwise. The only case where it is reasonable to use *DFROUTER* in my scenario is to generate demand for only one junction. This is indeed what I did with smaller *network* containing only junction 2.06, this way I learned how to use these tools. *DFROUTER* is not suitable for dense networks, such as cities, but is best suited for highway scenarios. Detailed explanation of the workflow can be found in [21] and [22].

DFROUTER takes network, detector definitions and measurement file as input and then produces routes and vehicles definitions. The minimal prompt may look like this:

```
dfrouter --net-file network.net.xml
         --measure-files measurements.csv
         --routes-output routes.rou.xml
         --emitters-output vehicles.xml
```

[2]Introduction to demand modelling: https://sumo.dlr.de/wiki/Demand/Introduction_to_demand_modelling_in_SUMO

There are so many other options for *DFROUTER* that it is not convenient to define all of them in a single command. The solution is to create *config* XML file (similar to *sumocfg*) that can be used to define all parameters and then use only this file as input to generate the traffic demand:

```
dfrouter --save-template config.dfrocfg
dfrouter -c config.dfrocfg
```

3.3.2 *Flowrouter*

Flowrouter is suited for scenarios where not all sources and sinks are covered with detectors so it is the best tool for my scenario and it produces the most realistic results. I generated all my traffic demand definitions with this tool. *Flowrouter* is in continuous development, a short description can be found in [13]. The script works in two steps.

1. It generates all possible *routes* possible in the network.

2. Maximum flow problem [23] is solved assuming the provided measurements as capacity.

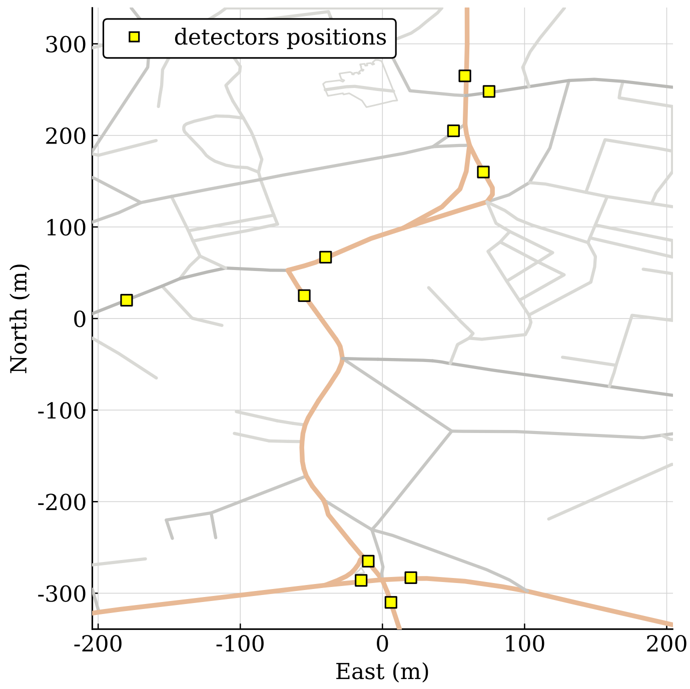
Similarly as *DFROUTER*, *flowrouter* takes *network*, *detectors* definition and measurement file as input. The output traffic demand in the *rou* file is generated as *flows* going through the *routes* aggregated in time intervals whose duration can be provided as an option. Unfortunately, *flowrouter* does not come with *config* file yet, so all inputs must be specified in the command. This can get tedious if you need more options so a good advice is to write *flowrouter* calls into a BASH script. This is how the minimal command may look like:

```
python $SUMO_HOME/tools/detector/flowrouter.py
      -n network.net.xml
      -d detectors.det.xml
      -f measurements.csv
      -o routes.xml
      -e flows.xml
      -i 10
```

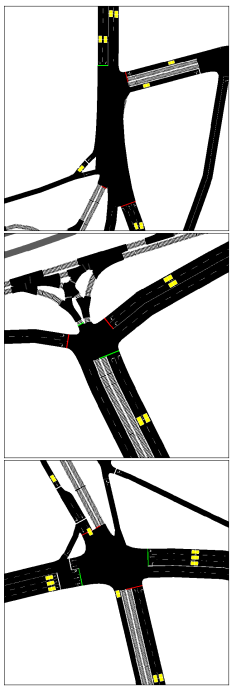
3.3.3 Processing the datasets

After taking a closer look at the ATR datasets I encountered some problems.

- The detector placed on left-turning lane on direction to the north at junction 1.01 malfunctioned and showed unrealistic number 255 in most of the readings, even at night. I excluded the detector from the scenario.



(a) *Positions of vehicles counting detectors*



(b) *Detailed views.*

Figure 3.3: *Positions of the detectors which were chosen for demand modelling. The pictures on the right are screenshots from SUMO-GUI. They depict the placement of the detectors (yellow rectangles) on the lanes of junctions (from top) 1.01, 1.03, 2.06. Junction 1.02 is missing.*

- There are pairs of detectors on the same road placed close to each other with different measurements in the same interval. This can be caused by using different technologies for counting vehicles with one being more erroneous than the other or maybe there was a jam in the particular interval. Anyway, in such cases I decided to use only one of the detectors so I do not confuse the routing tool. Positions of chosen detectors are depicted in Fig. 3.3.

In total I used 26 detectors out of 42 provided. The detectors have code names starting with the code name of the junction on which it lies and some other code after it.

The datasets I obtained are saved as CSV files. It was necessary to transfer them into a format suitable as input for *DFROUTER* or *flowrouter*. I wrote a Python script to do the job. The input for the routing tools has the following format:

```
Detector;Time;qPKW;qLKW;vPKW;vLKW
1.01_DVA1;0;10;0;25;0
... other entries ...
```

`qPKW` is the number of passenger vehicles that went through the detector during the interval. `vPKW` is the average speed of those vehicles. `qLKW` and `vLKW` are the number and speed of transport vehicles that went through the detector. My datasets do not include the speed readings or any other information. So I set all `qLKW` and `vLKW` to zero and `vPKW` to 25.

3.4 Public transport

Trams, buses and trolleybuses go through the area and there are several stops. Thus, I could not omit public transport from the simulations. I manually created *route* files with *vehicle type* definitions and *flows*. Moreover, SUMO allows *busStops* definitions and those need to be created as well. I did all of this based on timetable[3] of public transport in Brno. All public transport going in working days is simulated throughout the whole day. The *flows* are determined based on how many connections depart in particular interval. The code snippet below is a part of my scenario:

```
<additional>
   <vType id="TRAM" vClass="tram" color="yellow"/>
   <!-- other vTypes -->

   <busStop id="tram_mendl_left_in" lane="gneE6_0" lines="1"/>
   <busStop id="tram_vaclavska_out" lane="-4073052#0_0" lines="1, 2"/>
   <!-- other busStops -->
</additional>

<flows>
   <interval begin="18000" end="21600">
      <!-- 5:00 a.m. - 6:00 a.m. -->
   <flow id="tram1_5to6_0" type="TRAM" from="32348347#0.297" to="↘
      -4073052#0" number="7">
      <stop busStop="tram_mendl_left_in" duration="20"/>
      <stop busStop="tram_vaclavska_out" duration="20"/>
   </flow>
   <!-- other flows -->
   </interval>
   <!-- other intervals -->
</flows>
```

3.5 Traffic lights

I was provided with static signal plans for all four junctions in the area from our industry partner. Even though static plans are not used during peak hours and dynamic plans based on phases switching triggered by demand are used instead, I decided to use them. I was told by the respective partner that based on his look into the detailed log of traffic lights behaviour, the dynamic programs closely resemble the static programs. The only differences are in times when a vehicle of public transport request priority or when pedestrian pushes the button to request green phase on zebra crossing. Otherwise, the green times for all the lanes are almost

[3]Public transport timetable in Brno: `http://www.jrbrno.cz/`

the same for both static and dynamic signal plans. I do not simulate pedestrians of preemption for public transport, see Section 3.8 for more details.

I obtained signal plans with 60s, 80s and 100s long *cycle time (CT)*. The rule of thumb says that the denser the traffic flows are, the longer CT we should choose. I was told that that the 100s CT signal plans are used most of the day, from 7:00 a.m. to 8:00 p.m. Signal plans with shorter CT are active at night or in times when traffic is not expected to be big. I chose to use only the plans with 100s CT for all my simulations because I simulate EVs going through the area only in peak hours.

The signal plans definitions were given to me as scanned documents so it was necessary to put traffic lights definitions into SUMO manually. I wrote them into separated *tll* files for each TLJ. This work is tedious and error prone so it needs to be done with caution.

3.6 V2X communication

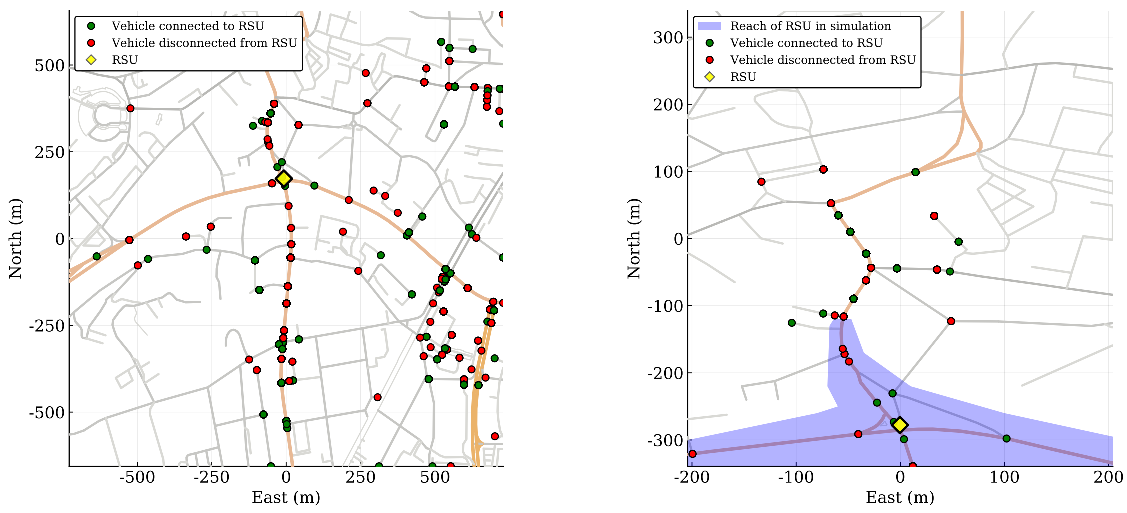
Public transport vehicles in Brno have capability to communicate with each other and with the infrastructure via installed V2X communication units. TLCs are also equipped with similar communication units. Moreover, there are several RSUs in the area. These units are still quite new and their potential is not fully used yet. This will change in near future.

I ported the scenario into Veins to simulate scenarios when EV communicates with TLC directly via IEEE 802.11p standard. The buildings exported from OSM now play crucial role for obstacle shadowing models. Parameters of the shadowing model can be read from the listing:

```xml
<root>
   <AnalogueModels>
      <AnalogueModel type="SimplePathlossModel" thresholding="true">
         <parameter name="alpha" type="double" value="2.0"/>
      </AnalogueModel>
      <AnalogueModel type="SimpleObstacleShadowing" thresholding="true"↘
          >
         <obstacles>
            <type id="building" db-per-cut="9" db-per-meter="0.4"/>
         </obstacles>
            <parameter name="carrierFrequency" type="double"  value="↘
                5.890e+9"/>
      </AnalogueModel>
   </AnalogueModels>
   <Decider type="Decider80211p">
      <!-- The center frequency on which the phy listens-->
      <parameter name="centerFrequency" type="double" value="5.890e9"/>
   </Decider>
</root>
```

One of the RSU is placed in the north-east corner of the junction 2.06 and is nowadays operated in a mode for testing. Among other data, the RSU logs GPS positions of vehicles that successfully connected to it as well as GPS positions where the connection was lost. The dataset with these logs was provided to me so I was able to compare V2X communication in simulation with real life measurements. The comparison can be seen in Fig. 3.4. Firstly, I tried to plot several points from the log on a larger area around my scenario to get an idea about the reach of the RSU. The results surprised me a bit because there are points quite far away from the RSU where I would not expect the RSU to function at all, especially the points at north-east tip of the map at Fig. 3.4a. However, as was confirmed to me by respective partner, those points are usually unpredictable connections caused by reflections from buildings. The points placed on the main roads approaching the position of the RSU (roads in orange color) are the points where the RSU is expected to work well.

I was interested in the reach of beacons sent via IEEE 802.11p in the scenario so I tried to simulate a few V2I enabled *vehicles* and observe where the connection is established with the RSU. The resulting area is depicted in Fig. 3.4b also with a few points from real life measurements. This plot shows that the communication as modelled in Veins can be compared to reality and that in most cases, it is even conservative because the RSU does not reach any car close to the junction 1.02. From this result I conclude that Veins is a good tool for testing new V2X applications in this scenario.



(a) *Larger area* (b) *Area of my scenario*

Figure 3.4: *Comparison of real life measurements of V2I communication between vehicles and RSU. On the left side, a larger map shows the reach of communication. On the right side, only my scenario is depicted. Use the position of the RSU for orientation.*

3.7 Emergency vehicle

With the network, traffic demand, traffic lights and V2X communication prepared, it is time to model EV. Here is the list of things one can do in SUMO in order to simulate EV[4]

1. Define *vType* equipped with *device.bluelight*. The *device* allows vehicle to violate red traffic lights and other vehicles are forced to create a corridor for EV (this is possible

[4]Emergency Vehicle simulation: `https://sumo.dlr.de/wiki/Simulation/Emergency`

only in a simulation with *sublane* model).

2. Set `vClass="emergency`, this allows vehicle to overtake on the right.
3. Allow vehicle to exceed speed limit by specifying `speedFactor` > 1 in *vType* definition.
4. Take over the control of simulated vehicle with TraCI and implement special behaviour.

I use the first three items from the list. Examples of similar settings used in EVs simulations can be found in [5, 6, 24, 25, 26]. The *vehicle type* definition is listed here, some less important parameters are omitted for brevity.

```
<vType id="EMERGENCY" vClass="emergency" length="6.5" width="2.1"\
      speedFactor="1.5" guiShape="emergency" laneChangeModel="SL2015"\
      carFollowModel="IDM">
      <param key="has.bluelight.device" value="true"/>
      <param key="has.fcd.device" value="true"/>
</vType>
```

Then you need to write *vehicle* specification (time of departure) and also the definition of *route*. I simulate only one *route* for EV which is depicted in Fig. 3.5. I did a few additional adjustments to the *network* to make the resulting behaviour of EVs more realistic. In reality, passenger vehicles are not allowed to drive on lanes that have tram rails but those lanes are otherwise usable for buses or EVs. To give a simulated EV freedom to choose a *lane* so it can go faster if some *lane* is more occupied than the other I set some extra *connections* at *junctions* as can be seen in Fig. 3.6. This changes in the *network* need to be treated also in the traffic lights program definitions in *tll* files. One option is to simply add one more character at a position of respective connection in traffic light phases states or to define *signal groups* in the program. I chose the first option so every extra *connection* extends the state of the traffic light phases definitions of respective *junction* by one character.

3.8 Simplifications made

During the creation of the scenario, I made several simplifications. I was either forced to make them (e.g. because I did not obtain necessary data) or I chose to make them for my own convenience (e.g. because the problem was too complex or insignificant). The simplifications are listed here.

- Preference of public transport vehicles on TLJ is not modelled. Nowadays, trams request TLC for preference when approaching the area described in 3.1. In result, their signal phase can be shifted along the cycle time as necessary, which affects the rest of the phases. I use only static signal plans in the simulation. Thus, the trams almost never stop on TLJs in reality, but they do in my simulation. So situations when both tram and EV request priority are not considered.

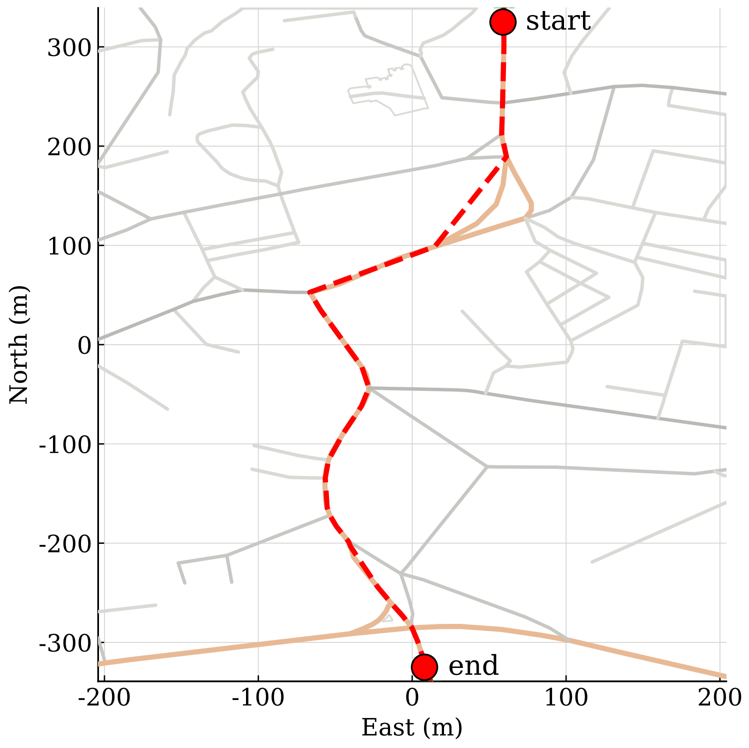
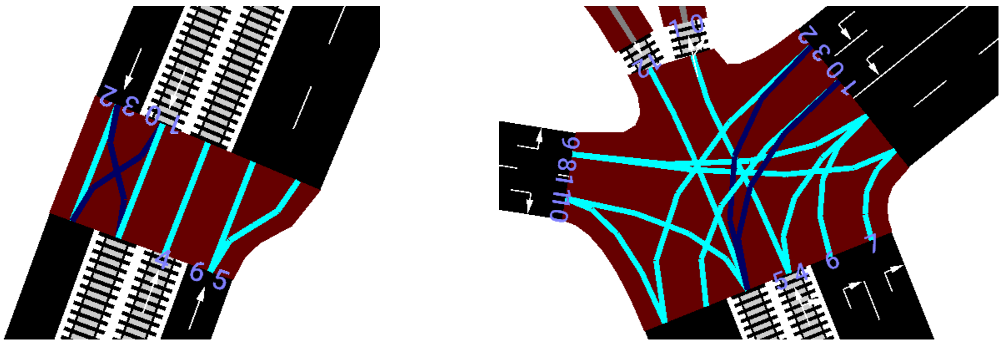


Figure 3.5: *The route of EV for simulations. According to partners from industry, EVs pass this route frequently. Note that all four TLJ are crossed.*

- Pedestrian crossings and pedestrians are not modelled. In reality, traffic lights for pedestrians affect prioritization methods for EV because the minimal green time for pedestrians is much longer than the minimal green time for vehicles.
- Synchronization of traffic lights is not implemented. In practise, traffic lights are synchronized in phases to provide green waves for traffic participants. When they desynchronize, TLC starts to synchronize them again, which can take several minutes. This could be partially solved by actuated traffic lights in SUMO. But I didn't want to introduce extra complexity into the simulation.
- No parking lots and parking vehicles are simulated. There are two big parking lots in the zone which might influence the flows on the streets.
- One way railroad track on Poříčí street is missing.
- Except from public transport and emergency vehicle, only one other *vehicle type* is simulated. The type is *passenger* vehicle with default parameters from SUMO.

3.9 Validation

To validate that the simulation scenario corresponds to reality, we need to compare counts of vehicles from real induction loops and counts of simulated vehicles from corresponding *E1 detectors*. I used three ways how to do it. I show the procedure and results from one full day, I chose Tuesday, November 20, 2018. The results for other days are similar.



(a) (b) *Extra connections at junction 1.03*

Figure 3.6: *Adjustments to connections of lanes on two junctions. Note the curious connections in dark blue color, they allow EV to switch lanes at junctions. Other simulated vehicles are not permitted to use these connections.*

- The *flows* computed by *flowrouter* can be compared with measurements with *flowFromRoutes.py* tool. It prints comparison and statistics to standard output. The command may look like this:

```
python $SUMO_HOME/tools/detector/flowFromRoutes.py
      -d detectors.det.xml
      -r routes.xml
      -e flows.xml
      -f measurement.csv
      --geh
```

  You can add `-i 60` to make the comparison aggregated in 60 minutes intervals and not in the whole simulation duration. The parameter `--geh` makes the script compute GEH statistics[5], last value in Table 3.3.

- Another approach is to collect *edge output* from simulation run and compare it with real measurements. There is a tool for it called *flowFromEdgeData.py*. Example of usage:

```
python $SUMO_HOME/tools/detector/flowFromEdgeData.py
      -d detectors.det.xml
      -e edgeData.xml
      -f measurements.csv
```

  Output from this one usually reports slightly worse error than from *flowFromRoutes.py*, that is because simulation (*edgeData*) can deviate a little bit from its plan (*flows*). The results are shown in Table 3.2 and Table 3.3(except GEH). Measurements are aggregated for groups of neighbouring detectors throughout the whole day.

[5]See Appendix III in: http://content.tfl.gov.uk/traffic-modelling-guidelines.pdf

- Third approach is to visually compare simulated and real measurements. Fig. 3.7 illustrates the results of such effort. I simulated the whole day with virtual *E1 detectors* and collected the counts of vehicles aggregated in 10 minutes interval (just like the real measurements). Then I plotted them both against each other for all detectors. The vertical axis shows the flow of vehicles through the detector per 10 minutes, the horizontal axis shows daytime. In some cases it might look that the simulation is way off, e.g. 1.01_DS3. But when you take into account that 1.01_DS3 and 1.01_DS4 are put on neighbouring lanes, the total numbers of vehicles more or less fit. It is the aspect of simulation that vehicles changed lanes as they did. The same holds for pairs 1.01_-DVC1, 1.01_DVC2 and 2.06_DVC1, 2.06_DVC2. This is not apparent from results of two previous two approaches because the detectors are grouped in them.

Taking into consideration that I did not obtain data from junction 1.02 and one of the detectors malfunctioned, the results are satisfactory. In conclusion, I am confident that the simulation scenario resembles reality.

Detector group	**Simulation**	**Real measurements**
1.01_DS{3,4}	8749	9495
1.01_DS5	3501	3226
1.01_DS6	3933	3813
1.01_DVA{1,2}	10024	9749
1.01_DVB	3592	3364
1.01_DVC{1,2}	8689	9395
1.03_DS{1,2}	2807	4975
1.03_DVB{1,2}	5610	7228
1.03_DVC{1,2}	10276	11294
2.06_DS2	9976	9427
2.06_DVA1	9509	9509
2.06_DVA2	2143	2142
2.06_DVB{1,2}	9841	9736
2.06_DVC{1,2}	7460	8030
2.06_DVD{1,2}	14361	14290

Table 3.2: *Comparison of simulation with measurements*

Avg sim flow	**Avg real flow**	**RMSE**	**RMSPE**	**GEH**
7364.73	7711.53	827.42	0.13	0.94

Table 3.3: *Statistics of simulation and measurements and their comparison*

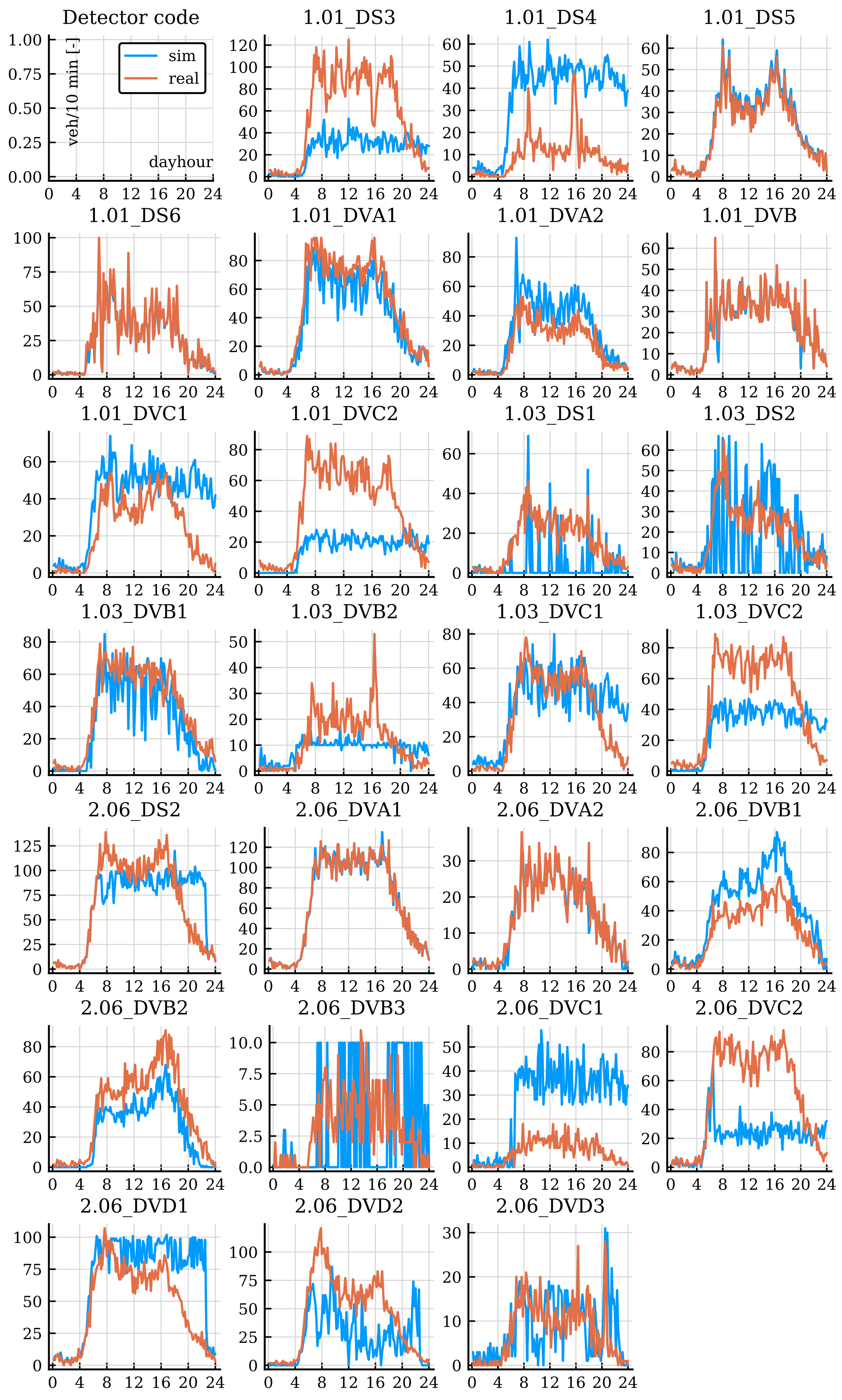


Figure 3.7: *Comparison of all day simulation with measurements from ATR. Day: Tuesday, November 20, 2018. The first plot at the top-left position serves as a legend.*

Chapter 4

Preference modes

Reasonably realistic traffic demand was set. The simulation scenario is now ready to be experimented with EVs. I have simulated in those three following modes.

1. EV gets no preference at TLJ. I will refer to this mode as 'no preference'.

2. EV gets preference just after crossing the virtual border set beforehand. I will refer to this mode as 'distance based preference'.

3. EV requests priority and TLC computes the time when preference phase starts based on the actual traffic situation in front of TLJ. I will refer to this mode as 'queue discharge based preference'.

In this chapter I will describe how each mode works and how is the functionality modelled in SUMO. In the case of queue discharge based preference, I also explain how Veins was used. The evaluation and comparison of all three modes is in the next chapter.

4.1 No preference

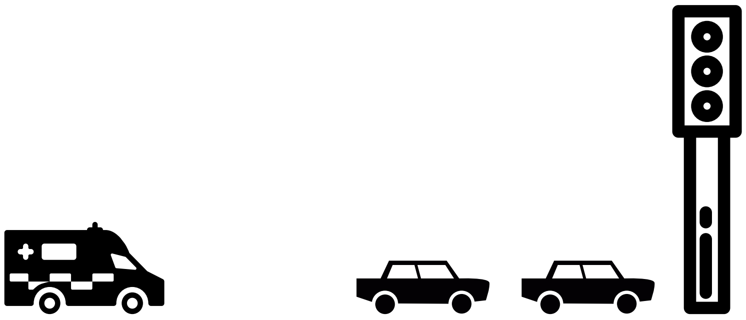


Figure 4.1: *Schematic picture of no preference mode*

To simulate a situation where EV gets no priority is easy. At this point, everything is ready. However, there are few things which need to be considered while simulating this mode and collecting outputs. The model is not flawless. EV sometimes gets stuck unrealistically behind another vehicle and is unwilling to change lane. I tried to find some parameters in lane-changing models to make it better but I was unsuccessful.

The behaviour of other vehicles that are trying to clear the way under the effect of EV's *device.bluelight* usually works as expected but sometimes a bug occurs and some vehicle blocks the way for EV.

Another imperfection is driving and overtaking in the opposite direction. In reality EVs drivers often drive in opposite direction to take over a queue. Unfortunately, *opposite driving* model in SUMO is currently incompatible with *sublane* model. There is a way to create a *network*, where driving in opposite direction is possible, if *sublane* is not used. However, if I would use this way, then I would also have to simulate also other modes without *sublane*, to make the simulations comparable in the final evaluation. I had to make a decision, whether to use *sublane* or *opposite driving*. I have chosen to use former because it is more realistic for overall simulation and *sublane* is needed for other vehicles to clear the way for EV. I think I have made the right decision. I also tried *opposite driving* and the resulting behaviour was not convincing enough.

4.2 Distance based preference

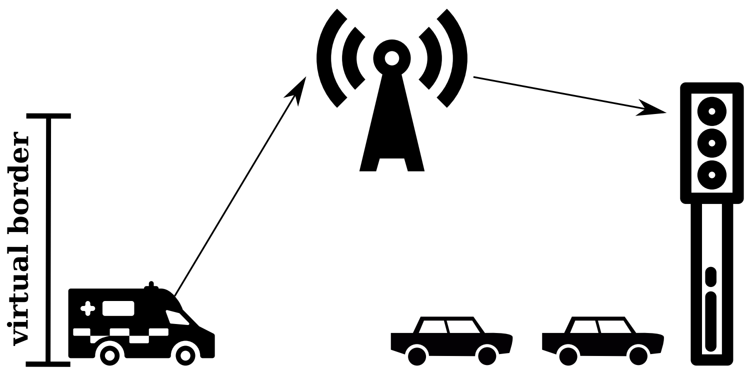


Figure 4.2: *Schematic picture of distance based preference mode.*

4.2.1 Description

This mode is currently used in the area for EVs preemption. The working principle is straightforward: preference starts after EV crosses a virtual border of a TLJ en-route.

Creating the virtual border is tedious job and is not replicable to more than one junction. A geographical point needs to be defined for every upstream lane connected to a TLJ which is about to have prioritization capability. Those points need to be at a specified driving distance from the TLJ (usually around 300 meters in urban areas). The points produce a shape that we call the virtual border. Now, the virtual border can be integrated into traffic lights system and plotted into a map.

An EV has *On board unit (OBU)* that is connected to GPS and is able to send messages via cellular network - LTE. A path of the EV is known, so if there is a TLJ with preference on the way, the OBU can check if the virtual border of the TLJ was crossed, with every GPS coordinates update. Right after the border is passed, EV signals priority request via LTE infrastructure.

There is a rule for traffic lights behaviour in the Czech Republic[1] which states that any green phase for vehicles cannot be active for less than 5 seconds (I will refer to this rule as

[1]Norm ČSN 365201-1

5 second rule). The minimal time is even longer for green phase of pedestrian crossings. TLC cannot start the preference immediately, but has to first check if the rule is not violated. If the EV requests priority in 'bad' time, e.g. when green phase just started for pedestrian crossing that intersects the path of EV, the preference phase is postponed by up to 20 seconds. The preference stops after EV gets behind the TLJ and is around 20 meters away. Maximal duration of preference is one minute.

4.2.2 Implementation

To model the distance based preference, I wrote Python program that runs the simulation using TraCI. The program takes *config* file and *rou* file with EV definition (*vehicle type*, *vehicle* and *route*) as inputs. Two constants are defined: `DIST_THRESHOLD=300` (the driving distance from a *junction* to the virtual border) and `DIST_THRESHOLD_AWAY=40` (the distance from the *junction* that needs to be surpassed to stop preference). The program is object-oriented, I wrote a few classes to make an abstraction of the behaviour 4.2.1. Here is a list of the classes with descriptions of their purpose:

- `EmergencyVehicle`

  An instance of this class holds information from provided *rou* file.

- `EmergencyVehiclesManager`

  Instanced only once at the beginning of the simulation. Holds list of `EmergencyVehicle` and can answer questions about their *route* or time of departure.

- `Listener4CheckingEmVehsDepart`

  Instanced only once at the beginning of the simulation. Subclass of `traci.StepListener`. The only task of this object is to wait for departures of `EmergencyVehicle` and then attach a new object of `Listener4ParticularEmVeh` for each departed `EmergencyVehicle`.

- `Listener4ParticularEmVeh`

  Instanced for each departed `EmergencyVehicle`. Subclass of `traci.StepListener`. Each instance has a reference to `TlsPreferenceManager`. An instance of this class checks the driving distance to *junctions* on the *route* of particular `EmergencyVehicle`. If the distance is less then the constant `DIST_THRESHOLD`, it notifies `TlsPreferenceManager` with priority request. After requesting priority, the instance checks whether the *junction* was passed already and if the `EmergencyVehicle` is at least `DIST_THRESHOLD_AWAY` from the *junction*. If yes, it notifies the `TlsPreferenceManager` to stop preference.

- `TlsPreferenceManager`

  Instanced only once at the beginning of the simulation. This object creates and activates preference program on *junctions* upon request. It keeps log about which *junctions* are in preference mode and saves their previous program and phase. The previous program and phase are activated again after receiving request to stop preference from `Listener4ParticularEmVeh`.

The basic workflow of the program can be described in four steps:

1. Parse *rou* file using *sumolib* and create objects of `EmergencyVehicle` type.
2. Initialize `TlsPreferenceManager` and `Listener4CheckingEmVehsDepart`.
3. Run simulation step.
4. Repeat 3. until there are no more *vehicles* or time is up.

The implementations of `traci.StepListener.step()` functions in `Listeners` take care of everything else in step 3.

Some more remarks

`TlsPreferenceManager` checks the 5 seconds rule when requested priority. Thus, the maximal theoretical delay of preemption is 8 seconds in my simulations (5 seconds till the end of green phase plus 3 seconds of yellow phase to stop the cars from other directions).

The preference program is created on demand. I compare current *state* of traffic lights with desired *state* for preemption (green only on *lanes* of EV path). If needed, I insert transition phase (orange for ways, others than prioritized) to stop other flows. Situation when red switches directly to green cannot happen in my scenario.

It can be used with any SUMO scenario provided that you specify the *rou* file. There might be problems with scenarios where traffic light program controls multiple *junctions*.

The program can handle multiple EVs running in the simulation. In the case of two EVs send priority request to the same TLC, the situation is resolved in first-come, first-served policy.

The source code is saved in `runner_LTE.py` file and is a part of this work. It is used from command line, there is also option `--no-gui` to run without GUI:

```
python runner_LTE.py -c config.sumocfg -e emergency.rou.xml
```

4.3 Queue discharge based preference

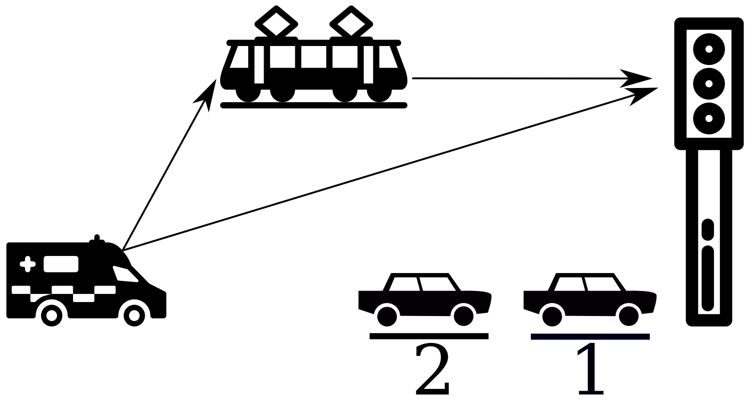


Figure 4.3: *Schematic picture of distance based preference mode.*

4.3.1 Description

The third mode is a V2X enabled preference based on direct communication between an EV and a TLC. The EV beacons GPS coordinates EV_{pos} and speed EV_{speed} every second. Public transport vehicles are able to forward the beacon. TLC decides, when should the preference phase start, based on EV_{pos} and current traffic situation in front of the TLJ. At this time, I assume that the state of the traffic in front of the TLJ is known to the TLC. By state of the traffic (or traffic situation), I mean the number of vehicles in the queue in front of the TLJ. The motivation for this method is to prevent situations, when the preference phase is started too early or too late and to remove the need of defining virtual borders.

A lot of time is wasted for other traffic participants when using distance based mode and the affected lanes in front of the TLJ are empty. It can happen that all the lanes, except the lane the EV is coming from, are stopped for tens of seconds without apparent reason to other road users.

On the other hand, in the rush hours when traffic demand peaks, EVs might struggle to pass the virtual border and vehicles cannot clear the way on time. In this situations, queue discharge based might start to prioritize sooner to accomplish the original purpose of preference: reducing a duration of EVs trip duration.

Similar method is used in [27]. The author develops an algorithm that computes the time needed for all vehicles in the queue to pass the TLJ. The preemption phase is started at that time so there are no vehicles approaching the TLJ from which the EV is coming at the time of arrival of the EV. I decided to try slightly different approach. This is my reasoning: the queue does not have to be discharged completely, it will be enough if the last vehicle of the queue is moving at the saturation speed. The following algorithm runs on TLC:

1. Compute the arrival time of the EV from EV_{pos} and EV_{speed} from the beacon $= AT$.
2. Get the number of vehicles in front of the stop line of TLJ $= w_0$.
3. Compute the time when the last vehicle in the queue reaches the saturation speed $= LT$.
4. Estimate how long will be the 'tail' of moving vehicles that are still in front of the stop line at the time LT.

5. Then compute the time that EV needs to surpass this distance $= XT$.

6. Compute time to start preference $= AT - LT - XT$.

The step 1. is straightforward. TLC gets driving distance of EV to the TLJ and divides it by EV_{speed}. I will explain other steps in the next section.

4.3.2 Queue discharge model

The task is now to develop a formula to compute the time needed for the last vehicle in the queue to accelerate to saturation speed (LT). I will use the exponential queue discharge flow rate and speed model given in [28, 29]. The derivations of formulas in the following lines are described in [29]. I will not repeat it here, I will only use them.

Considering a queue as an entity, its behaviour can be described by the following set of functions of the time since start of green phase:

$$v_s(t) = v_n\left(1 - e^{-m_v(t-t_r)}\right), \tag{4.1}$$

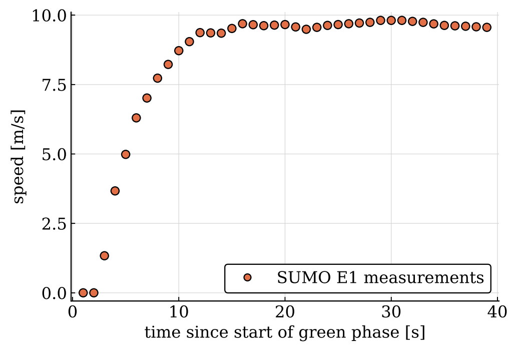
$$q_s(t) = q_n\left(1 - e^{-m_q(t-t_r)}\right), \tag{4.2}$$

$$h_s(t) = \frac{h_n}{1 - e^{-m_q(t-t_r)}}, \tag{4.3}$$

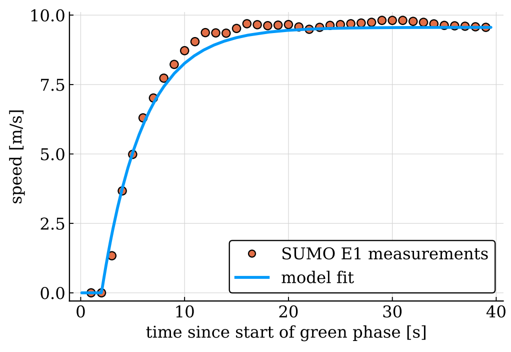
where

$$\begin{aligned}
t &= \text{time since start of green phase } [s],\\
t_r &= \text{response time of first vehicle } [s],\\
v_s(t) &= \text{discharge speed at time } t\ [km/h],\\
q_s(t) &= \text{queue discharge volume at time } t\ [veh/h],\\
h_s(t) &= \text{queue discharge headway at time } t\ [s],\\
v_n &= \text{maximum queue discharge speed } [km/h],\\
q_n &= \text{maximum queue discharge volume } [veh/h],\\
h_n &= \text{minimal queue discharge headway } [s],\\
m_v &= \text{a parameter in the speed model } [-],\\
m_q &= \text{a parameter in the discharge volume model } [-].
\end{aligned}$$

I estimated the parameters of the model to resemble behaviour of *vehicle types* in my scenario in SUMO. To do this, I created a new scenario (test scenario) with just a straight road with one TLJ, I put an *E1 detector* right to the stop line of the *junction* and laid down an *E2 detector* on the *lane* in front of the *junction*. A screenshot of the test scenario is in Fig. 4.5. Then I defined a *flow* of 20 *vehicles* to depart, just to stop at red lights. When all *vehicles* were standing still in the queue, a phase switched to green. Measurements of speed from *E1 detector* are plotted in 4.4a. The parameters of equation 4.1 were found using least squares regression. The result is in 4.4b. I obtained parameters m_v and v_n. SUMO measures speed in m/s, so the parameter v_n had to be converted to km/h.



(a) *Speed at junction stop line*



(b) *Equation 4.1 fitted to measured points*

Figure 4.4: *Speed measurements at the stop line and model fit*

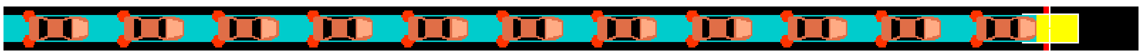


Figure 4.5: *A queue of vehicles in SUMO. Yellow rectangle is E1 detector, light blue area is the E2 detector.*

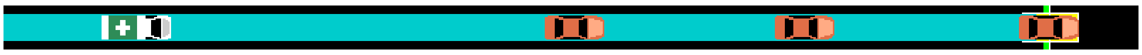


Figure 4.6: *The resulting behaviour of the queue discharge based prefernce. The EV reaches the queue at the time when all vehicles are going at saturation speed.*

The maximal queue discharge volume q_n (or saturation flow) was estimated from practical formula for "through movements at isolated intersections". In comparison with simulated result, it is reasonably accurate:

$$q_n = 1012 + 24.5 \cdot v_n. \tag{4.4}$$

For other development, we need to define average vehicle length L_v and average space gap between standing vehicles in the queue L_{sj}. I assume we can get these values (in SUMO, they are `length` and `minGap` of *vehicle type* definition). Their sum results in average jam spacing $L_{hj} = L_v + L_{sj}$. Parameters m_v and m_q from equations 4.1 and 4.2 are coupled by the relation

$$m_q = 1000 \cdot m_v \frac{v_n}{q_n L_{hj}}, \tag{4.5}$$

so we can compute the parameter m_q. The headway 4.3 at time t is in the relation with flow 4.2:

$$h_s(t) = 3600/q_s(t), \tag{4.6}$$

so the minimal (saturation) headway time follows the same relation

$$h_n = 3600/q_n. \tag{4.7}$$

Now we can compute average spacing of a queue moving at saturation speed (saturation spacing) as

$$L_{hn} = 1000 \cdot \frac{v_n}{q_n}. \tag{4.8}$$

Average driver response time to start moving after the vehicle in front of him starts moving is

$$t_x = h_n - 3.6 \cdot \frac{L_{hj}}{v_n} \tag{4.9}$$

Acceleration delay of a single vehicle is

$$d_a = t_s + h_n - t_x, \tag{4.10}$$

where t_s is the start loss of the first vehicle in the queue. The time to accelerate to saturation speed can be estimated as

$$t_a = \frac{v_n}{3.6 \cdot a_a}, \tag{4.11}$$

where a_a is average acceleration rate that can be computed from

$$a_a = \frac{(1 - m_a) \cdot v_s}{3.6 \cdot d_a}. \tag{4.12}$$

m_a in the last equation is a parameter of average acceleration characteristic which can be estimated from experimentally derived formula

$$m_a = 0.467 + 0.002 \cdot v_n. \tag{4.13}$$

All parameters of the model and their values are listed in Table 4.1. Assuming that the queue has w_0 vehicles, then the time when the last vehicle is going at saturation speed after green phase starts, could be computed by the following formula

$$LT = w_0 \cdot t_x + t_a. \tag{4.14}$$

Step 4. and 5. of the algorithm is to compute the time that the EV needs to surpass the remaining distance of the vehicles in motion in front of the TLJ (XT). Let $w(t)$ be a dynamic variable describing the number of cars at an approach to the intersection at time t and the initial condition is $w(0) = w_0$. Then assuming no other inflow of vehicles, the following first order differential equation holds:

$$\dot{w}(t) = -q_s(t). \tag{4.15}$$

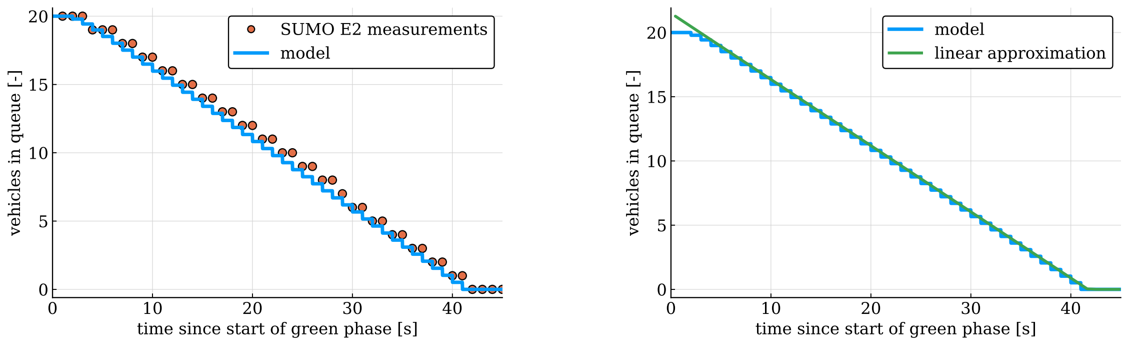
I used a numerical solver to do the first order euler discretization and solve. I plotted the solution against measurements from *E2 detector*. The result is in Fig. 4.7a. The solution of 4.15 can be approximated by a linear function of time t:

$$w_{lin}(t) = w_0 + 1.5 - \frac{q_n}{3600} \cdot t, \tag{4.16}$$

that is depicted in 4.7b. The approximation is used by TLC to compute the time XT:

$$XT = \frac{w_{lin}(LT) \cdot L_{hn}}{EV_{speed}}. \tag{4.17}$$

The time to start preference can now be computed in step 6. of the algorithm. The screenshot in 4.6 depicts the situation. Figures 4.9 and 4.10 show FCD outputs of the test scenario.



(a) *Queue discharge of 20 vehicles.*

(b) *Linear approximation of the solution w_{lin}*

Figure 4.7: *Queue discharge and linear approximation*

v_n	q_n	h_n	m_v	m_q
34.4	1855.5	1.9	0.25	0.68
L_v	L_{sj}	L_{hj}	L_{hn}	t_s
4.3	2.5	6.8	18.5	1.0
t_x	t_a	d_a	a_a	m_a
1.23	3.7	1.71	2.6	0.54

Table 4.1: *All parameters of the queue discharge model*

4.3.3 Implementation

In my simulations, EVs do not send their actual speed EV_{speed}. Instead, I have defined a parameter called `EV_desired_speed` that is set to maximal value that the EV can go: 75 km/h is set for main simulation runs, 50 km/h for testing scenario. The reason is to make the algorithm more conservative. It starts the preemption a little bit sooner because EV_{speed} is always <= `EV_desired_speed`.

I have also introduced `t_cons=5` (conservative time constant). The constant is substracted from the time to start preference to make it even more conservative, especially to cope with 5 second rule. I did not use `t_cons` in the test scenario.

Even though it would be possible to model this mode using only SUMO and TraCI, I decided to go into more details. I believe it makes the simulation more trustworthy and opens possibilities for new ideas. Thus, this mode is the only reason why I have bothered with V2X simulations. To model this mode of prioritization, I had to implement these modules in Veins:

- `EmergencyAppl`

  Application layer of EV. The only job is to send beacons with EV_{pos} periodically.

- `ForwarderAppl`

  Application layer of *vehicles* of public transport. This application only decrements hop-count of the beacon from EV and then resends it.

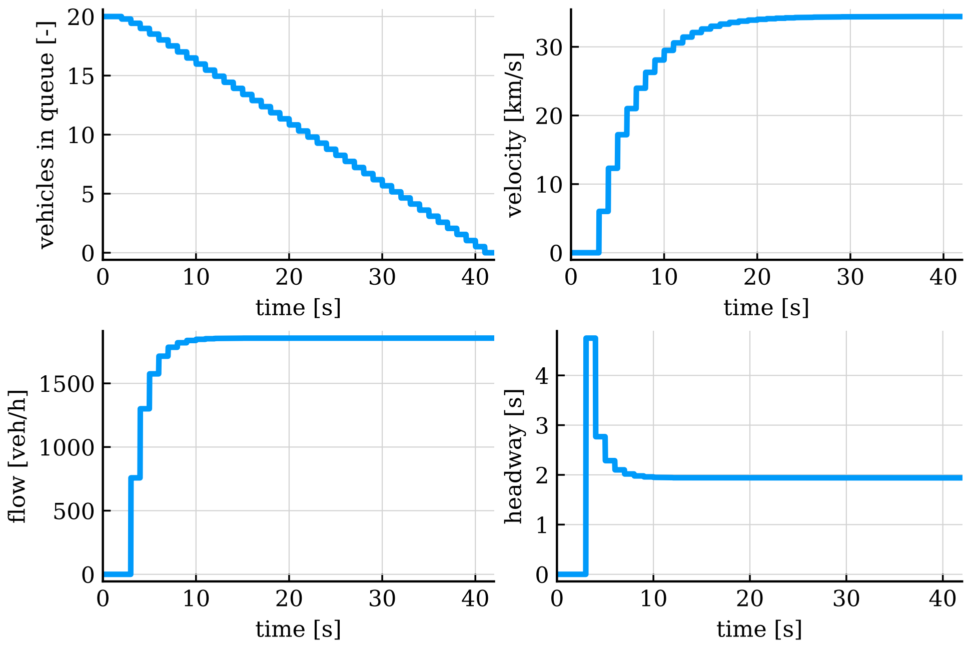


Figure 4.8: *Identified model with a solution of* (4.15) *for 20 vehicles in top left plot. TLC assumes that queues behave according to this model.*

- `TrafficLightAppl`

  Application layer of TLC. Receives beacons from EV and computes time to start preference, adds `t_cons`. Then schedules a message with instruction for `TrafficLightLogic` to start prioritization. *E2 detectors* placed on connections to the *junctions* are used to get number of vehicles in the queue. It also instruct `TrafficLightLogic` to stop preference and go back to previous program and phase when the EV is gone.

- `TrafficLightLogic`

  Implements the logic of preference switching. It tries to start priority phase immediately after receiving a message from `TrafficLightAppl`. This module also checks 5 seconds rule and creates preemption program for the EV. The traffic light program switch is realized by TraCI call. It also switches the previous program back.

Remarks about real adoption of the method

The benefit of the queue discharge based method is taking away the necessity to define the virtual border around every TLJ. This is because the TLC decides about the time of starting the preemption upon request and from actual traffic situation. Another nice feature is the fact that it can be implemented with no changes in the infrastructure. Only the algorithm needs to be programmed. Communication units are already installed.

However, if we wanted to implement the method on real TLC, we would have to deal with two problems:

1. How to obtain traffic data

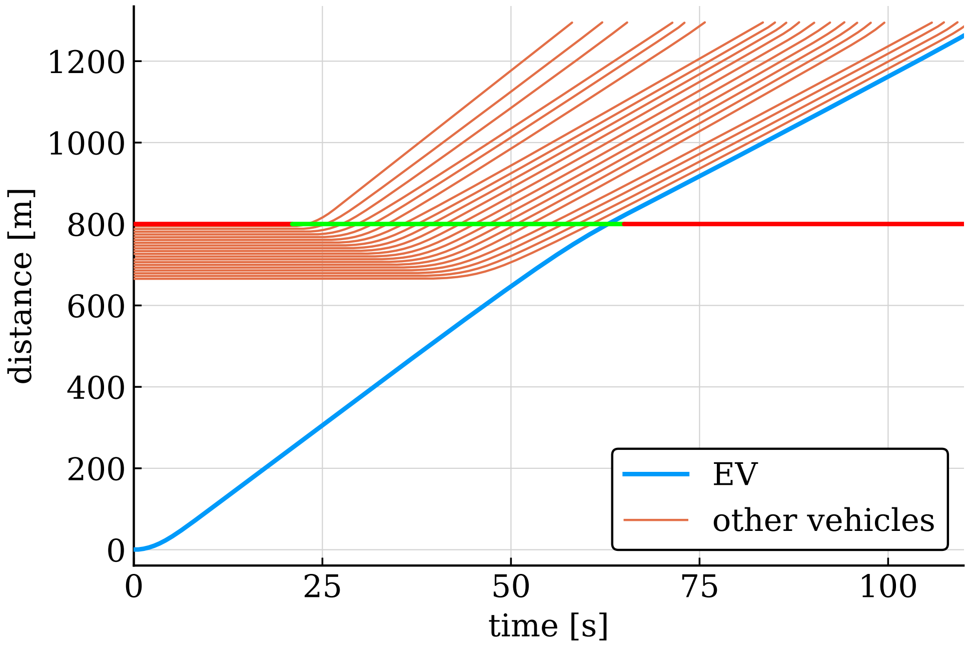
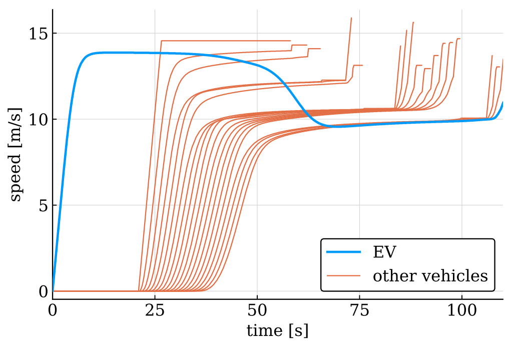


Figure 4.9: *Time-distance plot from FCD output of queue discharge based preference with 20 vehicles in the test scenario. TLJ is placed at 800 meters. The phase is depicted by red and green color. The EV has desired speed set to only* 50 *km*/*h.*

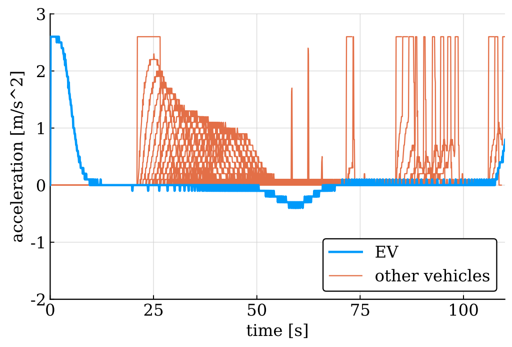
Currently, there is no precise system to count vehicles in front of the TLJ in real time. Induction loops might be used for estimation but this is not an easy problem. Another possibility is to use some online maps[2] services API.

2. Queue discharge model calibration

   The model I used is a rough estimation of the behaviour of real drivers. For real life usage, more rigorous identification is probably needed. But I think that some conservative estimate of parameters would also work.



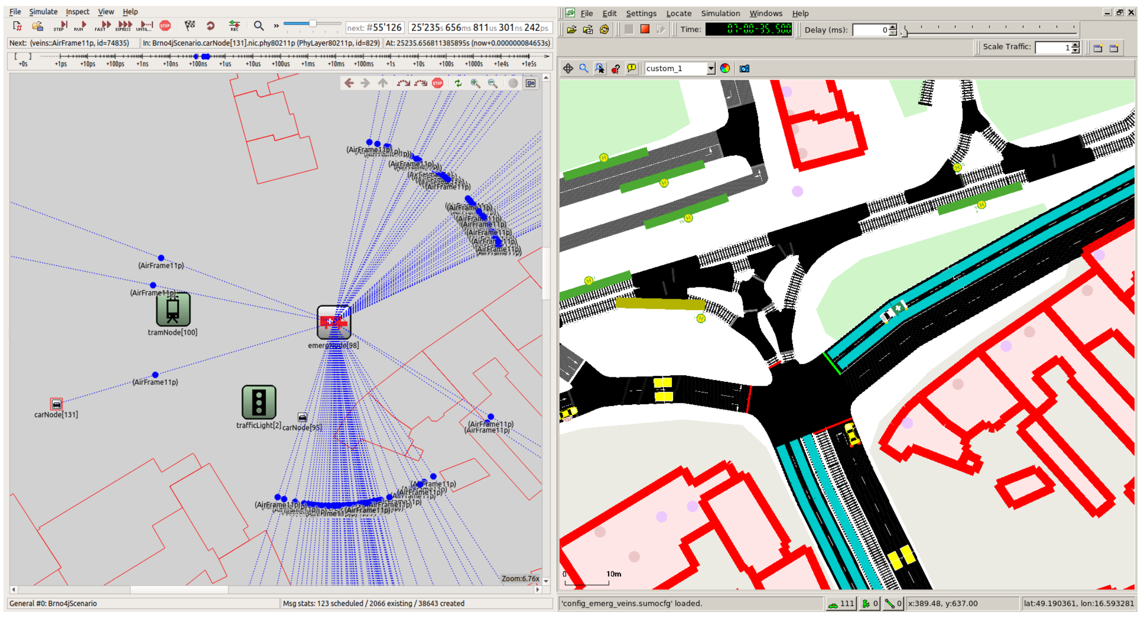
(a) *Time-speed*



(b) *Time-acceleration*

Figure 4.10: *FCD output visualisations from the test scenario.*

[2]For example HERE maps contain Traffic API `https://www.here.com/`



```
Next: (veins::AirFrame11p, id=74835)   In: Brno4jScenario.carNode[131].nic.phy80211p (PhyLayer80211p, id=829)   At: 25235.656811385895s (now+0.000000084653s)
```

```
General #0: Brno4jScenario   Msg stats: 123 scheduled / 2066 existing / 38643 created
```

```
'config_emerg_veins.sumocfg' loaded.   111   0   0   x:389.48, y:637.00   lat:49.190361, lon:16.593281
```

Figure 4.11: *Running simulation of the queue discharge based method in the scenario of Brno. On the left, the simulation of V2X communication in GUI of OMNeT++. On the right, the mobility simulation in SUMO-GUI. They are coupled by* `sumo_launchd.py`.

Chapter 5

Comparison

In this chapter, I compare the outputs from extensive simulations of three implemented modes. Firstly, I describe how I collected the outputs from simulations. Then I evaluate different kinds of SUMO outputs. I were interested in *tripinfo output* of EVs, *traffic light switches output*, *tripinfo output* of other *vehicles*.

5.1 Methodology

To evaluate performance of all three modes of preference, I conducted an experiment. I chose the morning peak hours of three days:

- Wednesday, November 14, 2018, 7:00 a.m.-9:00 a.m.,
- Thursday, November 15, 2018, 7:00 a.m.-9:00 a.m.,
- Tuesday, November 20, 2018, 7:00 a.m.-9:00 a.m.,

and I repeated the following procedure for each of those time intervals.

1. EV is set to depart at given time t_{depart}.
2. Simulation starts at time $t_{depart} - \Delta T$
3. Simulation ends at time $t_{depart} + \Delta T$
4. Outputs are collected from the simulation run.
5. The seed for random number generator is changed and step 2. is repeated until ten repetitions are over.
6. Set $t_{depart} = t_{depart} + t_{step}$ and repeat from 1.

In other words, to variate the traffic situation around the EV, I set the EV to depart at different times. I simulate ΔT before the EV departs to warm up the simulation (so other *vehicles* have time to fill the scenario). And I simulate ΔT after the departure of EV, so I will be able to measure impact on the other traffic.

I wrote Python script that can run the simulations. It takes initial time of departure of EV t_{dep}, time by how much to variate time of departure t_{step}, number of runs N and $mode$ specifier (whether to simulate no preference mode, distance based mode or queue discharge

Algorithm 1: *Pseudocode for batch simulations*

```
init: $seeds[10]$, $\Delta T$
getOptions: $t_{dep}$, $t_{step}$, $N$, $mode$
for $i$ in range(0, $N$) do
    $t_{dep} := t_{dep} + i \cdot t_{step}$
    setEvDepartAt($t_{dep}$)
    for $s$ in $seeds$ do
        beg $= t_{dep} - \Delta T$
        end $= t_{dep} + \Delta T$
        modifyXmlFiles()
        runSimulation(beg, end, mode=$mode$, seed=$s$)
        collectOutputs()
    end
end
```

based mode). The parameter N is a little bit misleading, because every simulation is repeated ten times with different seed. So if I start the script with $N = 2$, then 20 simulations will run, where the first 10 of them will be simulated with the EV departure time t_{step} before the last 10 of runs. I run the procedure using following values:

$$\begin{aligned} t_{dep} &= 25200, \\ \Delta T &= 300, \\ t_{step} &= 30, \\ N &= 240. \end{aligned}$$

The number 25200 corresponds to 7:00 a.m. in seconds from midnight. With 240 repetitions, incrementing t_{dep} by t_{step} each time, we end up 30 seconds before 9:00 a.m. The pseudocode of the algorithm is written in alg. 1. The outputs are distinguished by a unique prefix, holding the information about the source of the output:

$mode$_t_{dep}_$seed$_output_type.xml.

I run 240 simulation per mode per day and everything is repeated 10 times, totalling in 21600 simulation runs. The reason why I decided to repeat every simulation 10 times with different *seed* is to make sure that the results are not random. After the first simulation batch, I noticed that differences in trip durations for other vehicles are very small. Repeating the simulations with different *seed* produced similar results, so I am confident that the results presented in this chapter are replicable even with different parameters of the simulation. To discuss the results, I show the plots from one day (November 14, 2018) and one particular *seed*.

5.2 Trip duration of emergency vehicle

The plot 5.1 shows durations of EV trips throughout the test interval. It is apparent that the duration of EV trip without preference varies wildly. That is partially because of the problems discussed in 4.1 but also because the EV might hit TLJ with red light in very congested state. With different times of departure of the EV, different traffic lights phases might be encountered

EV trip time	Mean [s]	Std [s]
No preference	105.38	31.18
Distance based	74.12	3.20
Queue discharge based	76.62	5.76

Table 5.1: *Values from Fig. 5.1*

on its path. As for two preference modes, distance based method performs slightly better for EVs.

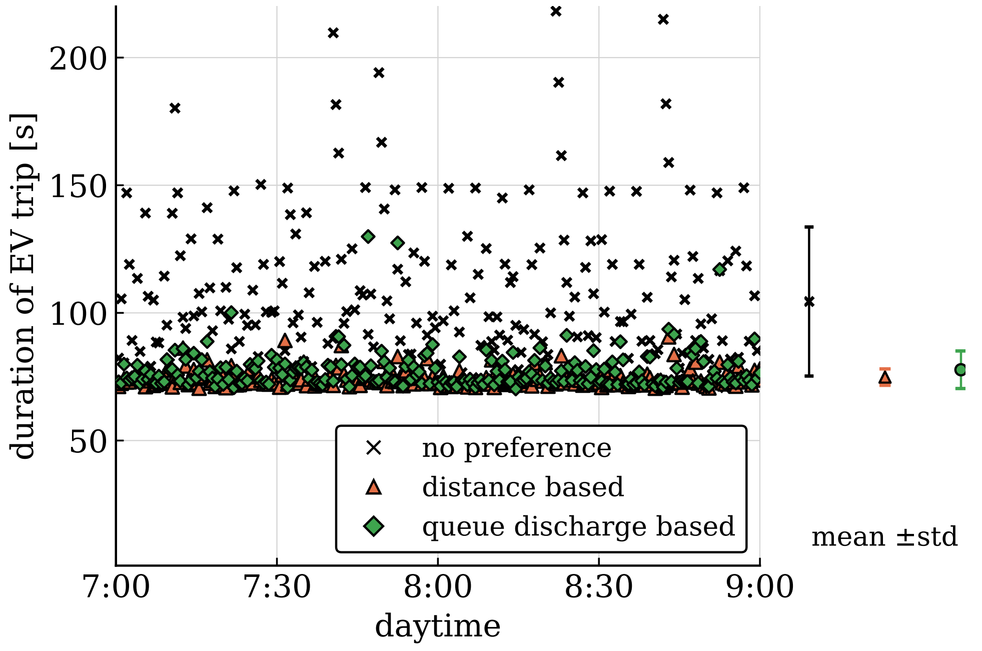


Figure 5.1: *EV trip durations for each mode throughout the testing interval.*

5.3 Duration of traffic light preference

This is the comparison of only preemption modes. Fig. 5.2 shows the duration of how long was each of the TLJs in preference phase. Queue discharge based method takes the traffic situation into account so the duration of prioritization varies according to it. I think this plot is reasonable result. Maybe, if the scenario was more congested, then the means would be closer to each other. However, I haven't tried this. The duration of preference in distance based mode varies only because of 5 seconds rule.

Duration of preference phase	1.01		1.02		1.03		2.06	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Distance based	27.38	2.41	26.17	2.17	25.36	2.48	28.60	2.16
Queue discharge based	23.18	5.36	20.96	8.36	19.44	6.51	19.31	4.41

Table 5.2: *Durations of preference phases. Values are visualised in Fig. 5.2*

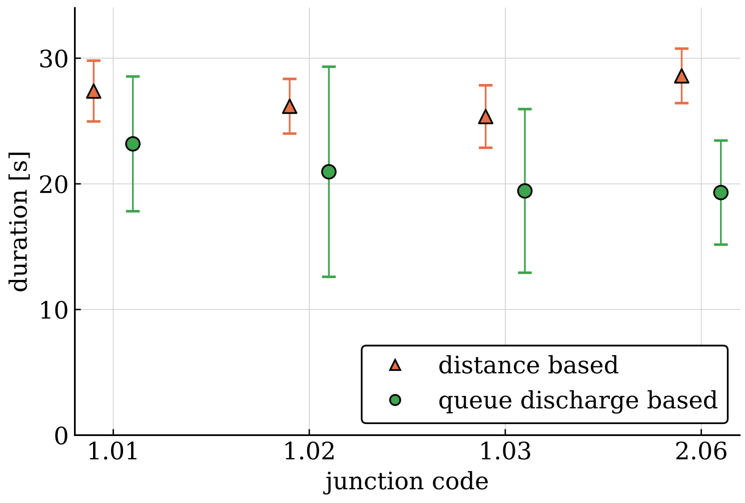


Figure 5.2: *Duration of preference phase of each of four TLJs*

5.4 Waiting time of other vehicles

Fig. 5.3 shows the means of aggregated waiting times of other vehicles and their standard deviations over respective 240 simulation runs. Waiting time is being counted when a *vehicle* is stopped involuntarily. Mean waiting time rose by around 3 seconds in the case of distance based preference and by around 2 seconds in the case of queue discharge based preference, compared to the mode with no preference.

Table 5.3 shows more *tripinfo* data of other vehicles. The best indicator of the impact of preference is the waiting time of other vehicles. It can be seen from the table that both, distance based mode and queue discharge based mode increase the mean waiting time of other vehicles in all three days.

In the table, I distinguish between all and finished trips because SUMO simulates less vehicles in preference modes. That is because the occupancy of certain *edges* might be higher at the moment and SUMO decides to insert a new vehicle later. The mean waiting time is aggregated for all vehicles, even unfinished. The table shows the data for all three days.

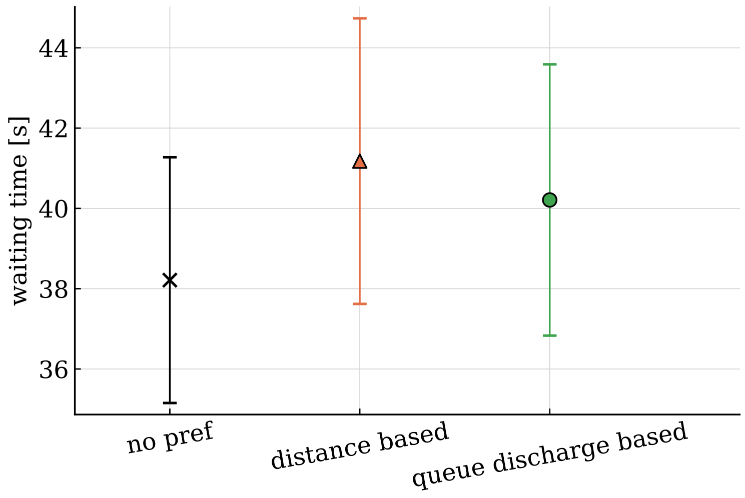


Figure 5.3: *Aggregated waiting time of other vehicles.*

Wednesday, November 14, 2018, 7:00 a.m.–9:00 a.m.			
Mode	No preference	Distance based	Queue discharge based
Trips [#] (finished)	510.75	503.88	504.81
Mean duration [s] (finished)	101.87	105.12	104.12
Max duration [s] (finished)	409.132	421.82	417.86
Waiting time [s] (all)	38.21	41.18	40.21
Max waiting time [s] (all)	278.11	284.25	277.77
Thursday, November 15, 2018, 7:00 a.m.–9:00 a.m.			
Mode	No preference	Distance based	Queue discharge based
Trips [#] (finished)	489.07	481.93	493.92
Mean duration [s] (finished)	106.03	109.17	108.15
Max duration [s] (finished)	416.50	427.75	423.65
Waiting time [s] (all)	41.61	44.47	43.40
Max waiting time [s] (all)	285.78	294.43	292.59
Tuesday, November 20, 2018, 7:00 a.m.–9:00 a.m.			
Mode	No preference	Distance based	Queue discharge based
Trips [#] (finished)	500.75	492.5	494.48
Mean duration [s] (finished)	103.03	106.59	105.31
Max duration [s] (finished)	417.92	425.12	422.91
Waiting time [s] (all)	39.69	42.88	41.62
Max waiting time [s] (all)	283.83	286.44	282.08

Table 5.3: *Comparison of the experiment done over three days in morning peak time. I simulate 5 minutes before and 5 minutes after EV departs. The time of departure of EV goes from 7:00 a.m. till 9:00 a.m., every 30 seconds. That gives 240 simulation runs per mode per day (2160 simulation runs). Values listed in the table are* ***mean values*** *of respective 240 simulation runs. Similar results were obtained for other batches.*

Chapter 6

Conclusion

In the last chapter I summarize what have been accomplished in this thesis and I propose a few projects that can pick up the results of this thesis.

The task was to develop a microscopic traffic simulation of a real area, simulate two modes of EVs going through them and then study the impact on the other traffic. All of these tasks were done successfully.

- Firstly, I described the tools which I used in the work. This resulted in the text that can be used as a tutorial for newcomers of the research group (Chapter 2).
- The simulation scenario based on real measurements was created and prepared for V2X simulations (Chapter 3).
- I described three modes of preference of EVs and developed programs to simulate them. Moreover, the queue discharge based preference uses a new approach and exhibits promising results (Chapter 4).
- Outputs from simulations were analysed and all three methods were compared against each other (Chapter 5).

I defined two problems in Section 1.1. Based on my simulations, I conclude that prioritization of EVs indeed prolongs the journey for other road users. It is not a big difference but it is definitely measurable. Then I proposed a new approach which removes the need of a virtual border and is a good trade-off between performance for EVs and for the other traffic.

6.1 Future work

The biggest improvement of the scenario would be modelling dynamic traffic lights with preference for public transport and simulate EVs in this environment. It might be interesting to study the impact of prioritization of EVs to the public transport.

There is a possibility to extend the scenario by several other adjacent junctions, that are newly equipped with V2X enabled communication units. This results from C-ROADS project effort.

Modelling pedestrians and zebra crossings would certainly make the simulation scenario more lively.

The scenario might be used as a testbed for developing new preference algorithms.

Bibliography

[1] R. Sánchez-Mangas, A. Garcia-Ferrer, A. de Juan, and A. Martín Arroyo, "The probability of death in road traffic accidents. how important is a quick medical response?" *Accident; analysis and prevention*, vol. 42, pp. 1048–56, 07 2010.

[2] R. P Gonzalez, G. R Cummings, H. Phelan, M. Mulekar, and C. B Rodning, "Does increased emergency medical services prehospital time affect patient mortality in rural motor vehicle crash? a statewide analysis," *American journal of surgery*, vol. 197, pp. 30–4, 06 2008.

[3] M. Fogue, P. Garrido, F. Martinez, J.-C. Cano, C. Calafate, and P. Manzoni, "Automatic accident detection: Assistance through communication technologies and vehicles," *IEEE Vehicular Technology Magazine*, vol. 7, pp. 90–100, 09 2012.

[4] S. Burke, E. Salas, and J. Peter Kincaid, "Emergency vehicles that become accident statistics: Understanding and limiting accidents involving emergency vehicles," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 45, pp. 508–512, 10 2001.

[5] L. Bieker-Walz, "Traffic safety evaluations for emgergency vehicles," *Young Researchers Seminar*, 06 2015.

[6] L. Bieker-Walz, "Cooperative traffic management for emergency vehicles in the city of bologna," *SUMO 2017 – Towards Simulation for Autonomous MobilityVolume: 31*, 05 2017.

[7] M. Treiber, A. Kesting, and C. Thiemann, *Traffic Flow Dynamics: Data, Models and Simulation*. Springer Berlin Heidelberg, 2012. [Online]. Available: https://books.google.cz/books?id=Xlsa9aaLc_QC

[8] S. P. Hoogendoorn and P. H. L. Bovy, "State-of-the-art of vehicular traffic flow modelling," in *Delft University of Technology, Delft, The*, 2001, pp. 283–303.

[9] T. Bellemans, B. De Schutter, and B. De Moor, "Models for traffic control," *Journal A*, vol. 43, no. 3–4, pp. 13–22, 2002.

[10] K. Nagel and M. Schreckenberg, "A cellular automaton model for freeway traffic," *J. Phys. I France.*, vol. 2, no. 12, 1992.

[11] S. Krauß, "Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics," *D L R - Forschungsberichte*, 01 1998.

[12] B. Ciuffo, V. Punzo, and M. Montanino, "Thirty years of gipps' car-following model," *Transportation Research Record Journal of the Transportation Research Board*, vol. 2315, pp. 89–99, 12 2012.

[13] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using sumo," in *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018. [Online]. Available: https://elib.dlr.de/124092/

[14] M. Behrisch, L. Bieker-Walz, J. Erdmann, M. Weber (Geb. Knocke, D. Krajzewicz, and P. Wagner, *Evolution of SUMO's Simulation Model*, 01 2014, pp. 1–21.

[15] J. Erdmann, "Lane-changing model in sumo," in *SUMO2014*, ser. Reports of the DLR-Institute of Transportation SystemsProceedings, vol. 24. Deutsches Zentrum für Luft- und Raumfahrt e.V., May 2014, pp. 77–88. [Online]. Available: https://elib.dlr.de/89233/

[16] Wikipedia contributors, "Mersenne twister — Wikipedia, the free encyclopedia," 2019, [Online; accessed 18-May-2019]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Mersenne_Twister&oldid=896072298

[17] A. Wegener, M. Piórkowski, M. Raya, H. Hellbrück, S. Fischer, and J.-P. Hubaux, "Traci: An interface for coupling road traffic and network simulators," in *Proceedings of the 11th Communications and Networking Simulation Symposium*, ser. CNS '08. New York, NY, USA: ACM, 2008, pp. 155–163. [Online]. Available: http://doi.acm.org/10.1145/1400713.1400740

[18] C. Sommer and F. Dressler, *Vehicular Networking*. Cambridge University Press, 2014. [Online]. Available: http://book.car2x.org/

[19] 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, January 2011.

[20] OpenStreetMap Wiki, "Main page — openstreetmap wiki,," 2014, [Online; accessed 7-May-2019]. [Online]. Available: https://wiki.openstreetmap.org

[21] T. Nguyen, D. Krajzewicz, M. Fullerton, and E. Nicolay, "Dfrouter—estimation of vehicle routes from cross-section measurements," *Lecture Notes in Control and Information Sciences*, vol. 13, pp. 3–23, 01 2015.

[22] J. Zambrano-Martinez, C. Calafate, D. Soler, and J.-C. Cano, "Towards realistic urban traffic experiments using dfrouter: Heuristic, validation and extensions," *Sensors*, vol. 17, pp. 1–29, 12 2017.

[23] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian Journal of Mathematics*, vol. 8, p. 399–404, 1956.

[24] L. Bieker-Walz, M. Behrisch, and M. Junghans, "Analysis of the traffic behavior of emergency vehicles in a microscopic traffic simulation," 05 2018.

[25] L. Bieker-Walz, Y.-P. Flötteröd, A. Sohr, and S. Ruppe, "Gaining insight into routing behaviors of emergency vehicle from real-world trajectories," 10 2018.

[26] L. Bieker-Walz, "Self-organizing traffic management for emergency vehicles," 09 2017.

[27] H. Noori, L. Fu, and S. Shiravi, “A connected vehicle based traffic signal control strategy for emergency vehicle preemption,” *in Proc. of the Transportation Research Board 95th Annual Meeting*, 01 2016.

[28] R. Akçelik, M. Besley, and R. Roper, “Fundamental relationships for traffic flows at signalised intersections.” 09 1999.

[29] R. Akçelik and M. Besley, “Queue discharge flow and speed models for signalised intersections,” *Transportation and Traffic Theory in the 21st Century, proceedings of the 15th International Symposium on Transportation and Traffic Theory*, pp. 99–118, 08 2002.