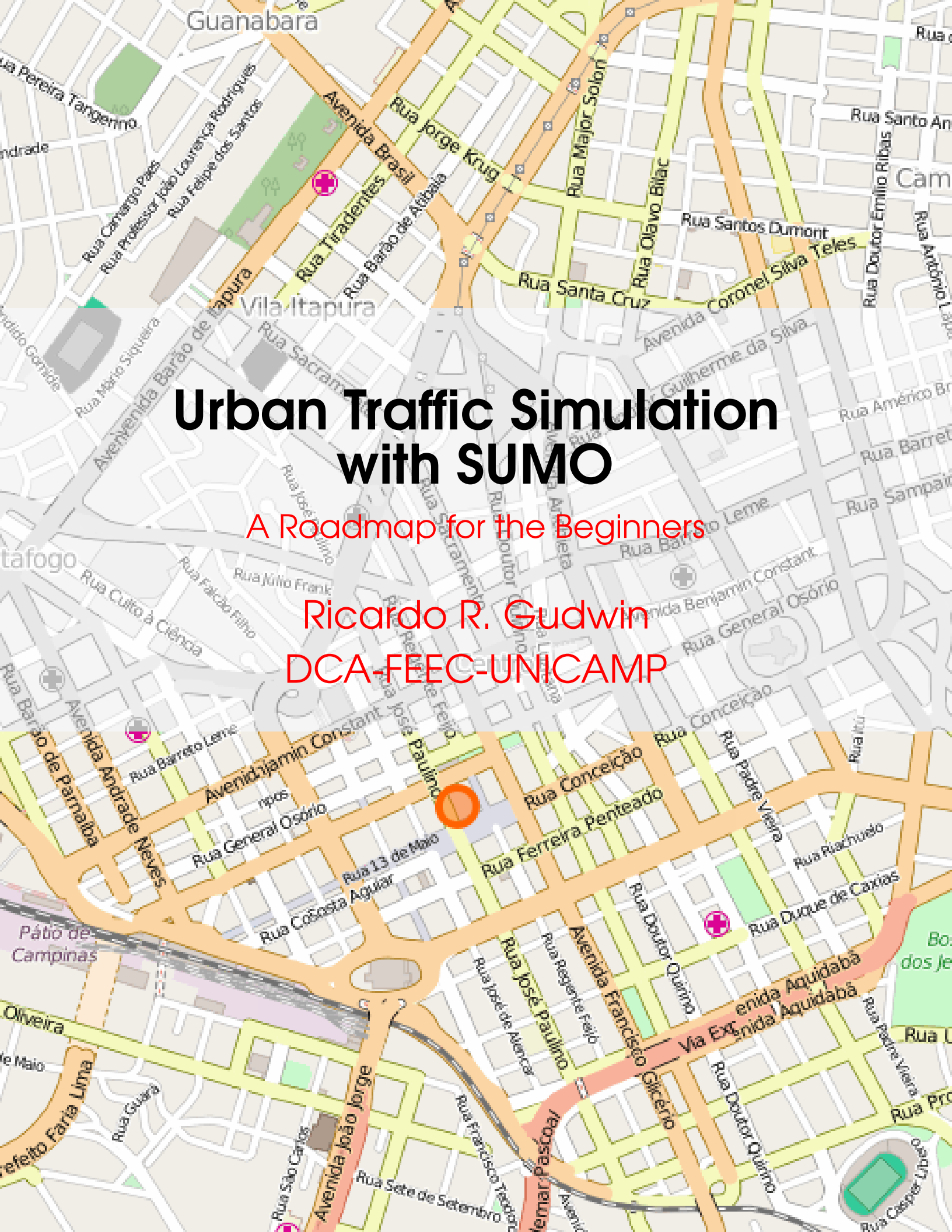


Urban Traffic Simulation with SUMO

A Roadmap for the Beginners

Ricardo R. Gudwin
DCA-FEEC-UNICAMP

PUBLISHED BY DCA-FEEC-UNICAMP

HTTP://WWW.DCA.FEE.UNICAMP.BR/

First printing, July 2016

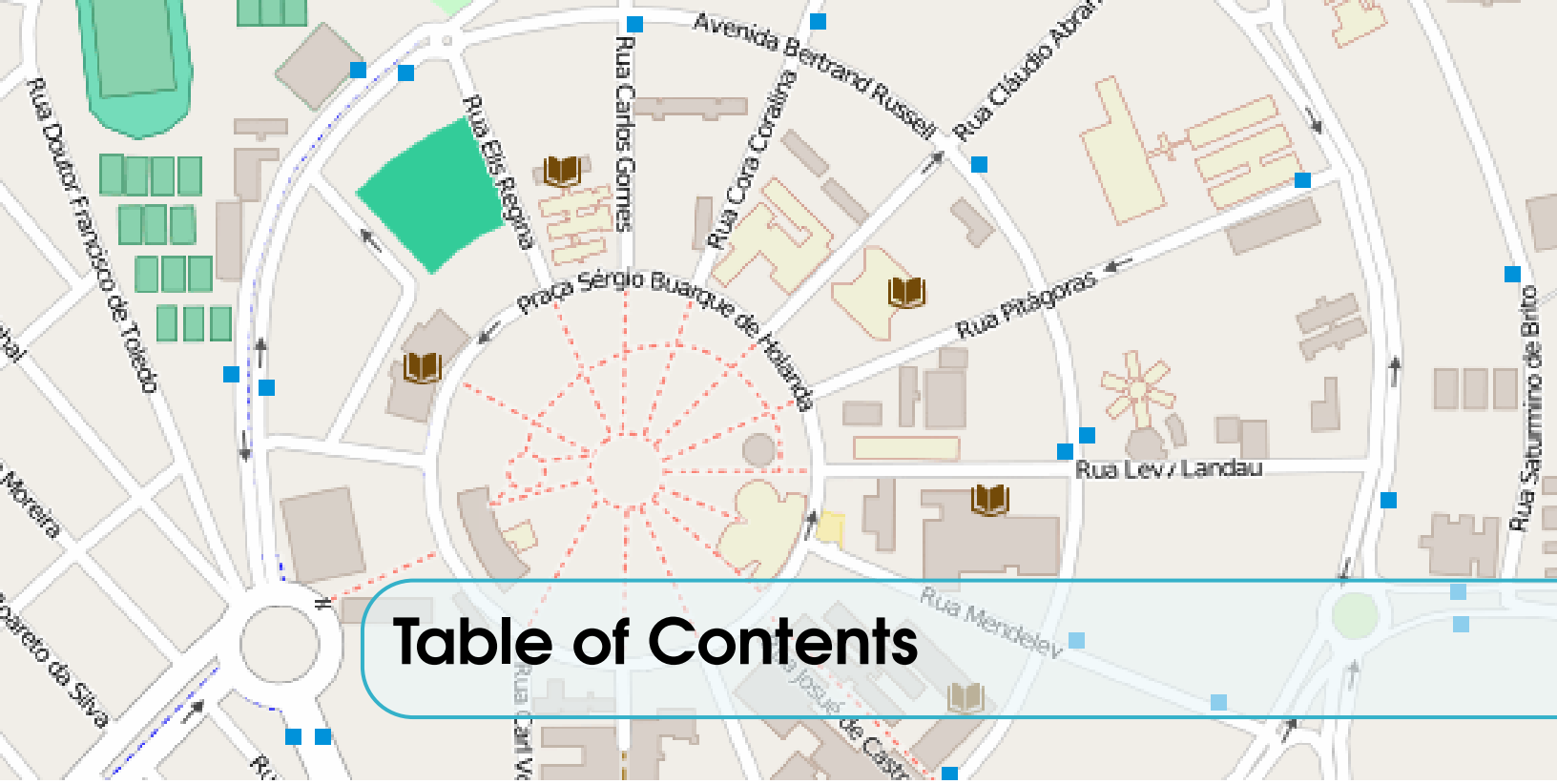


Table of Contents

1. Introduction

1.1 Motivation

The motivation for this document is to bring forward a roadmap of SUMO, the Simulator of Urban MObility, a software tool being developed by the DLR - Institute of Transportation Systems of DLR, the National Aeronautics and Space Research Center of the Federal Republic of Germany in Berlin[1]. SUMO is an open source (licensed under the GPL), highly portable, microscopic and continuous road traffic simulation package designed to handle large road networks. The simulation platform offers many features, like microscopic simulation[2], online interaction[3] and the simulation of multimodal traffic[4]. Time schedules of traffic lights can be imported or generated automatically by SUMO and there are no artificial limitations in the network size and the number of simulated vehicles. SUMO uses its own file formats for traffic networks, but it is able to import maps encoded in many public available formats like OpenStreetMap, VISUM, VISSIM and NavTeq. SUMO is implemented in C++ and uses only portable libraries.

There is extensive documentation about SUMO, on their web page available at:

http://sumo.dlr.de/wiki/Main_Page

This documentation might be, nevertheless, a little bit scary for a beginner user. The main motivation for this technical report is to provide an introduction to researchers interested in using SUMO as their research simulation tool, but do not have the time and do not want the burden to browse the many web pages in this web site in order to start using SUMO and running their simulation. The major information here available can be also found on

[1] See http://sumo.dlr.de

[2] vehicles, pedestrians and public transportation are modeled explicitly as individual entities by the simulator

[3] It is possible to interact with the simulation by external programs, using TraCI, the Traffic Command Interface

[4] Different modalities of traffic, by cars, trains, buses and pedestrians

these web pages, which are re-organized here in a more pedagogical format, appropriate for a beginner user, but without loosing the necessary deepness when it is important for the aforementioned goals.

SUMO is being used in several research projects found in the literature, trying to answer a large variety of research questions. Among these questions:

- Evaluate the performance of traffic lights, including the evaluation of modern algorithms up to the evaluation of weekly timing plans.
- Vehicle route choice has been investigated, including the development of new methods, the evaluation of eco-aware routing based on pollutant emission, and investigations on network-wide influences of autonomous route choice.
- SUMO was used to provide traffic forecasts for authorities of the City of Cologne during the Pope's visit in 2005 and during the Soccer World Cup 2006.
- SUMO was used to support simulated in-vehicle telephony behavior for evaluating the performance of GSM-based traffic surveillance.
- SUMO is widely used by the V2X community for both, providing realistic vehicle traces, and for evaluating applications in an on-line loop with a network simulator.

This document is structured in the following way. In chapter 2, we provide the user with the basic information to install SUMO in different platforms. In chapter 3 we provide the essential concepts for understanding SUMO and its main components. At the end of chapter 3, the reader should be able to prepare a basic network topology and a minimum traffic demand, being able to run his/her first simulations with SUMO. In chapter 4 we provide information about TraCI, the Traffic Command Interface, explaining how a simulation with SUMO can be connected to an external controller, in order to control the traffic running within the simulation.

The goal of this document is not to replace SUMO documentation, but organize it in a comprehensive way, making it easier for a novice to start using SUMO for his/her experiments. Many links to the SUMO documentation are provided within the text, and using any modern PDF reader, the reader might easily start browsing the SUMO documentation and getting access to the details necessary to configure SUMO properly for running experiments.

2. Installing SUMO

2.1 Downloading SUMO

In order to use SUMO, the first steps are determining your operational system and finding a suitable SUMO version to download and install. SUMO is available for Windows, Linux and MacOS. The information necessary for downloading the right piece of code for your operational system can be found in:

http://sumo.dlr.de/wiki/Downloads

Depending on your operational system, SUMO can be downloaded and installed in a single step. For example, if you use the last version of Ubuntu, SUMO is already included in the Ubuntu repository, and can be downloaded and installed simply by typing:

```
sudo apt-get install sumo sumo-tools
```

If you use MacOS, SUMO can be downloaded through the Macports repositories. The MacPorts Project is an open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading many open-source software on the MacOS operating system. You may find more information on Macports here:

https://www.macports.org/

After installing Macports, make sure that an X server is installed (e.g. XQuartz) and that you have a terminal application. Now start a terminal session and run:

```
sudo port install sumo
```

For either Windows, Linux or MacOS, if you want to be sure that you have the latest version of SUMO, you may download the source code and compile it, but usually this is not necessary. The latest version of SUMO code can be found at GitHub:

https://github.com/planetsumo/sumo

2.2 SUMO Installation

After downloading SUMO, depending on the way you downloaded the code, you might still have to install SUMO. In the Windows system, the traditional method of running the downloaded executable file will proceed and install SUMO. In Linux, if you used `apt-get` to download SUMO, it is already installed. The same happens if you used Macports in MacOS. If you downloaded the source code, you still need to compile and install the code. In Windows, all the SUMO code is stored in a folder, which you decide where during the process of installation. In Linux and MacOS, the installation will follow the Unix convention, and your executable files will be located at somewhere like `/usr/bin`.

SUMO is in fact the collection of many different programs, which must be used conjointly in order to setup and run a simulation. The following programs are the most important ones:

- Programs for running a simulation:
  SUMO: The microscopic simulation with no visualization; command line application
  SUMO-GUI: The microscopic simulation with a graphical user interface
- Programs for creating a network topology on which to run the simulation
  NETCONVERT: Network importer and generator; reads road networks from different formats and converts them into the SUMO-format
  NETEDIT: A graphical network editor.
  NETGENERATE: Generates abstract networks for the SUMO-simulation
- Programs for Traffic Demand Definition
  DUAROUTER: Computes fastest routes through the network, importing different types of demand description. Performs the DUA (Dynamic User Assignment)
  JTRROUTER: Computes routes using junction turning percentages
  DFROUTER: Computes routes from induction loop measurements
  OD2TRIPS: Decomposes O/D-matrices into single vehicle trips
  ACTIVITYGEN: Generates a demand based on mobility wishes of a modeled population
- Other programs
  POLYCONVERT: Imports points of interest and polygons from different formats and translates them into a description that may be visualized by SUMO-GUI

These programs comprise the main SUMO installation. There are, though, many other scripts which were developed by the community, which might be useful, and are not

distributed with the SUMO source code. In Linux, these additional tools are available through the `sumo-tools` package, and stored at `/usr/share/sumo/tools`. These include Shell scripts, Python scripts, Java libraries and Java programs. Many of these scripts use the main SUMO programs in an optimized way.

More information on SUMO tools can be found at:

http://sumo.dlr.de/wiki/Tools/Main

3. Understanding SUMO

3.1 Requirements for Running a SUMO Simulation

There are two ways for running a SUMO simulation. The first one is through the command `sumo` and the second one through the command `sumo-gui`. Both of them are command line programs. The difference between them is that `sumo` runs the simulation in background, without a graphical user interface. In this case, all the simulation runs in background. Using the `sumo-gui` program, a graphical user interface, showing the running simulation is presented to the user, which can interact with the simulation through this user interface.

There are basically two different pieces of information necessary in order to start a SUMO simulation:

- A Network Topology
- A Traffic Pattern Demand

A Network Topology comprises a network of roads, railways, pedestrian ways, aquatic routes or other means of moving cars, buses, trams, trucks, trains, boats or people.

A Traffic Pattern Demand comprises the cars, buses, trams, trucks, trains, boats or people moving around, in a given pattern along the network.

These two requirements can be unified in terms of a *configuration*, which needs to be defined in order to run a simulation. A *configuration* can be defined in SUMO in an XML configuration file (see Box 3.1)

Besides a Network File, describing the network topology (*.net.xml) and one or more Route Files, describing the traffic pattern in terms of vehicles (*.rou.xml), you might also specify Additional Files (*.add.xml). These additional files are optional, and describe add-ons for the simulation, like e.g. the definition of induction loops or other kinds of sensors installed in the network, a background image containing the air photography of the involved region, giving a more precise visualization of the environment, POIs (Points Of Interest), like building names, shopping stores, or other landmark which you want to

Box 3.1: Example of a configuration XML file: *example.sumocfg*

```
<configuration>
    <input>
        <net-file value="example.net.xml"/>
        <route-files value="example.rou.xml"/>
        <additional-files value="example.add.xml"/>
    </input>
</configuration>
```

be included in the visualization of the simulation, or either polygonal zones you want to represent in the simulation visualization.

Given a configuration file *example.sumocfg*, a simulation can be started in SUMO by simply typing in a command shell:

```
sumo -c example.sumocfg
```

or

```
sumo-gui -c example.sumocfg
```

Almost every file used in SUMO package is encoded in XML. Most SUMO files (road network descriptions, route and/or demand descriptions, infrastructure descriptions, etc.) are SUMO-specific, not following any standard. For some of the file types used by SUMO, an `xsd` (XML Schema Definition) exists. A detailed description of the many different sumo files can be found at:

http://sumo.dlr.de/wiki/Other/File_Extensions

3.2 The Network Topology

A **SUMO network file** (*.net.xml) describes the traffic-related part of a map, the roads and intersections where the simulated vehicles run along or across. At a coarse scale, a SUMO network is a directed graph. Nodes, usually named **junctions** in SUMO-context, represent intersections, and **edges** roads or streets. Note that edges are unidirectional. Specifically, the SUMO network contains the following information:

- a collection of *edges* representing segments of roads, streets, aquatic routes, railways or pedestrian pathways. An edge may have multiple *lanes*, including the position, shape and speed limit of every lane,
- a collection of *nodes* or *junctions*, together with the many *requests* specifying the right of way regulation of each junction
- a set of *traffic light logics* referenced by junctions,

- a set of *connections* between lanes at junctions, describing how from specific lanes in junction, it is possible to turn into other lanes going out of the junction.

Also, depending on the used input formats and set processing options, one can also find

- districts,
- roundabout descriptions.

Box 3.2: Example of *edge* and *lane* XML assignments in a *.net.xml network file

```
<edge id="<ID>" from="<FROM_NODE_ID>" to="<TO_NODE_ID>" priority="<PRIORITY>">
    <lane id="<ID>_0" index="0" speed="<SPEED>" length="<LENGTH>"
          shape="0.00,495.05 248.50,495.05"/>
    <lane id="<ID>_1" index="1" speed="<SPEED>" length="<LENGTH>"
          shape="0.00,498.35,2.00 248.50,498.35,3.00"/>
</edge>
```

Box 3.3: Example of *junction* definition and their *requests* in a *.net.xml network file

```
<junction id="<ID>" type="<JUNCTION_TYPE>" x="<X-POSITION>" y="<Y-POSITION>"
          incLanes="<INCOMING_LANES>" intLanes="<INTERNAL_LANES>"
          shape="<SHAPE>">
      <request index="<INDEX>" response="<RELATIVE_MAJOR_LINKS>"
               foes="<FOE_LINKS>" cont="<MAY_ENTER>"/>
      ... further requests ...
</junction>
```

Box 3.4: Example of *traffic lights* and their *phases* XML assignments in a *.net.xml network file

```
<tlLogic id="<ID>" type="<ALGORITHM_ID>" programID="<PROGRAM_ID>"
         offset="<TIME_OFFSET>">
    <phase duration="<DURATION#1>" state="<STATE#1>"/>
    <phase duration="<DURATION#1>" state="<STATE#1>"/>
    ... further states ...
    <phase duration="<DURATION#n>" state="<STATE#n>"/>
</tlLogic>
```

Box 3.5: Example of *connections* in a *.net.xml network file

```
<connection from="<FROM_EDGE_ID>" to="<TO_EDGE_ID>"
            fromLane="<FROM_LANE_INDEX>" toLane="<TO_LANE_INDEX>"
            via="<VIA_LANE_ID>" tl="<FROM_EDGE_ID>"
            linkIndex="12" dir="r" state="o"/>
```

Box 3.6: Example of a *roundabout* definition in a *.net.xml network file

```
<roundabout nodes="nodeID1 nodeID2 ..." edges="edgeID1 edgeID2 ..."/>
```

Although being readable (XML) by human beings, SUMO network files are not meant to be edited by hand. Rather, because there are many inter-dependencies among edges, junctions and traffic lights which might prevent a proper simulation, there are many programs available within the SUMO package which are able to generate these files, performing a consistency checking to avoid defective networks. In the following subsections, we present some different techniques for creating these files.

3.2.1 Creating Abstract Topologies

Geometrically simple, abstract road maps can be created with the NETGENERATE software. NETGENERATE is a command line application where you specify as input a set of command line parameters, receiving as an output a generated SUMO-road network. A description of all possible configurations with NETGENERATE can be found at:

http://sumo.dlr.de/wiki/NETGENERATE

Among the possible kinds of networks generated by NETGENERATE are Grid Networks, Spider Networks and Random Networks. Many kinds of Traffic Lights Signals specifications are possible.

3.2.2 Constructing Topologies by Hand

If NETGENERATE is not enough for the kind of network you want to build, you might generate your network by hand using XML description files together with the NETCONVERT software. It is important to notice that these XML files are not the same XML files used by the SUMO simulator. In fact, there are two different representations:

- The .net.xml file which is loaded into the simulation
- A set of plain-xml files which describe the network topology and geometry

The .net.xml file contains lots of generated information such as structures within an intersection and right-of-way logic, which are important for the consistency of the simulation. These are generated automatically by NETCONVERT. The plain-xml files are of different kinds, providing different information:

- node files: .nod.xml
- edge files: .edg.xml
- type files: .typ.xml
- connection files: .con.xml
- traffic lights logic files: .tll.xml

Details on the required XML structures for building these files can be found at:

http://sumo.dlr.de/wiki/Networks/Building_Networks_from_own_XML-descriptions

NETCONVERT can convert freely and without information loss between .net.xml files and plain-XML files. Only the plain-XML format is meant to be edited by users. In contrast, the .net.xml format has lots of subtle inter-dependencies between it's elements and should never be edited by hand.

It is possible to load a net.xml file along with plain-xml patch files into NETCONVERT to modify some aspects of an existing network.

3.2.3 Using a Graphic Tool to Create a Topology

You may also use the software NETEDIT for building your own road networks or for reworking those obtained from NETCONVERT or NETGENERATE. NETEDIT is a visual network editor. It can be used to create networks from scratch and to modify all aspects of existing networks. With a powerful selection and highlighting interface it can also be used to debug network attributes. NETEDIT is built on top of NETCONVERT. As a general rule of thumb, anything NETCONVERT can do, NETEDIT can do as well. NETEDIT has unlimited undo/redo capabilities and thus allows editing mistakes to be quickly corrected.

More information on NETEDIT can be found at:

http://sumo.dlr.de/wiki/NETEDIT

3.2.4 Importing a Topology from other file formats

The same software NETCONVERT allows to import road networks from different file formats. The following formats are actually supported:

- OpenStreetMap databases
- PTV VISUM (a macroscopic traffic simulation package)
- PTV VISSIM (a microscopic traffic simulation package)
- OpenDRIVE networks
- MATsim networks
- SUMO networks
- ArcView-data base files
- Elmar Brockfelds unsplitted and splitted NavTeq-data
- RoboCup Rescue League folders

Among these, one of the most popular choices is importing a real map from the OpenStreetMap database. Let's see a little bit more in detail how it works, for the sake of illustration.

Acquiring a Map from OpenStreetMaps

There are many ways in which a map can be obtained from OpenStreetMaps. The simplest way of doing it is using a Web browser and accessing the web page at:

www.openstreetmap.org

From this web page, you might be able to select a portion of the map you wish do download, and export it. An example of this solution can be seen on figure 3.1.

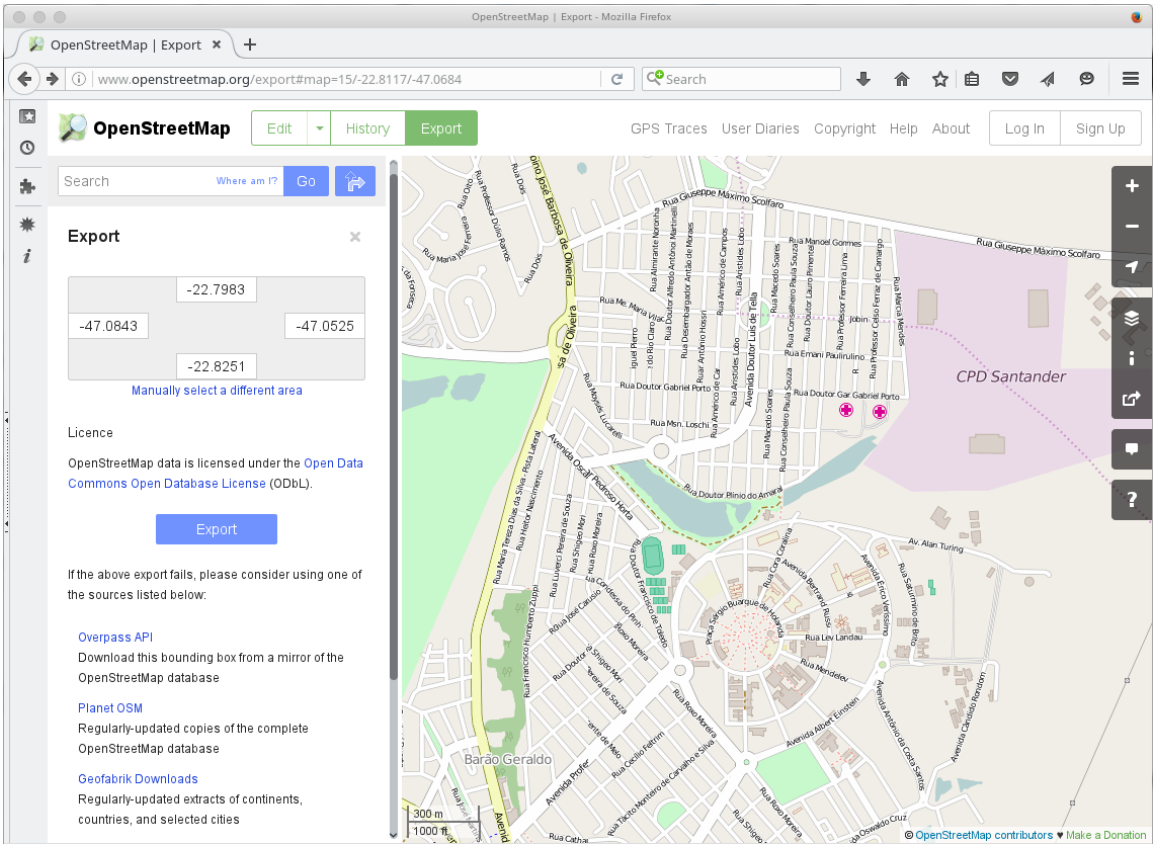


Figure 3.1: Browser interface for the OpenStreeMap web page

A second possibility is using some sort of standalone application enabled to use the OpenStreetMaps database. There are many available programs like this at the Internet. One of the most complete ones is JOSM, the Java OpenStreetMap Editor, which can be downloaded for free at:

https://josm.openstreetmap.de/

An example of the JOSM graphical user interface can be shown in figure 3.2. JOSM allows a better configuration of which details from map are to be downloaded. It is important to understand that the OSM files contains much more information than the information

necessary for performing a SUMO simulation. Depending on the size of the topology, the OSM maps can be really huge, and take some time to be downloaded. The use of JOSM can make the obtained file size to be reduced.

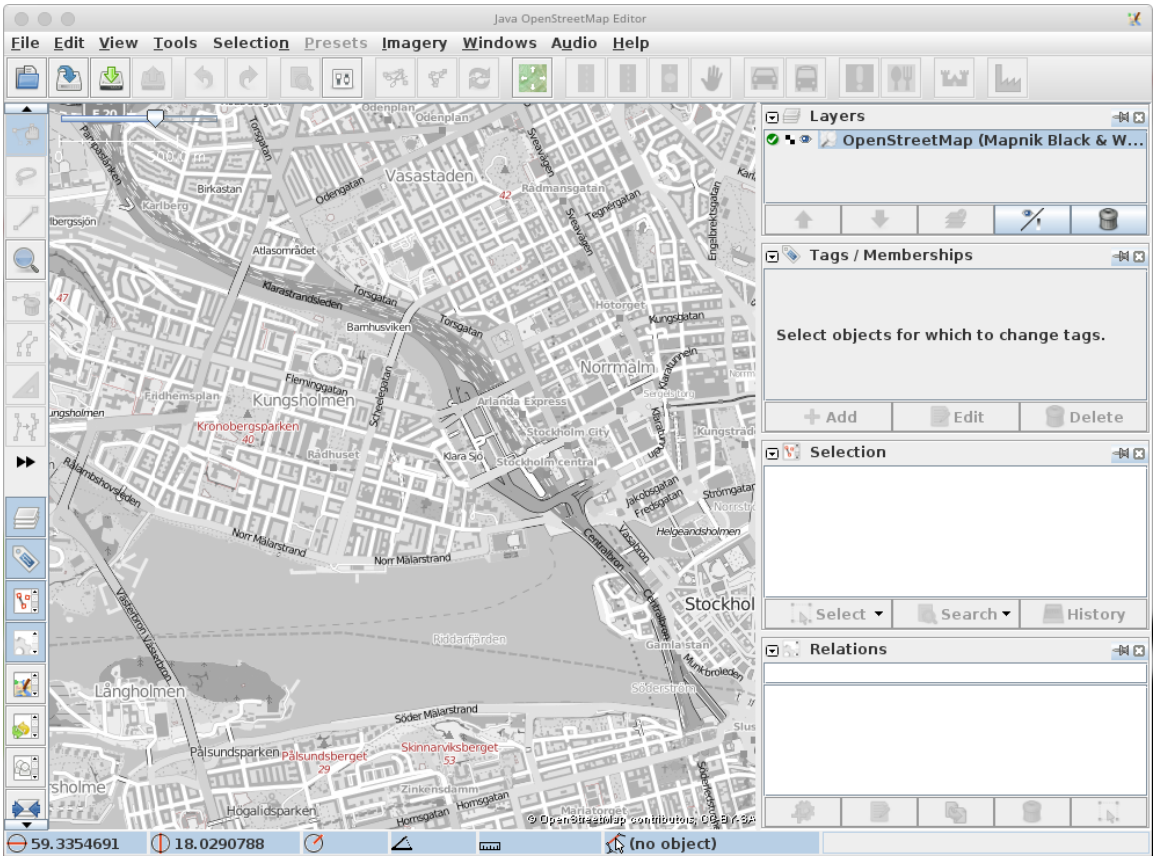


Figure 3.2: Graphical User Interface of the software JOSM

If you already have the coordinates of the bounding box surrounding the desired area (which can be obtained e.g. from the `www.openstreetmaps.org` website, you may use a web service with the following URL:

```
http://api.openstreetmap.org/api/0.6/map?bbox=<coordinates>
```

where `<coordinates> = <SW-longitude,SW-latitude,NE-longitude,NE-latitude>` as e.g. in

```
http://api.openstreetmap.org/api/0.6/map?bbox=13.278,52.473,13.471,52.552
```

Yet another possible way is through the use of a Python script which is called `osmWebWizard.py`. This script will open a browser, and present the interface in figure 3.3. You can select from the map the desired map, and click the *Generate Scenario* button. This Python script is usually together with the SUMO tools. In a Ubuntu Linux distribution, it might be found at `/usr/share/sumo/tools`

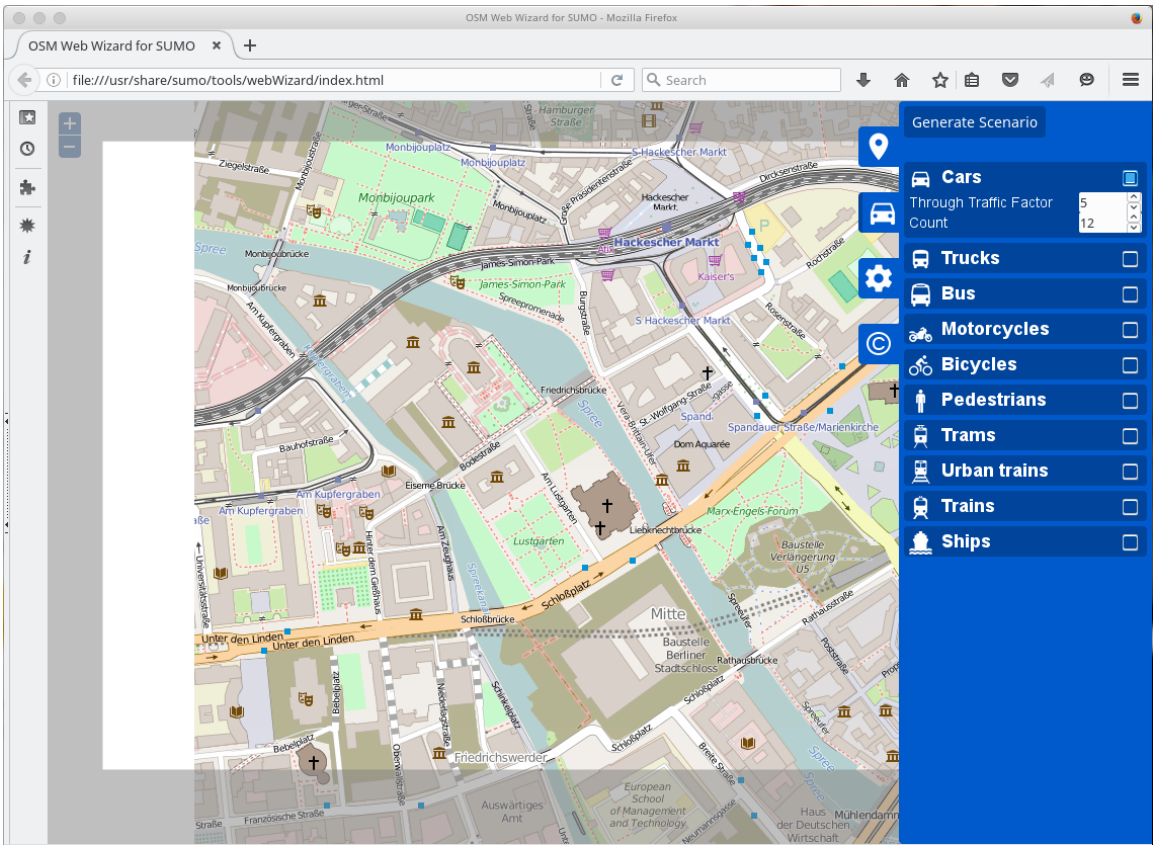


Figure 3.3: Browser interface for the *osmWebWizard.py* script

More information on getting an osm file can be obtained from the SUMO documentation at:

http://sumo.dlr.de/wiki/Networks/Import/OpenStreetMapDownload

Converting from an OSM file to a SUMO .net.xml file

With all these different options, the result will be an XML file, usually named `<something>.osm` or `<something>.osm.xml`. You can then use the `NETCONVERT` tool to convert the OSM file to the .net.xml file format, in order to use it for the SUMO simulation.

The simplest way of doing it is just a simple command:

```
netconvert --osm <name>.osm -o <name>.net.xml
```

For a realistic simulation, though, this might be not enough. Sometimes, information such as speed limit is missing in the OSM file and must be inferred from the abstract type of the road (i.e. motorway). Different simulation scenarios require different modes of traffic and thus different parts of the traffic infrastructure to be imported. Usually, conversion options must be specified, as e.g.:

```
    --geometry.remove --roundabouts.guess --ramps.guess
    --junctions.join --tls.guess-signals --tls.discard-simple
    --tls.join
```

Besides that, there is also a lot of information in OSM files, like POI (Points of Interest), buildings drawings, regions, etc, which are usually simply discarded while creating the SUMO .net.xml file. It is possible, though, to use type files to convert this information to useful information within SUMO. For example, all this graphic information not important for simulation can be showed in `sumo-gui`. You may use the `POLYCONVERT` tool to import polygons from OSM-data and produce a Sumo-polygon file. An example of a type file can be seen in Box 3.7.

Box 3.7: Example of *osmPolyconvert.typ.xml* file for OpenStreetMap type conversion

```
<polygonTypes>
  <polygonType id="waterway"               name="water"       color=".71,.82,.82" layer="-4"/>
  <polygonType id="natural"                name="natural"     color=".55,.77,.42" layer="-4"/>
  <polygonType id="natural.water"          name="water"       color=".71,.82,.82" layer="-4"/>
  <polygonType id="natural.wetland"        name="water"       color=".71,.82,.82" layer="-4"/>
  <polygonType id="natural.wood"           name="forest"      color=".55,.77,.42" layer="-4"/>
  <polygonType id="natural.land"           name="land"        color=".98,.87,.46" layer="-4"/>

  <polygonType id="landuse"                name="landuse"     color=".76,.76,.51" layer="-3"/>
  <polygonType id="landuse.forest"         name="forest"      color=".55,.77,.42" layer="-3"/>
  <polygonType id="landuse.park"           name="park"        color=".81,.96,.79" layer="-3"/>
  <polygonType id="landuse.residential"    name="residential" color=".92,.92,.89" layer="-3"/>
  <polygonType id="landuse.commercial"     name="commercial"  color=".82,.82,.80" layer="-3"/>
  <polygonType id="landuse.industrial"     name="industrial"  color=".82,.82,.80" layer="-3"/>
  <polygonType id="landuse.military"       name="military"    color=".60,.60,.36" layer="-3"/>
  <polygonType id="landuse.farm"           name="farm"        color=".95,.95,.80" layer="-3"/>
  <polygonType id="landuse.greenfield"     name="farm"        color=".95,.95,.80" layer="-3"/>
  <polygonType id="landuse.village_green"  name="farm"        color=".95,.95,.80" layer="-3"/>

  <polygonType id="tourism"                name="tourism"     color=".81,.96,.79" layer="-2"/>
  <polygonType id="military"               name="military"    color=".60,.60,.36" layer="-2"/>
  <polygonType id="sport"                  name="sport"       color=".31,.90,.49" layer="-2"/>
  <polygonType id="leisure"                name="leisure"     color=".81,.96,.79" layer="-2"/>
  <polygonType id="leisure.park"           name="tourism"     color=".81,.96,.79" layer="-2"/>
  <polygonType id="aeroway"                name="aeroway"     color=".50,.50,.50" layer="-2"/>
  <polygonType id="aerialway"              name="aerialway"   color=".20,.20,.20" layer="-2"/>

  <polygonType id="shop"                   name="shop"        color=".93,.78,1.0" layer="-1"/>
  <polygonType id="historic"               name="historic"    color=".50,1.0,.50" layer="-1"/>
  <polygonType id="man_made"               name="building"    color="1.0,.90,.90" layer="-1"/>
  <polygonType id="building"               name="building"    color="1.0,.90,.90" layer="-1"/>
  <polygonType id="amenity"                name="amenity"     color=".93,.78,.78" layer="-1"/>
  <polygonType id="amenity.parking"        name="parking"     color=".72,.72,.70" layer="-1"/>
  <polygonType id="power"                  name="power"       color=".10,.10,.30" layer="-1"
               discard="true"/>
  <polygonType id="highway"                name="highway"     color=".10,.10,.10" layer="-1"
               discard="true"/>

  <polygonType id="boundary" name="boundary"    color="1.0,.33,.33" layer="0" fill="false"
               discard="true"/>
  <polygonType id="admin_level" name="admin_level"    color="1.0,.33,.33" layer="0" fill="false"
               discard="true"/>
</polygonTypes>
```

Using a type file like the one in box 3.7, you might generate a polygon file using:

```
polyconvert --net-file <name>.net.xml --osm-files <name>.osm
            --type-file typemap.xml -o <name>.poly.xml
```

The generated polygon file can now be used in a configuration file for `sumo-gui`, like the one in box 3.8.

Box 3.8: Example of an *example.conf.xml* file with extra polygon information

```
<configuration>
    <input>
        <net-file value="example.net.xml"/>
        <additional-files value="example.poly.xml"/>
    </input>
</configuration>
```

More information on transforming an `osm` file into a SUMO `.net.xml` file can be obtained from the SUMO documentation at:

http://sumo.dlr.de/wiki/Networks/Import/OpenStreetMap

3.3 Traffic Pattern Demand

After having generated a network, one could take a look at it using `SUMO-GUI`, but no cars would be driving around. One still needs some kind of description about the vehicles and how they flow in the network topology. This is called the traffic demand. In order to understand the traffic demand, the following terminology is necessary.

A **trip** is a vehicle (or people) movement from one place to another defined by the starting edge (street), the destination edge, and the departure time. A **route** is an expanded trip, which means that a route definition contains not only the first and the last edge, but all edges the vehicle will pass through, from its origin up to its final destination. SUMO and SUMO-GUI need *routes* as input for vehicle movements. This means that a route for each vehicle, since the vehicle enters into a street (which can happen anywhere in the topology) until its final destination (which again, can be anywhere in the topology), passing through all middle streets, need to be provided in order for the simulation to run. There are several tools to generate routes for SUMO.

By now, `SUMO` contains four applications for generating routes. `DUAROUTER` is responsible for importing routes or their definitions from other simulation packages and for computing routes using the shortest-path algorithm by Dijkstra. Additionally, in combination with the simulation, the `DUAROUTER` can compute the dynamic user assignment formulated by C. Gawron. Another option, `JTRROUTER`, may be used if you want to model traffic

statistically, using flows and turning percentages at junctions. A third option, OD2TRIPS helps you to convert OD-matrices (origin/destination-matrices) into trips. The final option, DFROUTER, computes routes from a set of measurements collected from selected observation points. An option that does not generate the route file in itself, but can be used together with DUAROUTER, is the program ACTIVITYGEN, which can be used to turn population statistics into a trip file, and then using DUAROUTER generating a traffic demand.

3.3.1 Specifying Random Traffic

The easiest way of generating traffic for a SUMO simulation is using a random generated traffic. The user might use the Python script randomTrips.py (available at the tools directory in SUMO) in order to generate a set of random trips for a given network, which will choose a source and a destination edge either uniformly at random or with a modified distribution. The resulting trips are stored in an XML file suitable for DUAROUTER which can be called automatically by the script with the right options enabled. The trips are distributed evenly in an interval defined by configurable begin and end times. The number of trips is defined by a configurable repetition rate. Every trip has an id consisting of a prefix and a running number. For example the following command:

```
randomTrips.py -n name.net.xml -e 5000 -r name.rou.xml
```

will generate a route file for the topology network indicated by name.net.xml, with the name name.rou.xml with trips starting since the time of 0 seconds up to 5000 seconds.

There are many parameter which might be tuned in order to configure different probabilities distributions and other features. More information on creating random traffic can be found at:

http://sumo.dlr.de/wiki/Tools/Trip

3.3.2 Defining Traffic by Hand

It is also possible to define the demand file manually or to edit generated files with a text editor. Before starting, it is important to know that a vehicle in SUMO consists of three parts:

- a vehicle type which describes the vehicle's physical properties,
- a route the vehicle shall take,
- and the vehicle itself.

Both routes and vehicle types can be shared by several vehicles. It is not mandatory to define a vehicle type. If not given, a default type is used. The driver of a vehicle does not have to be modeled explicitly. For the simulation of persons which walk around or ride in vehicles, additional definitions are necessary.

More information on the details on the XML required for generating a route file by hand can be found at:

http://sumo.dlr.de/wiki/Definition_of_Vehicles,_Vehicle_Types,_and_Routes

3.3.3 Using Trips to Generate Routes

An easier method than defining the route files by hand is defining just a set of trips, and using the application DUAROUTER to turn your trips into routes. In some sense, this is what the script randomTrips.py does while generating random traffic. Each trip consists of, at least, the starting and the ending edge and the departure time. The set of trips should be encoded into a trip XML file. You might create this file by hand or write your own script to import custom data.

The primary output of DUAROUTER is a .rou.xml file. Additionally a .rou.alt.xml with the same name prefix as the .rou.xml file will be generated. This route alternative file holds a routeDistribution XML tag for every vehicle. This tag is used during dynamic user assignment (DUA) but can also be loaded directly into SUMO.

There are three different routing algorithms used by DUAROUTER to generate the routes: the Dijkstra's algorithm, the A* (A-star) algorithm and the Contraction Hierarchies algorithm. All available routing algorithms return the path with the shortest travel time by default.

More information on building a trip file can be found at:

http://sumo.dlr.de/wiki/Demand/Shortest_or_Optimal_Path_Routing

A complete description of the many configuration options for DUAROUTER can be found at:

http://sumo.dlr.de/wiki/DUAROUTER

Dynamic User Assignment

There are many possible strategies for generating routes from trips. One desirable strategy is to find a route for each vehicle (each trip from the trip-file) such that each vehicle cannot reduce its travel cost (usually the travel time) by using a different route. This strategy is called *Dynamic User Assignment*, and it does so iteratively by calling DUAROUTER to route the vehicles in a network with the last known edge costs (starting with empty-network travel times) and further calling SUMO to simulate "real" travel times result from the calculated routes. The result edge costs are used in the net routing step.

The number of iterations may be set to a fixed number or determined dynamically depending on the user options. In order to ensure convergence there are different methods employed to calculate the route choice probability from the route cost (so the vehicle does not always choose the "cheapest" route). In general, new routes will be added by the router to the route set of each vehicle in each iteration (at least if none of the present routes is the "cheapest") and may be chosen according to the route choice mechanisms. The two methods which are implemented are the *Gawron* and *Logit* methods.

The tool <SUMO_HOME>/tools/assign/duaIterate.py can be used to compute the (approximate) dynamic user equilibrium. It is important to notice that this script requires copious amounts of disk space. An example of using the duaIterate.py script is the following:

```
python dualterate.py -n <network-file> -t <trip-file> -l <nr-of-iterations>
```

3.3.4 Using Observation Points to Generate Routes

SUMO contains a routing module named DFROUTER. The idea behind this router is that nowadays, most highways are well equipped with induction loops, measuring each of the highways' entering and leaving flows. Given this information one may assume that the flows on the highway are completely known. DFROUTER uses directly the information collected from induction loops to rebuild the vehicle amounts and routes. This is done in several steps, being mainly:

- Computing (and optionally saving) the detector types in the means that each induction is set to be a source detector, a sink detector or an in-between detector
- Computing (and optionally saving) the routes between the detectors
- Computing the flow amounts between the detectors
- Saving the flow amounts and further control structures

The idea behind the DFROUTER assumes that a network is completely covered by detectors, meaning that all off- and on-ramps have an induction loop placed on them. Such an information whether an induction loop is a pure source or sink or whether it is placed between such is but not given initially. It must be computed. To do this, the DFROUTER needs the underlying network as well as a list of detector definitions where each describes the position of an induction loop. The network, being a previously built SUMO-network, is supplied to the DFROUTER.

Given a network and the list of detectors, DFROUTER assigns types to detectors and saves the so extended list into a file. You can also generate a list of points of interests (POIs) which can be read by SUMO-GUI where each POI represents a detector and is colored by the detector type: green for source detectors, red for sink detectors, blue for in-between detectors, and black for discarded detectors.

Now that we do know where vehicles enter and where they leave the network, we may compute routes for each of the pairs. The generated file only contains routes, no vehicle type definitions and no vehicles.

Normally, only routes starting at source detectors and ending at sink detectors are computed.

More information on setting up routes using observation points can be found at:

http://sumo.dlr.de/wiki/Demand/Routes_from_Observation_Points

More information on DFROUTER can be found at:

http://sumo.dlr.de/wiki/DFROUTER

3.3.5 Using Flow Sources and Turn Probabilities to Generate Routes

The JTRROUTER is a routing applications which uses flows and turning percentages at junctions as input. The following parameter must be supplied:

- the network to route the vehicles through,
- the description of the turning ratios for the junctions (defaults may be used for this, too), and
- the descriptions of the flows.

A call may look like this:

```
jtrrouter --flow-files=<FLOW_DEFS> --turn-ratio-files=<TURN_DEFINITIONS>
          --net-file=<SUMO_NET> --output-file=MySUMORoutes.rou.xml
          --begin <UINT> --end <UINT>
```

To describe the turn definitions, one has to write an XML file. Within this file, for each interval and each edge the list of percentages to use a certain follower edge has to be given.

More information on setting up routes using flow sources and turn probabilities can be found at:

http://sumo.dlr.de/wiki/Demand/Routing_by_Turn_Probabilities

More information on JTRROUTER can be found at:

http://sumo.dlr.de/wiki/JTRROUTER

3.3.6 Using Origin/Destin (OD) Matrices to Generate Routes

Origin/Destin (OM) Matrices are a format used by different software tools developed by PTV Group (Visum, Vissim, etc), a company located in Karlsruhe in Germany, since 1979[1]. Visum is a software for traffic analyses, forecasts and GIS-based data management. VISSIM is a microscopic traffic simulator just like SUMO. PTV Group solutions are used in many countries to organize optimal traffic flow in real cities. OD Matrices are sometimes available from traffic authorities from different countries.

OD matrices encode the amounts of vehicles driving from one district or traffic assignment zone (TAZ) to another within a certain time period. VISUM format is one of the possible input files which can be converted to SUMO, in order to generate a .net.xml topology file. If NETCONVERT is used in a VISUM file map, the information on districts stored in the VISUM file are parsed and stored within the generated SUMO network file. For topology maps obtained from a different source, a TAZ file need to be supplied in order to partitionate the map into traffic assignment zones.

The software OD2TRIPS computes trip tables from O/D (origin/destination) matrices, using the "V"- and the "O"-formats from Visum, or the Amitran format. OD2TRIPS reads all matrices and generates trip definitions, according to the traffic among zones specified in the OD matrix.

More information on setting up routes using OD Matrices can be found at:

http://sumo.dlr.de/wiki/Demand/Importing_O/D_Matrices

[1] http://www.ptvgroup.com

More information on OD2TRIPS can be found at:

http://sumo.dlr.de/wiki/OD2TRIPS

3.3.7 Activity-based Demand Generation

ActivityGen generates demand from a description of the population in a city, using a simple activity-based traffic model. A *Statistics file* describes many aspects from the city population, which might be useful for the generation of traffic. The following information can be used:

- **General Information:** The number of inhabitants, number of households, the children age limit for starting driving, the retirement age limit where people are allowed to drive, a car rate among inhabitants, an unemployment rate among inhabitants, the foot distance limit, after which inhabitants will require a transportation way, the incoming traffic and the outgoing traffic.
- **General parameters:** The car preference rate, which defines the rate of inhabitants preferring using cars instead of other transportation ways, the mean time per Km in the city, the free time activity rate, a uniform random traffic to be generated, and the departure variation, in time.
- **Population distribution by age:** described as the number of people among many different age layers.
- **Working hours:** The proportion of inhabitants opening and closing work at different times.
- **Population distributin by streets:** For different streets, the number of people per meter street (relative value, normalized by the total number of inhabitants), and the number of work positions per meter street (relative value, normalized by the total number of city's work demand).
- **City gates:** position of incoming and outgoing positions specifying the relative rate of each gate as an incoming position and also the proportion of the outgoing vehicles, leaving the city through the output gate.
- **Schools:** for each school within the map, the exact position of schools in the street, age of the youngest pupils of the school, age of children not accepted in the school any more, maximum number of pupils accepted, time of class beginning (school time), time of class ending (home time).
- **Bus Line:** For each bus line, the maximum time needed for a bus to do the end-to-end trip, the reference id of the chosen station, time of the beginning of a new frequency, end time of the frequency, time between two buses.

The population is distributed according to the statistics into households located in streets. People are likely to use different means of transportation in relation to their location, the availability of the different means and their destination. Three kind of Means are used in ActivityGen: feet or bike, buses or cars. All of them have their own possibilities and availability characteristics. Trips by foot are available only for very short distances. But in this case, the person is very likely to go by foot. The bike isn't really used (buses and cars are preferred), it is supposed to serve in cases where none of the three means (Feet, buses and cars) are not available. Bus stations are located in the city corresponding to the real

bus line network given in input (statistical data over the city). Someone enough close to a bus station and whose destination is enough close to another bus station is eligible for the public transportation means. Householders having one or more cars can drive (not children) or be accompanied (escorted) by another adult who needs a car too. Children can only be accompanied (to school for example). Some households have no car, in this case they have to live enough close to a bus station. In the case of having a destination far from any bus station, they can go by foot or ride a bike. But this case doesn't generated any motorized vehicle traffic.

More information on setting up routes using activities can be found at:

http://sumo.dlr.de/wiki/Demand/Activity-based_Demand_Generation

More information on ACTIVITYGEN can be found at:

http://sumo.dlr.de/wiki/ACTIVITYGEN

3.4 Simulation Results

Using a `.net.xml` and a `.route.xml` files it is possible to run a complete simulation in sumo, either with the `sumo` or `sumo-gui` commands. In the case of `sumo-gui`, the user might be able to see the vehicles flowing in the network and zoom on specific parts of the topology for a more detailed view of the vehicles flow. Nevertheless, after the end of the simulation nothing else is generated. In fact, SUMO allows the generation of a large number of different measures. Per default, though, all of them are disabled, and have to be triggered individually. Some of the available outputs (raw vehicle positions dump, trip information, vehicle routes information, and simulation state statistics) are triggered using command line options. The others have to be defined within "additional files" inserted in the simulation configuration.

A detailed description of the kind of information which can be stored regarding the simulation results, can be found at:

http://sumo.dlr.de/wiki/Simulation/Output

A very detailed and precise simulation feedback is possible (data for each vehicle or person, at precise events where they have a change in status). Nevertheless, one of the most interesting kind of feedback is possible with the `--duration-log.statistics` option. With this option turned on, the following averages for all vehicle trips will be made available:

- RouteLength: average route length
- Duration: average trip duration
- WaitingTime: average time spent standing (involuntarily)
- TimeLoss: average time lost due to driving slower than desired
- DepartDelay: average time vehicle departures were delayed due to lack of road space

When setting this option and using SUMO-GUI, the network parameter dialog will also show a running average for these traffic measures (The dialog is accessible by right-clicking on the network background).

3.4.1 Setting up the Configuration File

Even though there are many command-line options which can be given to SUMO, the best way of choosing the desired simulation results and invoking specific processing options is inserting tags in the configuration file. There is a specific tag `<output>` which is used to specify the information you want from the simulation, and also other tags like `<processing>`, which is used to invoke specific processing commands, `<report>`, which specifies the kind of online report and `<gui_only>` for specifying options affecting the GUI. For altering GUI settings, the best way of doing it is pressing `CTRL-V` while using the GUI, performing the changes you wish and saving into a gui.xml file, which can be further used with the `gui-settings-file` option. If the tag `duration-log.statistics` is used, you may get on-line statistics by using the right mouse button of your mouse anywhere on the simulation view and pressing the "Show Parameter" option. Box 3.9 shows an example of how these tags can be used to control processing options and outputs from the simulation.

Box 3.9: Example of a configuration file with additional options configured for obtaining simulation results and using optional command processing.

```
<configuration>
    <input>
        <net-file value="example.net.xml"/>
        <route-files value="example.rou.xml"/>
        <additional-files value="example.add.xml"/>
    </input>

    <output>
        <summary-output" value="results.out.xml"/>
    </output>

    <processing>
       <ignore-route-errors value="true"/>
    </processing>

    <report>
       <verbose value="true"/>
       <duration-log.statistics value="true"/>
   </report>

   <gui_only>
       <gui-settings-file value="example.gui.xml"/>
       <start value="true"/>
       <game value="true"/>
   </gui_only>
</configuration>
```

While preparing a simulation for human view, an important parameter to be saved in a GUI xml file is the *delay*. If you use the default delay (0 - zero), the simulation will be too quick to be viewed. A delay of 1000 will bring you closer to real time. A delay of 200 will give you a fast but viewable simulation.

The available options for configuration can be grabbed from the `sumoConfiguration.xsd` file which can be obtained from:

http://sumo.dlr.de/xsd/sumoConfiguration.xsd

There is also a copy of this file with all the other `xsd` definitions at the `data/xsd` directory of your sumo instalation (`${SUMO_HOME}/data/xsd`).

Additional information on the meaning of each parameter can be obtained here:

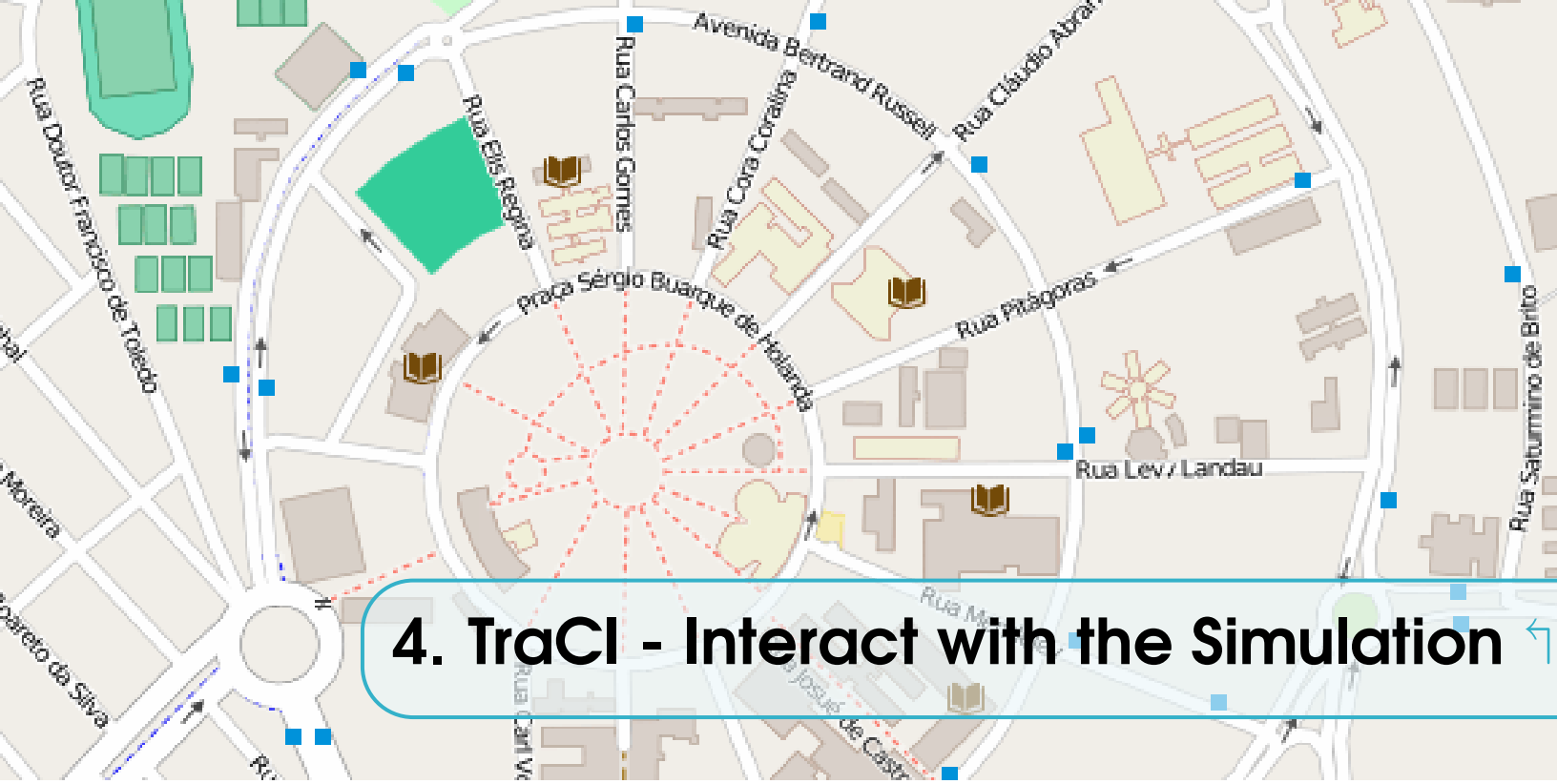
http://sumo.dlr.de/wiki/SUMO

If you want the most recent options (depending on the version of your sumo installation), you may try the following commands:

```
man sumo-gui
or
sumo --help
```

3.4.2 SUMO Gaming Mode

There is a special *Gaming Mode* which can be invoked for SUMO-GUI, which hides all the controls in the simulator view, and presents some statistics in the screen. To enter in Gaming Mode, you just need to press <CTRL-G> at your keyboard. To go out from Gaming Mode it is just the matter of pressing <CTRL-G> again. During Gaming Mode the mouse becomes not responsive, but you can move the view using the arrow keys from the keyboard, together with the + and - keys for the zoom focusing. It is also possible to program the simulation to enter automatically on gaming mode by using the <game> tag at the <gui_only> session of the configuration file, as is illustrated in box 3.9.



4. TraCI - Interact with the Simulation

4.1 Introduction

In chapter 3, we learned how to build up a topology, with a map containing roads and pathways for the simulation, and how to generate a traffic, with different modalities of available information. But all the control in this simulation will assume fixed programs for the traffic lights controlling the junctions. The behavior of these traffic lights can be defined automatically by SUMO (or fine-tuned case by case), but all the lights will run with fixed controllers. SUMO allows external programs to interact with the simulation, being able to grab information from each involved object in the simulation (vehicles, people, traffic lights, inductive loops, etc). The interaction with external programs is made available using *TraCI*, the "Traffic Control Interface". Giving access to a running road traffic simulation, it allows external programs to retrieve many parameters and variables from simulated objects and to control their behavior "on-line" in a SUMO instance.

TraCI uses a TCP based client/server architecture to provide access to SUMO. This is not, though, the default operation mode. If properly configured (defining a remote port), SUMO acts as a server that is started with a few additional command-line options, as the example shown in box 4.1.

When started with the –remote-port <INT> option, SUMO only prepares the simulation and waits for an external application, that takes over the control. Please note, that the –end

Box 4.1: Example of an additional command-line option, where <INT> is the port SUMO will listen on for incoming connections.

```
$ sumo -n network.net.xml -r routes.rou.xml --remote-port <INT>
```

<TIME> option is ignored when SUMO runs as a TraCI server. In this case, SUMO will run until the client demands the end of the simulation.

A detailed description of the TCP operations available to interact with SUMO during the simulation can be found at:

http://www.sumo.dlr.de/userdoc/TraCI.html

The SUMO community has developed TraCI wrapper interfaces in many different programming languages, such as Python, Java and C++. In the next section, we will give some interaction examples using the Java wrapper, known as "TraCI4J".

4.2 TraCI4J - A Java Package implementing the TraCI Protocol

TraCI4J is a Java library implementing the TraCI protocol, and allowing the user to interact with the simulation running in SUMO in many ways. It was developed at ApPeAL (Applied Pervasive Architectures Lab), Politecnico di Torino, by Enrico Gueli. A summary of some TraCI4J classes can be seen in figure 4.1.

The simulation can be started, stopped and advanced step by step. While the simulation is running, many pieces of information can be retrieved, both static (e.g. the road network topology) and dynamic (e.g. position and speed of vehicles). A set of TraCI4J classes match the corresponding TraCI objects, each with methods allowing for value reading and state changing.

The library is currently in alpha development stage. This means that not all of the TraCI features are available as TraCI4J classes. Those which are available, though, work

Package it.polito.appeal.traci

Class Summary

Class	Description
Edge	Representation of an edge in the SUMO environment.
InductionLoop	Representation of an induction loop in the SUMO environment.
LaArDetector	Representation of a Lane Area detector (E2) in the SUMO environment.
Lane	Representation of a lane in the SUMO environment.
MeMeDetector	Representation of a Multi-Entry Multi-Exit detector (E3) in the SUMO environment.
POI	Represents a POI (Point-of-Interest) in the SUMO environment.
Route	Representation of a route in the SUMO environment.
TrafficLight	Representation of a traffic light in the SUMO environment.
Vehicle	Representation of a vehicle in the SUMO environment.
VehicleType	Representation of a class of vehicles in the SUMO environment.

Figure 4.1: **TraCI4J class summary.**

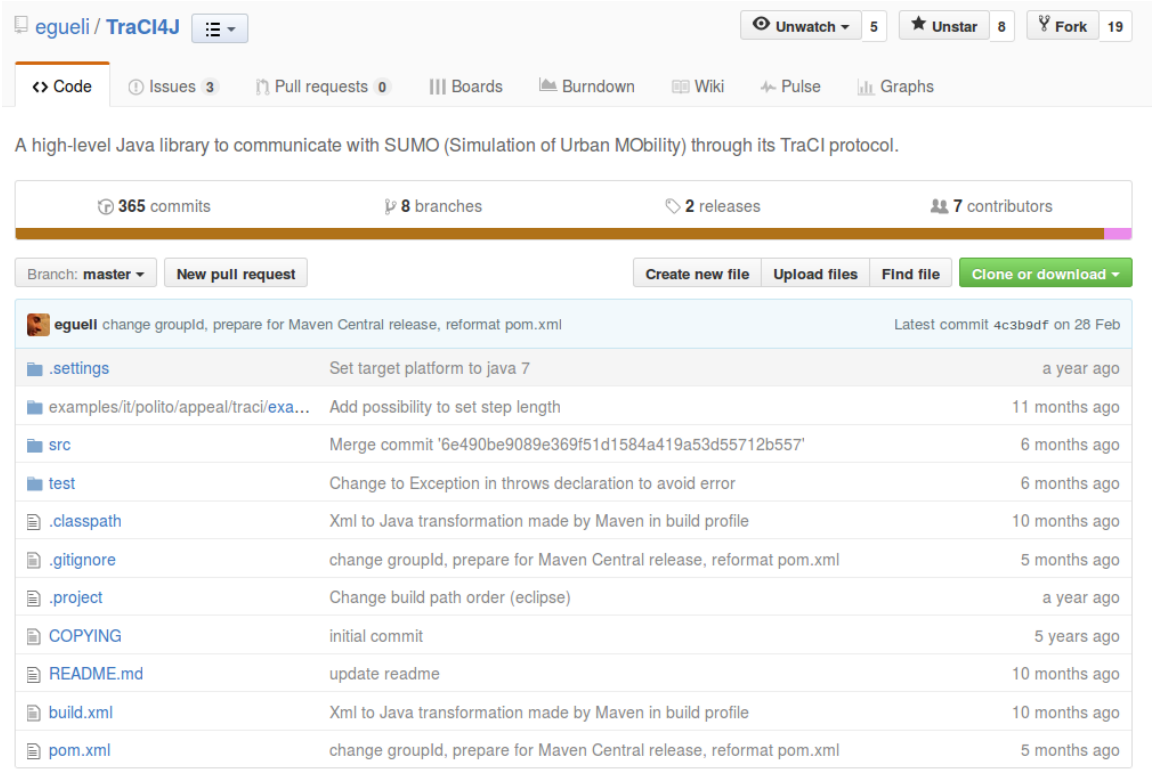


Figure 4.2: **TraCI4J GitHub Page**

as expected. Some others still need to be written. Nevertheless, TraCI4J is available as open-source Java code, and so any function not yet available can be included without too much effort. You can download TraCI4J with the tools provided via GitHub at:

`https://github.com/egueli/TraCI4J`

Figure 4.2 shows the root directory of TraCI4J as it appear on GitHub.

There are no binaries: the library is available as source code in an Eclipse project. Import it in your workspace and Eclipse will try to automatically build it. The library can also be built outside Eclipse via Ant. The default target in the build.xml file will try to compile all code to .class files.

4.2.1 Preparing your Simulation for Interaction

If you don't like the SUMO-GUI interface, you can use TraCI4J only with the purpose of having access to SUMO objects, like Junctions, Edges, Lanes, Traffic Lights, Induction Loops or Vehicles. You can use it, for example, to create an alternative view to SUMO-GUI, using the data coming from the SUMO simulation.

But the interesting possibility brought by TraCI4J (and TraCI as a whole) is to interact in an active way with the simulation, for example controlling cars or traffic lights during a

Box 4.2: How to use it.
First of all, you need:

- A working SUMO installation (0.23.0 or higher)
- Be familiar with SUMO, i.e. be acknowledged with its basic principles, how to set up the input files, how to run it...
- A SUMO file set (a config file, a net description file and a routes file at least)
- A Java SE 1.7 virtual machine
  You can find some usage examples at `https://github.com/egueli/TraCI4J/blob/master/examples/it/polito/appeal/traci/examples`

running simulation. In order to do that, though, you might first prepare your simulation for such an interaction.

Originally, all the requirements for running a simulation are to define the topology and the traffic pattern, as explained in sections 3.2 and 3.3. The Traffic Lights will follow the original programs specified within the `.net.xml` file and nothing else is necessary. But if we want to actively control the traffic lights, first we need some sort of information source for making decisions. And then, we need to have access to the objects we want to control, for example, vehicles or traffic lights. In such a regard, a first problem might comes: what kind of information can be sensed from the simulation ? As a simulation, we have access to all objects within it. But it will be realistic to assume we know all the position and routes of all vehicles in order to program the traffic lights ? With TraCI4J it is possible to gather this information, and if we want to assume that **some** of this information might be available (e.g. by vehicles running some *smartphone* application which provides such an information), this is possible to be simulated. In some cases, we will have to "instrument" our network installing induction loops or other kinds of detectors in order to grab a realistic information from the simulation. SUMO provides three kinds of detectors which can be installed in a network:

- Induction Loop Detectors (E1 Sensors)
- Lane Area Detectors (E2 Sensors)
- Multi-Entry, Multi-Exit Detectors (E3 Sensors)

These are explained in next sections.

4.2.2 Using Inductive Loop Detectors to Grab Information from Environment

SUMO provides a way of defining *Inductive Loop Detectors* to be included in a SUMO simulation. This might use the information collected from these detectors to be reported in an output report and sumo-gui is able to represent such inductive loops in the network topology. An example of such representation is shown in figure 4.3.

In order to include induction loops in the simulation, the definition of all induction loops must be defined in an "additional file" and included in the SUMO config file.

An example is given in Box 4.3 below.

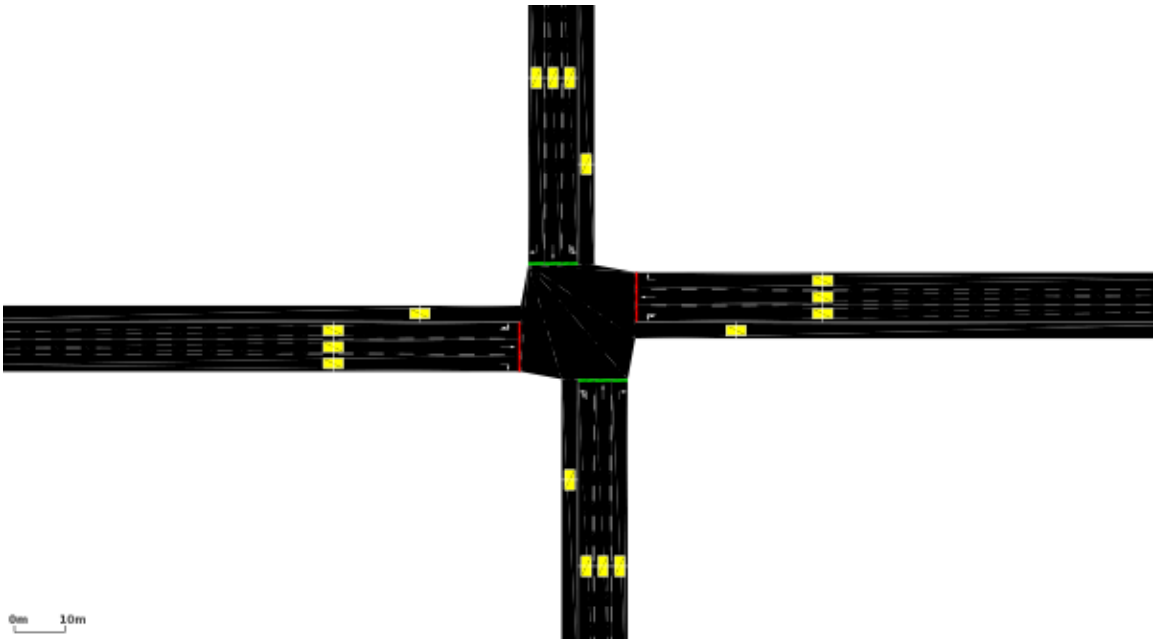


Figure 4.3: **Example of an Induction Loop Representation in SUMO-GUI**

Box 4.3: Example of inductive loops definitions. The canonical format for defining an inductive loop is given by:

```
<additional>
   <inductionLoop id="<ID>" lane="<LANE_ID>" pos="<POSITION_ON_LANE>"
          freq="<AGGREGATION_TIME>" file="<OUTPUT_FILE>"
          friendlyPos="true" splitByType="true"\>
</additional>
```

An example, omitting some parameters is the following:

```
<additional>
  <inductionLoop id="myLoop1" lane="foo_0" pos="42" freq="900" file="out.xml"/>
  <inductionLoop id="myLoop2" lane="foo_2" pos="42" freq="900" file="out.xml"/>
  ....
</additional>
```

In the configuration file, a line should be included with the <additional-files> tag reporting the file name with the additional information.

```
<input>
    <net-file value="net.net.xml"/>
    <route-files value="input_routes.rou.xml"/>
    <additional-files value="e1.add.xml"/>
</input>
```

The *id* tag represents a unique name in which the induction loop will be referred. The *lane* tag inform the lane id where the loop is installed. The *pos* tag indicates how far from the beginning of the lane the sensor is installed. The *freq* tag defines the period (in seconds), in which the sensor integrates information in order to compute its provided statistics. The *file* tag indicates the file where the sensor statistics is recorded. In the case you don't need statistics to be recorded, a solution is to point file as being `/dev/null` or an equivalent pseudo-file. The *friendlyPos* is a flag that, if set, disables the error reporting if the detector is placed behind the lane and automatically corrects the position, placed 0.1 meters from the lane's end or at position 0.1. The *splitByType* is also a flag, which splits the reporting information among a per-vehicle type base.

In a large simulation, with dozens of traffic lights, the instrumentation of all of them with the installation of inductive loops can be a burden, specially if you imported the network from an OSM map, and induction loops are not specified in these maps. Fortunately, there is a SUMO tool called *generateTLSE1Detectors.py* which can detect all the traffic lights in a `.net.xml` file and install an induction loop on each lane reaching the junction where the traffic light is available. For example, in a Linux installation, this script can be found at:

```
/usr/share/sumo/tools/output
```

The information from an induction loop available from TraCI is the following:

- The time step of the beginning of the time period where the last interval of data integration was started
- The time step of the end of the time period where the last interval of data integration was completed
- The id of the detector
- The number of vehicles that have completely passed the detector within the integration interval
- The last information extrapolated to an interval of one hour
- The percentage (0-100%) of the time a vehicle was at the detector
- The mean velocity of all vehicles completely collected during the integration interval.
- The mean length of all completely collected vehicles.
- All vehicles that have touched the detector. Includes vehicles which have not passed the detector completely (and which do not contribute to collected values).

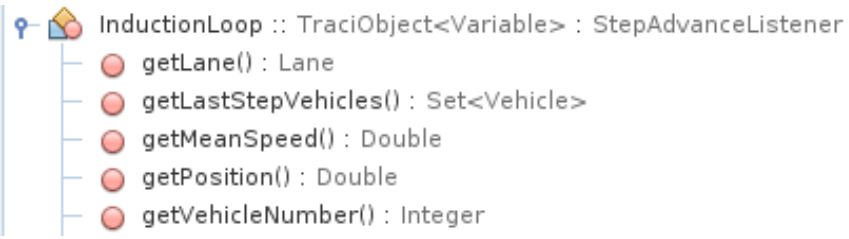


Figure 4.4: **The InductionLoop class in TraCI4J**

Figure 4.4 presents the `InductionLoop` class in TraCI4J, and the available methods. This information can be queried at each simulation step and can be used by the control algorithms in order to decide the Traffic Lights state for the next simulation step.

4.2.3 Lane Area Detectors

Lane Area Detectors are very similar to Inductive Loops (E1-Detectors), but in contrast to them, they cover an area instead of a cross section. They are called E2-Detectors in SUMO. The sensor is tailored for measuring queues of standing/jammed vehicles. However, there is less temporal precision in regard to entering/leaving (E1 has sub-second interpolation). For some purposes it may be necessary to combine both detectors.

Most of the attributes have the same meaning as for induction loops. Because an area detector has a certain length, the parameter "length" must be supplied as a further parameter. It may be a negative number which lets the detector be extended upstream to the given beginning position. The optional parameter "cont" let the detector continue over the current lane onto this lane's predecessors when the detector's length plus its position is larger than the place available on the lane.

In Box 4.4, we show how a Lane Area Detector is represented in an additional file. The procedure for providing them is similar to Inductive Loops.

Box 4.4: Example of a Lane Area Detector representation. The procedure is similar to an inductive loop definition.

```
<additional>
   <laneAreaDetector id="<ID>" lane="<LANE_ID>" pos="<POSITION_ON_LANE>"
         length="<DETECTOR_LENGTH>" freq="<AGGREGATION_TIME>"
         file="<OUTPUT_FILE>" cont="<BOOL>" timeThreshold="<FLOAT>"
         speedThreshold="<FLOAT>" jamThreshold="<FLOAT>" friendlyPos="<BOOL>"/>
</additional>
```

In the same way as for Inductive Loops, there is a script, called *TLSE2Detectors.py*, located in the same place as for its relative, which can automatically generate a lane area

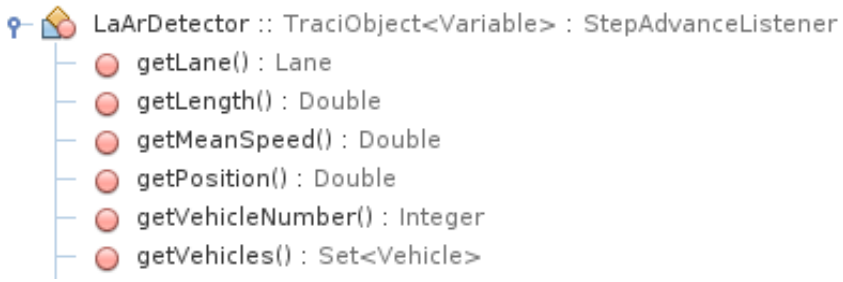


Figure 4.5: **The LaArDetector class in TraCI4J**

detector for all the lanes of all Traffic Lights in a given network.

Figure 4.5 shows the `LaArDetector` class at TraCI4J, which can be used to grab information from the detector during the simulation.

4.2.4 Multi-Entry, Multi-Exit Detectors

A Multi-Entry/Multi-Exit (MeMe) Detector, or an E3 type Detector, according to SUMO, is basically an extension of an induction loop meant to be used to count vehicles entering or exiting a closed area. In fact, it works as a distributed collection of entries and exits counting how many vehicles entered and exited the area. It can also be seen as an extension of a Lane Area Detector, where the covered area can span for many blocks, or neighborhoods. Equally to InductionLoops and LaAr detector, MeMe Detectors have to be defined within an additional file which has to be loaded by the simulation. Due to its distributed nature, the descriptions of MeMe Detectors have to include the set of entry- and the set of exit-cross-sections. Due to this, it is not possible to use a single tag to specify a detector. Instead, the description consists of the following parts:

- A beginning tag that describes some global attributes of the detector just as the descriptions of inductive loop detectors and area detectors do.
- A set of tags that describe the detector's entry points.
- A set of tags that describe the detector's exit points.
- A closing tag that must match the opening tag.

An example of a MeMe Detector representation is given in Box 4.5. The file with all the MeMe Detectors must be also included in the configuration file, in the same way as for Induction Loops and Lane Area Detectors.

Again, in order to automatically generate MeMe Detectors for a given network topology, a script called `generateTLSE3Detectors.py` is provided and can be used to install MeMe detectors in all inputs and outputs surrounding a circular area of a defined distance around the Traffic Lights available in the network topology.

Box 4.5: Example of a Multi-Entry Multi-Exit Detector representation. The procedure is similar to an inductive loop definition.

```
<additional>
   <entryExitDetector id="<ID>" freq="<AGGREGATION_TIME>" file="<OUTPUT_XMLFILE>"
   timeThreshold="<FLOAT>" speedThreshold="<FLOAT>">
      <detEntry lane="<LANE_ID>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detEntry lane="<LANE_ID>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detExit lane="<LANE_ID>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>
      <detExit lane="<LANE_ID>" pos="<POSITION_ON_LANE>" friendlyPos="<BOOL>"/>

      ... further entries ...

   </entryExitDetector>
</additional>
```

Figure 4.6 shows the `MeMeDetector` in TraCI4J, illustrating the available methods for gaining information from this sensor.

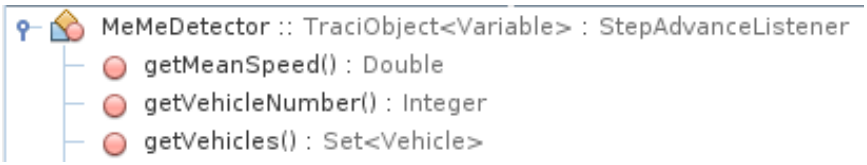


Figure 4.6: **The MeMeDetector class in TraCI4J**

4.2.5 Actuating on Traffic Lights

The most complex class provided by TraCI4J for interacting the SUMO simulation is the class `TrafficLight`. A complete description of this class is beyond the scope of this document. In fact, operating with Traffic Lights is the most difficult part of using TraCI4J.

Figure 4.7 shows an example of how the program of a traffic light can be created/adapted using NETEDIT.

All possible phases need to be defined and checked for all possible junction inputs and

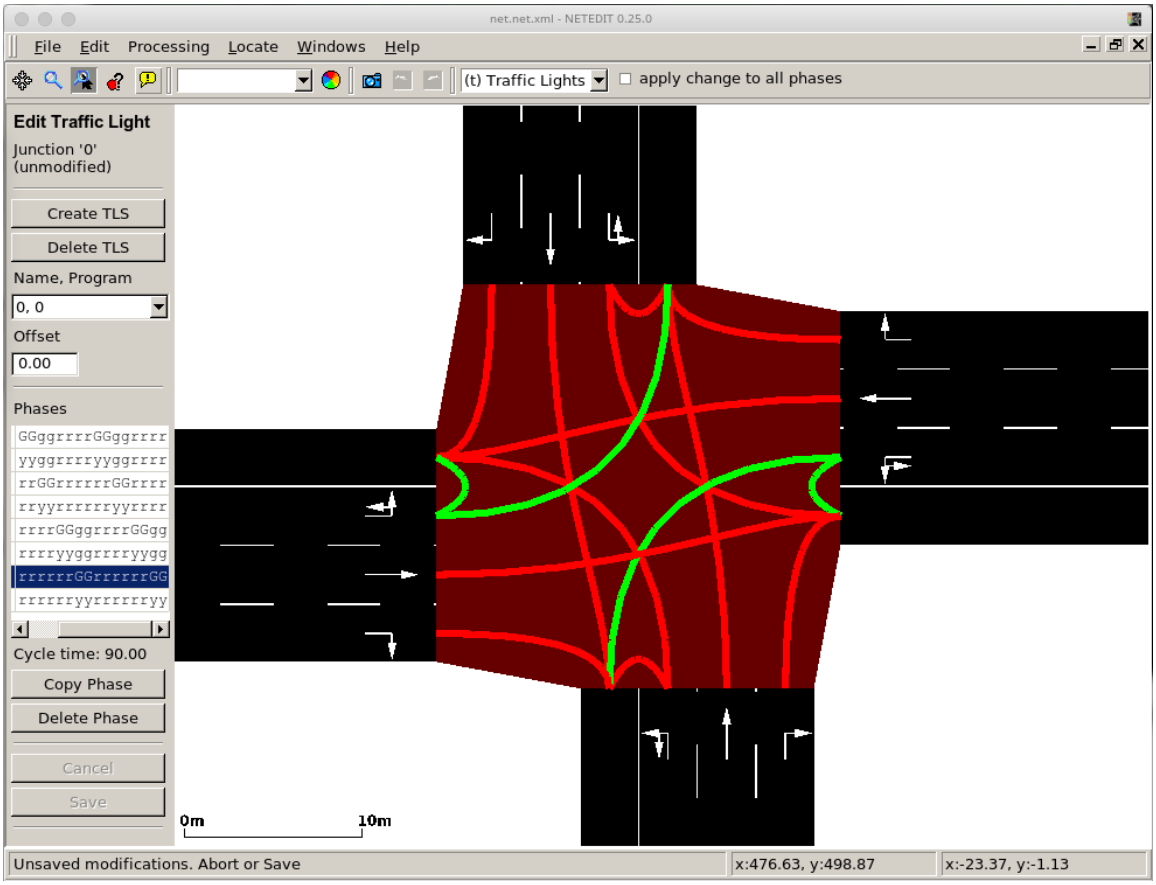


Figure 4.7: **An Example of Traffic Light edition in NETEDIT**

outputs. Usually, when you import a map from OSM or other tools, the `NETCONVERT` tool generates a set of traffic lights phases which is feasible and usable in the simulation, without creating crossing flows which might result in accidents. But this is not a guarantee that the best program for the traffic lights is the one generated by `NETCONVERT`. There might be situations in which some kinds of conversions must be avoided and forbidden, and the traffic lights programs replaced by a different one. In fact, TraCI4J allows for a complete reprogramming of Traffic Lights, with the `TrafficLight` class. During the simulation, the programmer can fully replace the allowed states of a Traffic Light. It provides access to the current program (and the programmed times), which can be used, if the new control so wish. But this program can be completely replaced. So, the programmer has full control on the traffic lights. Among the many possible interactions with a Traffic Light through the `TrafficLight` Java class, the programmer is able to:

- Get access to all the lanes reaching the junction controlled by the traffic light
- Get access to the current program (automatic time-based program) stored in the traffic light
- Get access to the map linking all junction input lanes to all output lanes.
- Get access to the actual state of the traffic light (the current phase)
- Get access to how much time has passed since the last phase switch.
- Get access to how much time the current phase is supposed to remain.
- Change the complete program definition
- Change the current Phase (including defining completely new possible light states - a new phase, not already in the program)
- Change all the phases duration.
- Change the phases order

Even though this is a very powerful class, the task of controlling a Traffic Light is potentially the most difficult task for a beginner. Unfortunately, the design of methods available in this class for the programmer is still needing some improvement in the current version of TraCI4J, and will require some exercise from the beginner, if a more complex behavior is necessary. Nevertheless, there are examples and tutorials that might guide the beginner through a process of creating his/her own classes and methods for operating traffic lights.

4.2.6 Actuating on Vehicles

Another complex class in TraCI4J (even though not so complex as the TrafficLight class) is the `Vehicle` class. It is important to know that in SUMO, each Vehicle is an object with its own attributes and behavior. Because there are many vehicles which in principle might behave equally, it is possible to define Vehicle Types, generalizing some properties which will remain the same for all the vehicles of the same type. From the point of view of a Vehicle Type, it is possible to redefine most of the vehicle type parameters, like acceleration and deceleration, length, maximum speed and minimum gap to other vehicles. Specifically to each Vehicle, it is possible to define and redefine the vehicle color, the change lane model, the route intended by the vehicle, the speed and the final target. It is possible still to grab information about the vehicle CO2 emission, the CO emission, the actual edge where

the vehicle is, the actual lane, the position within the lane, the fuel consumption, the Hc emission, the noise emission, the NOx emission, the PMx emission, the overall position (not the relative position within the lane) and the vehicle's id and type.

Again, just like in the case of Traffic Lights, to actuate in a vehicle will demand some responsibility. As it is possible to change vehicle's routes, there is always a chance that the reprogrammed route is unfeasible. In this case, this will make the vehicle to be "teleported" out from the simulation before the final route is reached. If you change the speed, SUMO will assume that you are controlling the vehicle and will maintain the speed until you command another change in speed or give back to SUMO the control (by setting a speed of -1). This can potentially cause accidents and indeed SUMO is prepared to handle situations like accidents and crashes that might congestion the affected lane and create delays in traffic, just like in a real case.

4.2.7 TraCI4J Limitations

It is important to know that the TraCI protocol is much more elaborated than what is available in TraCI4J. There are many functions of the TraCI protocol, e.g. for interacting with People, with Public Transportation methods like buses and trains, where people can get in and out, and even with more complex information which is available from sumo, but not available through the actual implementation of TraCI4J.

If you check the TraCI documentation and verify that indeed the kind of information you need is not available in TraCI4J, you might probably have to switch to other languages, like Python or C++ which have a better interface to TraCI, or try to extend TraCI4J in order to implement the missing functions. One of the biggest problems with this last alternative is that unfortunately, the code of TraCI4J is a little bit cumbersome, and difficult to understand and consequently, difficult to receive contributions, specially from a novice. One particular design decision which makes contributions more difficult is the decision from the TraCI4J developers to automatically generate some Java classes from XML templates, which is possible through Maven. Another shortcoming is that the TraCI4J was developed to be compiled through Maven. Even though the developers provided an Ant script for those not acquainted with Maven, this works only for the code as it is downloaded from GitHub repository. Any small modification might require significant engineering in order to be compiled, due to these strange design decisions, which very much limit outside contributions to the project.

4.2.8 Simple Code Examples

In order to illustrate how a beginner should organize his/her program to control a SUMO simulation, this section presents some small fragments of code meant to point out the main requirements for creating a program able to interact with SUMO.

One important required information is that, when controlled by and external program, SUMO submits completely itself to the external program. This means that the simulation mode is completely different than a standalone simulation.

The difference start in the sumo initialization. In order to have external programs

interacting to it, SUMO needs to be started with the `--remote-port <INT>` option. This means that sumo should not start immediately, but should prepare itself and awaits until a first interaction comes from the network through port `<INT>`. After this first interaction, SUMO has an internal variable `TargetTime`, which rules how much time SUMO should keep simulating the environment without expecting a further communication from the external program. If this `TargetTime` is reached, without a further communication from the external program, SUMO stops and waits for the external program to communicate before re-starting the simulation. This means that your program must be continuously polling SUMO for it to run the simulation. There is also the possibility of setting this `TargetTime` to be 0, what means that SUMO will always be waiting for the next poll from the external program, and will run a simulation step by step.

The first requirement is then to create a connection from your program to SUMO. This is achieved through the `SumoTraciConnection` class. There are basically two ways for using this class. Both of them are shown in Box 4.6.

Box 4.6: Stablishing a connection with SUMO
The following code is used to connect to a running sumo instance:

```
SumoTraciConnection sumo;
try {
   sumo = new   SumoTraciConnection(InetAddress.getByName(ipServidor), port);
} catch (Exception e) {e.printStackTrace()}
```

The following code is used to let TraCI4J to start a new sumo or sumo-gui instance and try to communicate with it

```
SumoTraciConnection sumo;
sumo = new SumoTraciConnection(config_file, (int) System.nanoTime());
sumo.addOption("start", "1"); // auto-run on GUI show
sumo.addOption("quit-on-end", "1"); // auto-close on end
try {
     sumo.runServer(withgui);
    } catch (IOException | InterruptedException e) {e.printStackTrace();}
```

In the first code fragment in Box 4.6, your program assumes that sumo (or sumo-gui) is already running, and your program just needs to know the server address running sumo and the port in which it was started. In the second code fragment, you will be asking TraCI4J to start the sumo program itself. In this case you don't need to worry with defining a port, because TraCI4J will take care of this. You might then have to give some external commands, first to define some options for the sumo program (which is performed using the `addOption` method of the `SumoTraciConnection` class) and the `runServer` method, where you will be able to define if you wish to start sumo with or without the GUI. Also, in this case, you need to provide the file name for a SUMO configuration file, which will rule the instance of SUMO being started.

Supposing you don't have path problems, and the SUMO programs can be easily found by your operational system, after that you will find yourself with a running instance of sumo, with (or without) a GUI showing the network topology defined in your configuration file.

The next step is now, from your program, to enter into an infinite loop where at the end of this loop, your program calls the `nextSimStep()` method, as shown in Box 4.7.

Box 4.7: Calling the `nextSimStep()` method within an endless loop.

```
int timeStep = sumo.getSimulationData().queryCurrentSimTime().get();
// Static information
List<TrafficLight> ltl = getTrafficLights(); // created method
List<Lane> ll = getLanes(); // created method
List<Edge> le = getEdges(); // created method
List<Junction> lj = getJunctions(); // created method
// Dynamic information (require updates)
List<Vehicle> lv = getVehicles(); // created method
List<InductionLoop> lil = getInductionLoops(); // created method
while(exitcondition == false) {
   sumo.nextSimStep();
   updateVehicles();       // created method
   updateInductionLoops(); // created method
   nextphase = DecidePhase(); // created method
   setCurrentState(nextphase); // created method
   sleep(timeStep);  // Thread.sleep()
}
sumo.close();
```

In this code fragment, we first get from sumo the current `TargetTime`, and use it to let your program to sleep using `Thread.sleep()` or something else. Then you might collect the static structure of your network (if you wish), and further the dynamic structure (because the list of vehicles and the state of the induction loops might change from each simulation step). Then you enter into a loop where you first make the poll to sumo, collecting the newest information, then you update your dynamic information, decide how to control your traffic lights, actuate in the traffic lights and sleep for the next loop.

All the methods indicated as created methods in box 4.7 will require other TraCI4J commands in order to be implemented. For example, in box 4.8, we show how to grab the actual value of the vehicles repository.

Box 4.8: This code shows how to get the current number of vehicles in the simulation.

```
Collection<Vehicle> vehicles = sumo.getVehicleRepository().getAll().values();
numVeiculos = vehicles.size();
```

In box 4.9, we show how to get all the traffic lights and all the possible phases for each of them.

Box 4.9: This code shows how to get all the traffic lights and all the possible phases for each of them.

```
Map<String,Lane> mapAllLanes = sumo.getLaneRepository().getAll();
Map<String, TrafficLight> mapTrafficLights =
sumo.getTrafficLightRepository().getAll();
for(Entry<String,TrafficLight> trafficLightPairs:mapTrafficLights.entrySet())
{
  TrafficLight trafficLight = trafficLightPairs.getValue();
  /** Traffic Light Phases*/
  ArrayList<TLState> TLStates = new ArrayList<TLState>();
  Logic[] logics=trafficLight.queryReadCompleteDefinition().get().getLogics();
  for(Logic logic: logics) {
        Phase[] phases = logic.getPhases();
    for(Phase phase :phases) {
        TLStates.add(phase.getState());
    }
  }
```

In box 4.10, we show how to change the phase of a traffic light to a desired state.

Box 4.10: This code shows how to change the phase of a traffic light.

```
ChangeLightsStateQuery lstQ = trafficLight.queryChangeLightsState();
lstQ.setValue(trafficLightPhases.get(phaseIndex));
try {
    lstQ.run();
} catch (IOException e) {e.printStackTrace();}
```

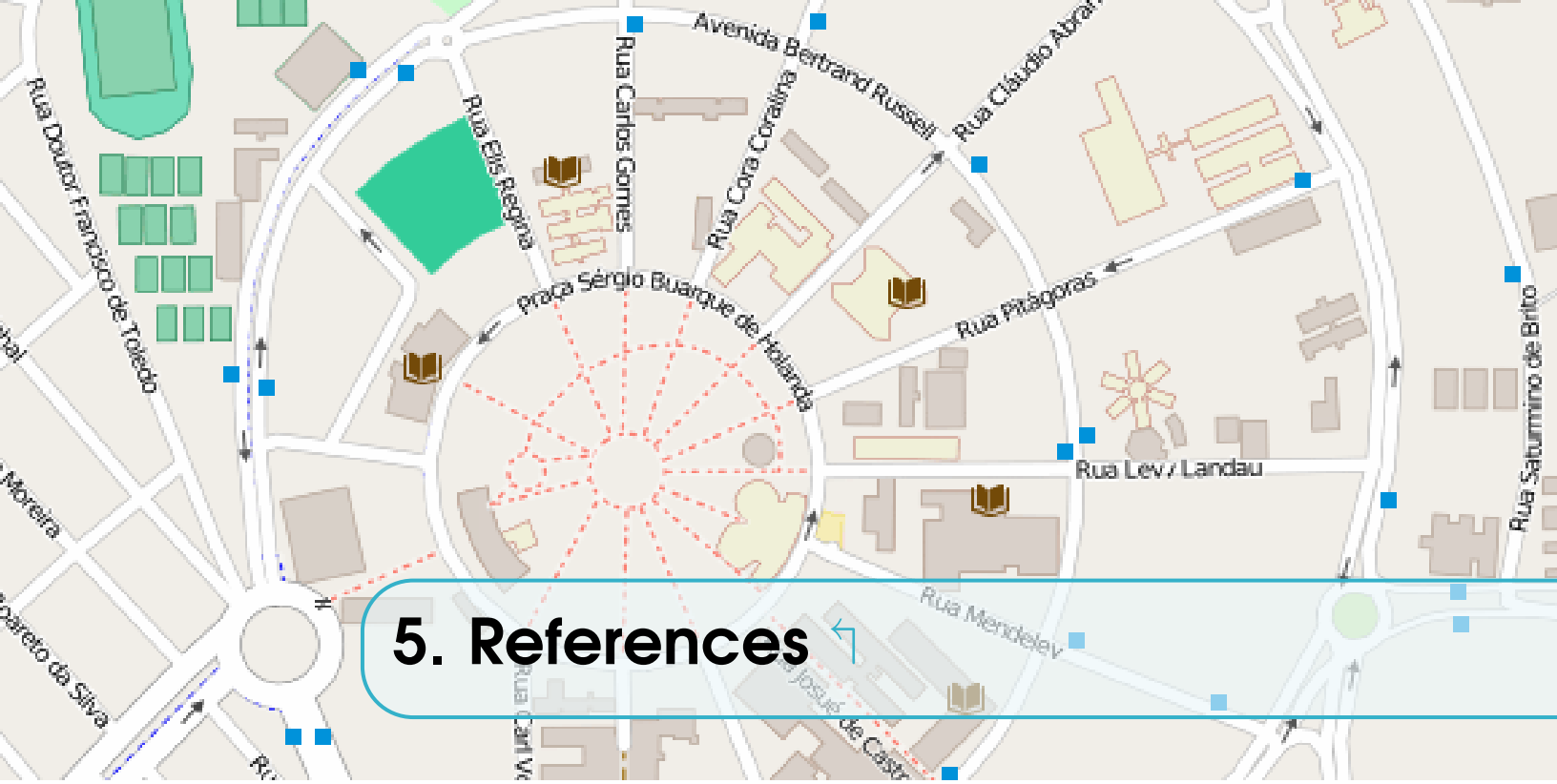
4.3 Conclusion on using TraCI

The use of TraCI4J is not so difficult. If you just need the basic, there are many examples which will show you how to achieve your intention and get and use the information you require for running your simulation. If you need something more complicated, there is a chance you will have to implement it changing TraCI4J code or using another language, where the TraCI implementation is more complete. The wrappers available in other programming languages will work in an analogous manner as in TraCI4J, as all of them are just implementing the TraCI protocol.

There are many other methods and classes available for interaction. Nevertheless, in this section, we have shown the basics of connecting, advancing steps, closing the simulation,

reading values from elements, and writing values (or changing states) to elements, which are most of the types of interaction one will want to do.

To conclude this document, we expect that by this time, you are not a novice anymore, and can start using SUMO and TraCI4J for doing your own simulations.



5. References

SUMO Web Site - http://sumo.dlr.de
The SUMO Documentation - http://sumo.dlr.de/wiki/Main_Page
The Open Street Map Web Site - http://www.openstreetmap.org