1 GatcomSUMO: A Graphical Tool for VANET Simulations Using SUMO and OMNeT++

P. Pablo Garrido Abenza, Manuel P. Malumbres and Pablo Piñol Peral
Dpto. Física y Arquitectura de Computadores; Miguel Hernández University, Spain
{pgarrido, mels, pablop}@umh.es

1.1 Abstract

A graphical application named GatcomSUMO is proposed to facilitate the realization of VANET simulations using the SUMO traffic simulator and the OMNeT++ network simulator. The main aim of this tool is to make easier the steps involved in the setup of VANET simulations. So, the tasks related to the generation of abstract scenarios, the use of network scenarios based on real maps, and the vehicles mobility modelling are integrated in a user-friendly graphical interface. This tool performs the necessary actions by invoking the set of utilities included in SUMO, saving the user from typing complex command-line orders. GatcomSUMO visualizes the network and routes easily, and they are created in such a way to meet the requirements of the involved simulators, in particular OMNeT++, in order to avoid misunderstandings and runtime errors. In addition, the application allows the conversion of the spatial coordinates used by SUMO into those used by OMNeT++, which is essential to place fixed elements like RSUs (Road-Side Unit) in the scenarios. GatcomSUMO is an open-source tool which is developed in Java, so, it can be run on the most commonly used platforms.

Keywords: VANET, SUMO, OMNeT++, Veins.

1.2 Introduction

A Mobile Ad hoc Network (MANET) is a kind of wireless network composed of a number of mobile nodes that can communicate directly with each other, without the use of any infrastructure. In turn, a Vehicular Ad hoc Network (VANET) is a particular kind of MANET in which the communication network is composed of mobile vehicles and fixed infrastructure, like Road-Side Units (RSU), so Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) communications may take place. Whereas in a MANET the nodes can move freely in any direction, in a VANET the vehicles can only move on the roads network. VANETs can be applied to many fields, such as improving road safety or implementing traffic information systems. The use of simulation techniques is essential when working with wireless networks, because the implementation of a real testbed is usually an expensive and difficult task, especially in the case of VANETs. Furthermore, a VANET simulation is a more complex problem because two components are needed: a wireless network simulator to allow the wireless communication between the vehicles, and a traffic simulator for controlling their mobility. Several well known network simulators can be used by the scientific community, such as Riverbed Modeler [15], ns-2 [10], ns-3 [14], OMNeT++ [22], and JiST/SWANS [1]. OMNeT++ is an open-source discrete event simulator that can be used in a great variety of domains; it has

frameworks available for wired and wireless networks, queueing networks, satellite simulators, or cloud computing systems, to name a few. Frameworks like MiXiM [24], INET [21], or Castalia [12] are based on the OMNeT++ simulator and allows to simulate wireless networks.

On the other hand, there are a number of traffic simulators capable of generating mobility traces, which can be later processed by the network simulator. VanetMobiSim [4], VISSIM [8], and SUMO [6] are examples of well known traffic simulators. In real VANETs, the data received by one vehicle may determine its mobility behaviour. For example, when a vehicle receives a message with a warning about an accident located in its route, it can look for an alternative route to bypass the location of the accident. This behavior can be modeled by linking the traffic simulator with the network simulator in some way. SUMO (Simulation of Urban MObility) is a microscopic traffic simulator which models streets and roads, and supports many types of vehicles (multi-modal). SUMO offers an API (Application Programming Interface) named TraCI (Traffic Control Interface) [23] which allows an external application to control the simulation and to get information about the network and the simulated objects, that is, about the vehicles moving in the SUMO environment. In other words, SUMO allows to couple both traffic and network simulations through the TraCI interface. There exist several implementations of TraCI using different programming languages including C++, Java, Python or Matlab. Moreover, the above mentioned network simulators can also be linked to SUMO by means of some middleware like TraNS [13], iCS [5], Veins [19], or VSimRTI [17]. TraNS (Traffic and Network Simulation) was intended for linking SUMO with ns-2, however, according to its website, its development has been suspended. On the other hand, iCS (iTETRIS Control System) connects SUMO with the ns-3 simulator. Veins (Vehicles in Network Simulation) is an OMNeT++ package that implements TraCI in order to connect SUMO with some network simulator framework like MiXiM [24] or INET [21]. Finally, VSimRTI (V2X Simulation Runtime Infrastructure) is a more generic framework, as it allows to connect both VISSIM and SUMO traffic simulators with some different network simulators like ns-3, JiST/SWANS (Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator), and OMNeT++.

For the purposes of our research we have selected OMNeT++ as the network simulator, SUMO as the traffic simulator and the Veins package to incorporate the VANET-specific protocols and to link both simulators. All of them are open-source software, which is a key factor to enhance reproducibility of research [20]. In order to create a simulation, SUMO includes a set of tools (binaries and scripts) that receive files in XML (Extensible Markup Language) format, and generate output files with the same language. Although these tools and the XML files syntax are well documented, those files are difficult to edit by a human. Building a VANET simulation can be a time consuming and prone to errors task for any user, being a truly overwhelming process for a beginner. The aim of the GatcomSUMO tool is to automate every step needed in this task, although it can be used for other purposes too. Specifically, GatcomSUMO allows to generate abstract networks, to download and convert real maps from OpenStreetMap to be used with SUMO, and to generate routes for the mobile nodes. The necessary configuration files for the OMNeT++ simulator can also be generated. In addition, GatcomSUMO also incorporates a set of useful tools when working with wireless networks, such as the communication range and signal propagation loss calculators, and several unit converters (velocity, signal power, temperature) which provide the corresponding configuration parameters to be imported into the OMNeT++ simulator. GatcomSUMO reduces the learning curve for designing and implementing VANET simulations, and helps the researcher to really focus on research.

The remainder of the paper is organized as follows. Firstly, an overview of the SUMO workflow is presented in section 1.3, briefly describing the commands used for generating both the networks and the traffic demands. Section 1.4 shows the additional files needed for building VANET simulations with the OMNeT++ network simulator; if the reader has some experience with these simulators the previous sections can be skipped. In section 1.5, we enumerate some related works and summarize their main achievements. Then, in section 1.6, GatcomSUMO is described in detail. Finally, section 1.7 concludes the paper and proposes some future work.

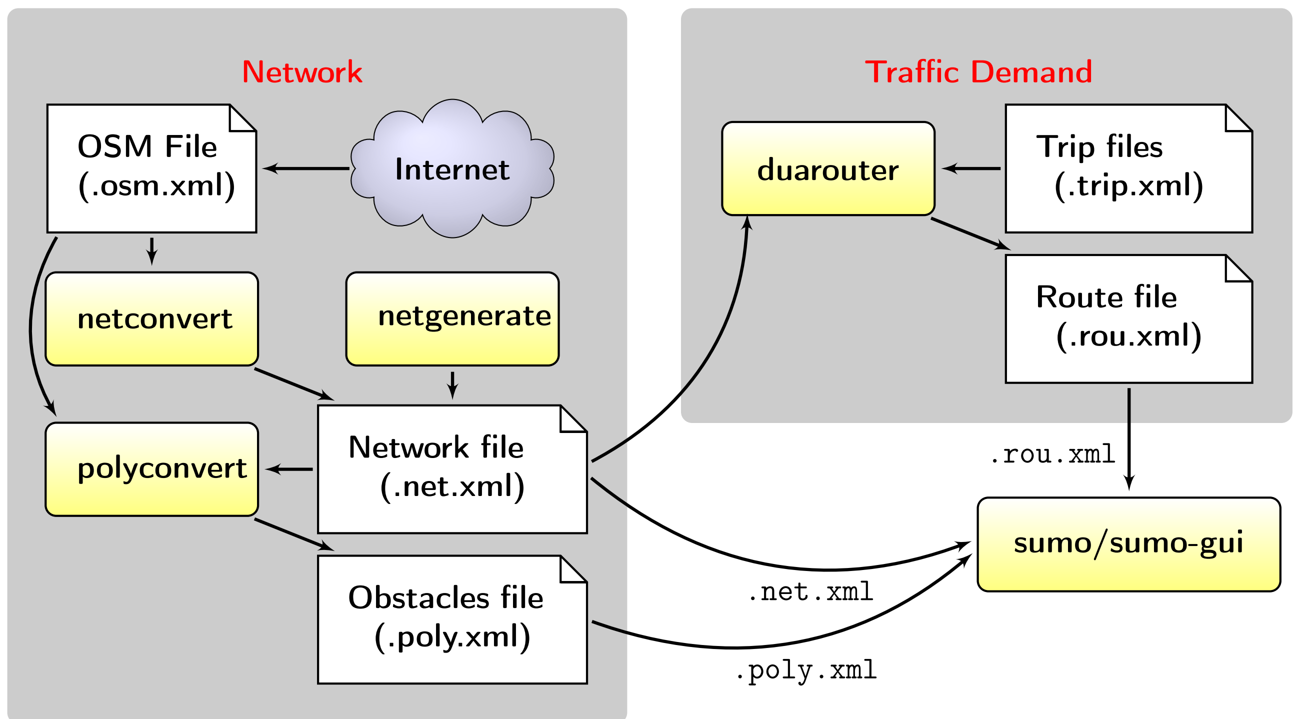


Figure 1.1: SUMO workflow

1.3 SUMO Workflow

This section describes the usual steps needed in order to conduct a simulation that involves to the SUMO simulator. The usual workflow when working with SUMO is depicted in Fig. 1.1. As can be seen, SUMO works with two types of data: the network and the traffic demand, which are explained in the following sections.

1.3.1 Network

The network represents the roads or streets where the vehicles will move along. It is stored in ASCII text files with a plain-XML format (.net.xml). In brief, these files are compounds of a list of streets (*edges*), a list of lanes in each street (*lanes*), and a list of nodes that represents intersections between several streets (*junctions*). Optionally, they can include other information like connections or traffic lights. Fig. 1.2 shows a fragment of the `'net'` section, which is the main part of a network file. Although the SUMO network files (.net.xml) can be edited manually, they are usually generated with some utility; if a network file needs to be modified manually, it is possible to use a visual network editor like the `netedit` tool included in SUMO. There are other command-line tools in SUMO for building either abstract networks (`netgenerate`), or import them from existing networks (`netconvert`). With `netgenerate` the user can generate from scratch three kind of abstract or synthetic networks with geometrical structure: grid, spider, and random networks (Fig. 1.3). GatcomSUMO allows to input the necessary arguments in a user friendly way in order to invoke to `netgenerate` with the desired command line. As a result, a network file (.net.xml) is generated. The following example generates a grid network named 'grid500.net.xml':

```
netgenerate --grid true --grid.number 5 --grid.length 500
            --output-file grid500.net.xml
```

An advantage of the grid networks is that the length of the streets are well known, which allows to validate the desired communication range in our simulations. Moreover, the names of nodes

```
<net version="0.25" ...>
  <location netOffset="-312300.75,-5585447.93"
     convBoundary="10110.00,0.00, 15311.67,6350.00"
     origBoundary="30.355195,50.395771, 30.574502,50.527149"
     projParameter="+proj=utm +zone=36 +ellps=WGS84 +datum=WGS84
        +units=m +no_defs" />
  <edge id="33632393" from="384747802" to="1502836202" priority="5"
     type="highway.service" shape="11371.46,5855.44 11349.99,5778.89 ...">
     <lane id="33632393_0" index="0" speed="13.89" length="181.09"
        shape="11369.87,5855.88 11347.97,5777.79 11354.61,5775.75
           11329.48,5686.17" />
  </edge> ...
</net>
```

Figure 1.2: Network file (.net.xml) - 'net' section

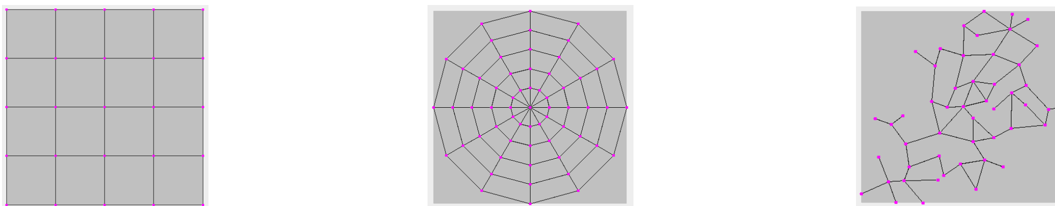


Figure 1.3: Abstract networks: Grid, spider and random

and edges are easy to remember, which is very useful when defining by hand our own routes for the vehicles (traffic demand). For example, an edge defined from node '0/0' to node '0/1' is assigned the identifier '0/0to0/1', and so on.
On the other hand, when working with VANETs scenarios, it is very common to use real maps. With the netconvert utility the user can import road networks in many formats, as OpenStreetMap (OSM) [3] and PTV VISUM/VISSIM [8]. In this work only the OpenStreetMap format is considered, as it has enough data and quality for our experiments. All the maps downloaded from OpenStreetMap are based on the WGS84 (World Geodetic System 1984) coordinate system, which is the same ellipsoid used by GPS (Global Positioning System), and a coordinate system which is very close to the one used by GLONASS (Global Navigation Satellite System). Nowadays, both GPS and GLONASS are widely used to get the positioning information in vehicles and smartphones, so, it is feasible to use real tracks saved with a GPS receiver device on maps downloaded from OSM. All the maps that we have used in this work have been validated with tools like Osmose (OpenStreetMap Oversight Search Engine) [2], and, visually, by overlaying real GPS tracks (.gpx files) with them, by means of JOSM (Java OpenStreetMap) [16] or JGPSTrackEdit [9].
There are several ways for downloading maps from OSM. Firstly, by selecting a rectangular area from a web browser (http://www.openstreetmap.org/export). Secondly, by typing an URI (Uniform Resource Identifier) in a web browser indicating directly the coordinates in longitude/latitude of the south-west and north-east corners of the rectangular area (bounding box) to download. There are many utilities on the Internet to determine those coordinates, such as the one included in OpenStreetMap itself. For example, the following URI retrieves part of the city of Elche by using the OSM API, generating a file named 'map.osm.xml':

```
http://api.openstreetmap.org/api/0.6/map?bbox=-0.697,38.266,-0.677,38.283
```

The OSM API limits the maximum number of downloaded objects (e.g., the number of nodes is limited to 50000), and the maximum size of the bounding box (0.25 degree in either dimension). Nevertheless, it is possible to overcome these restrictions by using the OSM Extended API (XAPI),

```
<?xml version="1.0"?>
<routes>
  <!-- Vehicle types -->
  <vType id="rsu" length="1.0" maxSpeed="0.0"  accel="0.0" decel="0.0"/>
  <vType id="car" length="5.0" maxSpeed="70.0" accel="2.6" decel="4.5"
         sigma="0.5"/>

  <!-- Routes -->
  <route id="route-rsu0" color="Orange"
         edges="0/4to0/3 0/3to1/3"/>
  <route id="route-car0" color="Blue"
         edges="1/3to2/3 2/3to3/3 3/3to3/2"/>

  <!-- RSU/Vehicles -->
  <vehicle id="rsu0" type="rsu" color="Yellow"
           depart="0.0" route="route-rsu0"/>
  <vehicle id="car0" type="car" color="Blue"
           depart="1.0" route="route-car0"/>
</routes>
```

Figure 1.4: Routes file (.rou.xml) - A fixed RSU and a car

```
<trips>
   <trip id="route-1" depart="0.0" color="Magenta"
    from="29350093#0" to="34691079#8" via="136456446#0" />
</trips>
```

Figure 1.5: Trip file (e.g. mytrip.xml)

which allows much larger requests. As it will be explained later, GatcomSUMO builds URIs as shown above for retrieving the OSM maps in XML format, as well as several image formats (PNG, SVG, etc.).
A third option consists in the use of the `wget` command, which receives the previous URI as an argument and exports the map to the OSM file specified:

```
wget "<URI>" -O elche.osm.xml
```

The downloaded OSM file needs to be processed with the `netconvert` utility, which transforms geo-coordinates to metric coordinates of the network map in such a way that can be used by SUMO. The following is the minimal command line for this task:

```
netconvert --osm-files elche.osm.xml -o elche.net.xml
```

In addition, when working with realistic VANETs scenarios, it would be of interest to define the obstacles found in the scenario, like buildings, since they modify the radio coverage of vehicles circulating in their surroundings. At this point, the OSM files have the advantage of providing side information, apart from streets, lanes and junctions, to define extra polygons in the areas where vehicles cannot cross (buildings, parks, etc.). The obstacles can be generated automatically with the `polyconvert` utility, which takes as input a network file (.net.xml) and an OSM file (.osm.xml), and generates a new file (.poly.xml) with shapes that Veins identifies as obstacles. For example:

```
polyconvert --osm-files   elche.osm.xml  --net-file elche.net.xml
            --output-file elche.poly.xml
```

Although polyconvert accepts several input formats, it cannot process the abstract network files, so that kind of networks could not be used in a SUMO simulation with obstacles. However, GatcomSUMO has been extended with the functionality of generating the obstacles file only for grid networks, as it is very easy to compute the polygons as rectangles, without the use of the polyconvert utility.

1.3.2 Traffic Demand

As SUMO is a microscopic traffic simulator, each vehicle follows its own route, which is expressed as a list of edges. The mobility of all vehicles (traffic demand) is written into a routes file (.rou.xml). SUMO allows multi-modal simulations, that is, the use of distinct types of vehicles. The routes file (Fig. 1.4) starts with the definition of the vehicle types according with several physical parameters (e.g. length, maximum speed, acceleration, etc.), followed by the list of edge identifiers (ID) of every single route defined. Finally, the file contains the definition of individual vehicles or sets of vehicles (flows), including its type, the starting time, and the assigned route to follow.
There are several ways to generate these files, such as writing by hand explicit routes for each individual vehicle or flow, specifying the starting and the ending edge and searching the shortest path between both points with some tool (duarouter), using turning percentages at junctions (jtrrouter), using O/D (Origin/Destination) matrices (od2trips) with the PTV VISUM/VISSIM format, or generating them randomly with some scripts like randomTrips.py or dua-iterate.py. Several things should be taken into account especially when writing a route file by hand. Firstly, the routes have to be connected, that is, each edge in a route should be followed by an adjacent edge. Secondly, a route needs at least two edges, even if a vehicle is going to be stopped (speed equals to 0). Thirdly, the route assigned to each vehicle has been previously defined, and finally, the vehicles are sorted by the departure time, otherwise a warning message will be shown indicating that some vehicles will be ignored. Although there exists a script that checks some of the previous conditions (routecheck.py), there are other situations in which OMNeT++ runtime errors occurs, for example, when a vehicle leaves the playground area or moves to a negative coordinate. These conditions are difficult to check without the existence of some utility like GatcomSUMO.
The duarouter tool generates routes by specifying only the source and destination edges, and optionally, some intermediate edges too. This is known as a *trip*, and it is specified in an XML file as shown in Fig 1.5. duarouter receives the road network file (.net.xml) and the trip file and looks for the shortest path between the source and destination edges by means of a particular routing algorithm ('dijkstra' by default); as a result, a route file in XML format is generated:

```
duarouter --net-file elche.net.xml --trip-files mytrip.xml
          --output-file myroute.xml
```

SUMO includes some other tools and many more options, but in this section, only those that are supported by GatcomSUMO have been described.

1.4 OMNeT++ Workflow

As explained above, apart from the traffic simulator, the other necessary component in a VANET simulation is the network simulator. This section briefly explains how to use all the already generated SUMO files with the OMNeT++ network simulator, as well as other required files. The flow of data (⟶) and references between files (---▸) are illustrated in Fig. 1.6, using the city of Elche as an example. The three files generated with SUMO (.net.xml, .rou.xml, and .poly.xml) need to be specified in a file 'elche.sumo.cfg', which is referenced from the following file 'elche.launch.xml'; this file references the same three previous files too. In turn, this file is referenced from the OMNeT++ configuration file (usually named omnetpp.ini). In addition, another XML file is needed with the

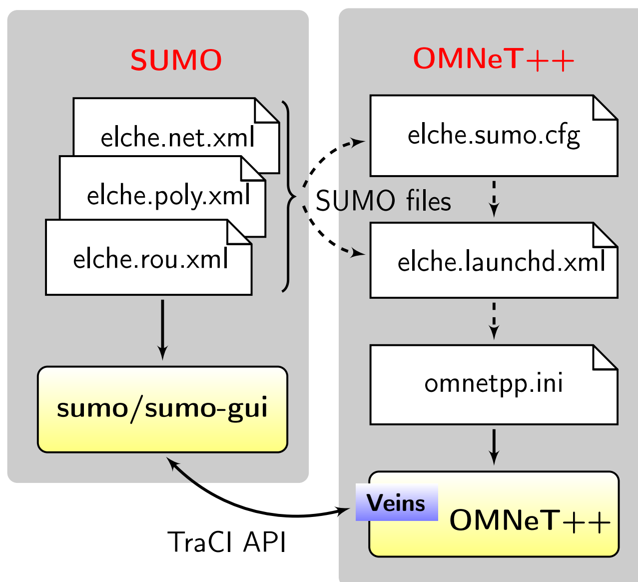


Figure 1.6: OMNeT++ workflow

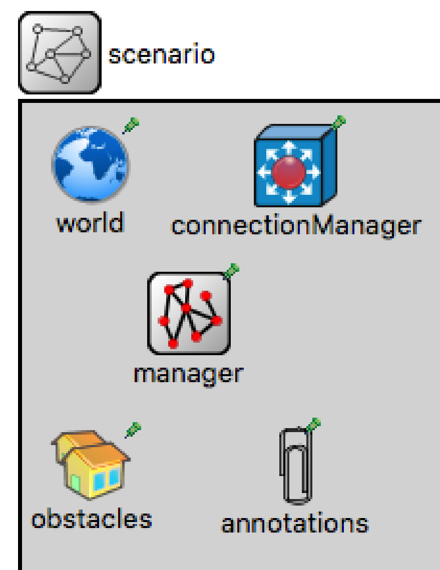


Figure 1.7: OMNeT++ scenario

analogue models parameters and the attenuation of the wireless signal depending on the different types of obstacles in the network (e.g. config.xml). This file should consider each type of obstacle defined in the obstacles file (.poly.xml), such as `building`, `parking`, `industrial`, `...`; on the contrary, they will not be shown in the OMNeT++ simulation window. The content of all of these files are omitted due to the space that would be needed. However, a fragment of them can be found in Fig. 1.18; as it will be shown later, GatcomSUMO generates all of them.
Some objects are referenced from the omnetpp.ini file, which need to be declared in an OMNeT++ scenario (scenario.ned); the design view of such scenario is shown in Fig. 1.7, but the source code is omitted again. Most of them belong to the Veins framework; the most relevant one is `manager`, which is an instance of the `TraCIScenarioManagerLaunchd` Veins class, and it is used by Veins for connecting with SUMO via the `sumo-launchd.py` script, and for creating the objects whose mobility will be controlled by SUMO.

1.5 Related Work

As can be seen, many software components and configuration files are needed in order to build a minimal VANET simulation, and each step requires some experience to do it properly. Several works have been published describing tools that provide additional functionality to the SUMO framework in order to to improve its usability.
eNetEditor [7] has a Graphical User Interface (GUI) that allows the creation of valid road networks for SUMO. The syntax of the network files (.net.xml) has been extended in order to support custom parameters about charging stations for electric vehicles, in a similar way than the definition of bus stops. However, the output network file is still compatible with SUMO. The main work of *eNetEditor* is done by building commands for invoking the SUMO tools. Specifically *eNetEditor* uses `netconvert`, `jtrrouter`, and `dfrouter` for generating routes. In addition, the routes can be calibrated with `duarouter`, in order to take into account traffic conditions, not only static conditions; depending on traffic, the shortest path may not be the fastest one.
TrafficModeler [11] is a tool developed in Java that allows to build the necessary configuration files for SUMO in a user-friendly way. It generates vehicle routes compatible with SUMO using several realistic traffic models. For example, it allows to generate activity-based traffic demand, that is, routes based on the daily activities of the population. Although the main purpose of *TrafficModeler* is

the generation of vehicular traffic for SUMO, it also allows basic modifications to the road networks. *SUMOPy* [18] is a tool written in Python which can be used from a GUI as well as from the command line interface with several provided scripts. The main aim of *SUMOPy* is to make SUMO more accessible to users with no programming experience. It can read and write road networks in the native SUMO format, download a real OpenStreetMap, convert it to a SUMO network file (.net.xml) and extract a polygons file (.poly.xml). Similarly to *TrafficModeler*, it allows to define activity-based traffic demand and additional vehicle types. Other traffic generation models can be used, such as the Origin/Destination (O/D) matrices, the junction turning definitions, or the random trips generators. For this task, *SUMOPy* uses some Python functions or the utilities provided by SUMO, like `od2trips`, `duarouter`, and `jtrrouter`. This tool allows to examine the output files generated as a result of the simulation, which can be done interactively from the GUI.
Despite these tools overcome some handicaps, the GatcomSUMO application presented in this work fits better for the VANET field, especially when using the OMNeT++ network simulator.

1.6 GatcomSUMO

GatcomSUMO was born with the same basic idea as *TrafficModeler* and *SUMOPy*, that is, to simplify the use of the SUMO traffic simulator, specifically with network and traffic demand generation. Similarly to those tools, GatcomSUMO builds commands and delegates its work to the SUMO utilities (e.g. `netconvert`, `duarouter`, etc.). However, SUMO is not the only tool used when working with VANETs environments. The researcher needs a network simulator like OMNeT++ together with some frameworks extensions (e.g. Veins, INET, etc.) for the simulation of wireless networks.
The development of GatcomSUMO represents an effort to simplify each task involved with this complex software tool chain. The learning curve of SUMO is hard, and accomplishing a basic network simulation could take a long time and be frustrating. Even for an expert, doing a new experiment in which either the network or the routes of the vehicles change can be exasperating. GatcomSUMO is an application developed in Java that allows working in a user-friendly way; it does not contain any external script. The only requirements to execute it are to have SUMO and a JRE (Java Runtime Environment) installed, in addition to the executable or JAR (Java ARchive) file of the application.
The graphical interface is structured in several main tabs according to the steps to follow during the setup of a simulation with SUMO and OMNeT++ (Fig. 1.8). Each main tab may contain several secondary tabs to cover all the provided functionality. Firstly, it is necessary to build or import the network, which contains nodes and edges ('Network' tab). Secondly, it is necessary to define the mobility of the vehicles ('Traffic demand' tab). Both the network and routes can be visualized in order to validate them and have a whole view of the network scenario ('Map' tab). Then, the set of files needed by OMNeT++ to run the simulation can be generated automatically ('OMNeT++' tab). Finally, the application includes several tools that provide different functionalities which may be useful for the researcher ('Tools' tab), like the conversion between geographical coordinate systems, some calculators for determining the transmission range and the values of corresponding configuration parameters, and the equivalences between different magnitude units (velocity, temperature, signal power, etc.). All the main tabs will be explained in detail in the following sections.

1.6.1 GatcomSUMO - Network

The 'Network' tab shows the options for either generating abstract networks, or downloading OSM real networks. Concerning to the abstract networks, the application can generate grid, spider and random networks (Fig. 1.3). After writing the values for the required parameters of the selected network, it will be created by pressing the [Create Network] button (Fig. 1.8), which invokes the

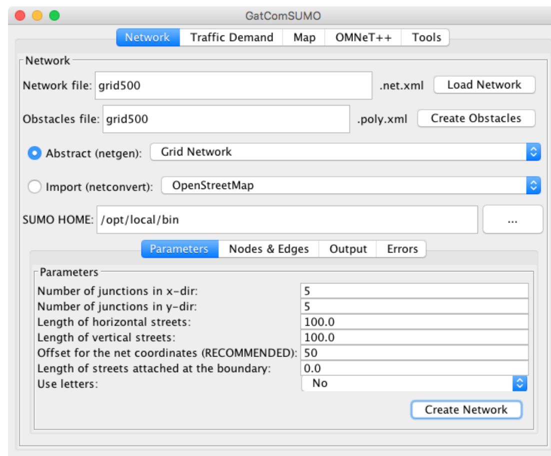


Figure 1.8: Network tab

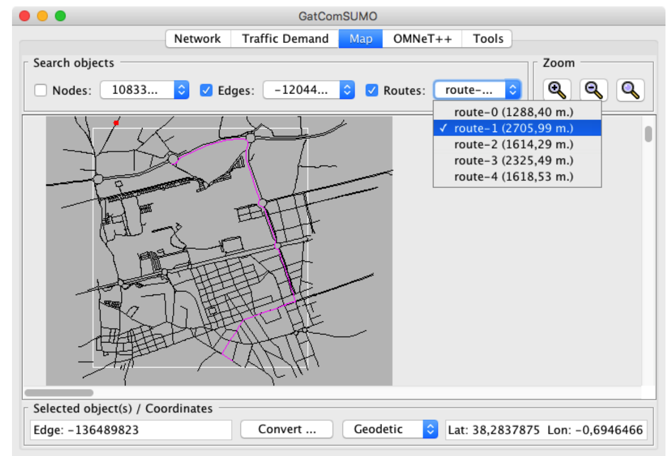


Figure 1.9: Map of city of Elche

`netgenerate` tool in the background (see the previous section for an example). If an offset is not used, the first street will be located at the origin `(0,0)`, and some of its lanes may be assigned to negative coordinates; this is not a problem for SUMO, but an error will be launched when running an OMNeT++ simulation, because vehicles can not move onto negative coordinates. So, it is recommended to add an offset value, and add it to the parameters `*.playgroundSizeX` and `*.playgroundSizeY` in the OMNeT++ configuration file (omnetpp.ini).

Another kind of networks supported by GatcomSUMO are those downloaded from OpenStreetMap. The application allows to download, import and generate the obstacles file in an intuitive way. At the present time it is necessary to know the real coordinates of the rectangle area to download, either geographic or UTM coordinates. Once those coordinates are determined, by clicking the [Download] button, GatcomSUMO builds the proper URI request as a text string taking into account the coordinates specified, creating an instance of the `URLConnection` Java class for retrieving the map. In the end, the geo-coordinates will be used, as shown in the previous section, but this is transparent for the user. Apart from downloading the map in the XML format (.osm.xml), the user can download the same area in a raster or vector graph format as PNG, JPEG, SVG, or PDF. In addition, the download dialog computes the playground size, which can be useful if the user needs to work with an area of an specific size. Once the OSM file (.osm.xml) has been downloaded, it is necessary to convert it to the SUMO network file format (.net.xml). GatcomSUMO allows to do this task by clicking the [Convert] button, which builds the command as a text string for invoking the `netconvert` tool and executes it. A set of predefined options is shown, and they can be changed before the converting process, if needed.

Once the network has been created, either abstract or OSM network, just clicking in [Create Obstacles] will generate the obstacles file (.poly.xml). The `polyconvert` utility cannot deal with abstract networks, but for the specific case of grid networks, GatcomSUMO computes them as rectangles a little smaller than the rectangles formed by the intersections of the streets (Fig. 1.14). In case of OSM networks, the `polyconvert` is used as shown above. From the user's point of view, there is no difference in the way in which the obstacles files are built for each type of network.

Finally, before defining the traffic demand, it is necessary to load the network file into the application. This is done by clicking the [Load Network] button. Once the network has been loaded, the user can visualize the map, identify a node or edge from the map by clicking over it, or locate an object in the map by selecting the edge or node from the list (Fig. 1.9); this will be necessary when specifying edges to build the routes. If the user needs the data created in a previous session, the network and the traffic demand can be loaded by clicking in the corresponding button.

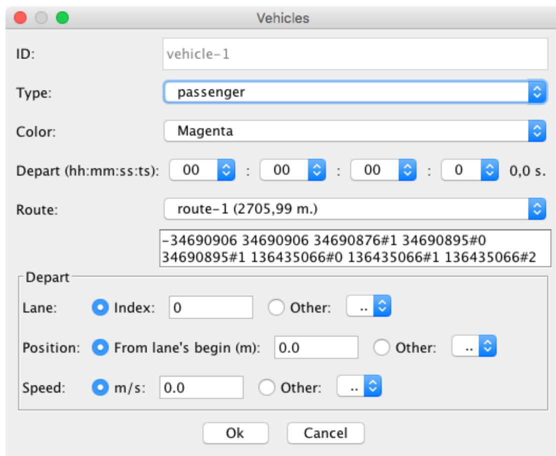


Figure 1.10: Vehicle dialog

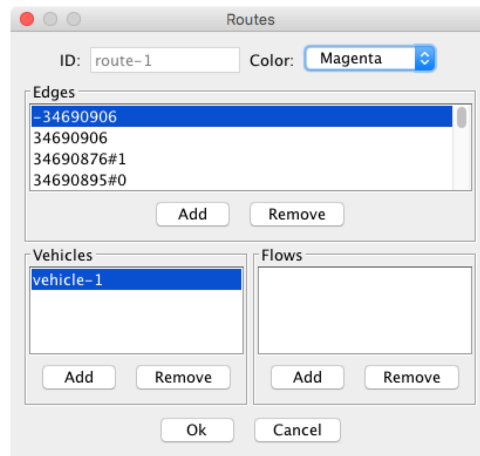


Figure 1.11: Route dialog

1.6.2 GatcomSUMO - Traffic Demand

GatcomSUMO can easily generate robust routes and write them to a route file (.rou.xml). At first place, the user has to define vehicle types, although the application already incorporates several types, one for each vehicle class defined in SUMO by default, like `passenger`, `emergency`, `taxi`, or `bus`, to name a few. After that, a route can be defined by hand by selecting edges from a list of edges. When the user selects an edge from the list, only the adjacent edges to the last added one are shown, making the process faster and, at the same time, avoiding the possibility of building an unconnected route. The user can visualize any route in the map, with the associated color to validate it before running the simulation (Fig. 1.9). If some vehicles are already defined, the created routes can be assigned to one or more of them, either by selecting its route from the vehicle definition (Fig. 1.10), or by adding a vehicle to the current route in the route dialog (Fig. 1.11).

The above procedure is only feasible when routes are known and there are few routes to create. Another way to create new routes is the use of *trips*; a *trip* is a route computed knowing only the origin and destination edges, which may be selected from the list of available edges in the map or randomly. Optionally, the user can add one or more intermediate edges (Fig. 1.12), in order to force the route to cross a particular edge in the path to the destination. Once the required data has been defined, the program generates an intermediate file for the trip (.trip.xml) which is used as input to the SUMO `duarouter` tool, together with the network file (.net.xml), in order to look for a path between source and destination with some routing algorithm; currently, GatcomSUMO uses the shortest-path Dijkstra's algorithm only (see the previous section for an example). The output file of `duarouter` is a route file (.rou.xml) containing the list of edges of the route found, creating a route object in memory. In case of building a greater number of routes, GatcomSUMO allows to specify a number of iterations, and the previous process is repeated as many times as the user requests. This works similarly to the `randomTrips.py` script, but with the following improvements. Firstly, the program detects, at each iteration, if a valid route between a given source and destination cannot be found; for example, when an isolated edge is selected as the origin or destination of the route. In that case, the application ignores the current iteration and retries again with a different source or destination (in case of being randomly choosen). Secondly, the program allows to apply some filters before or after the route creation process: (1) the length of the generated routes should be less or greater than the specified value, (2) to filter routes that have some edge outside the bounding box, because OMNeT++ generates a runtime error when a vehicle leaves the simulation area, and (3) it is also possible to specify the vehicle type permitted by the edges within each route; otherwise, a runtime error occurs when a vehicle tries to move through an edge that does not allow that kind of vehicle. At the end of the process, the user can preview the generated temporary routes, their distance, and the list of edges that compose them. When the user selects one or more routes,

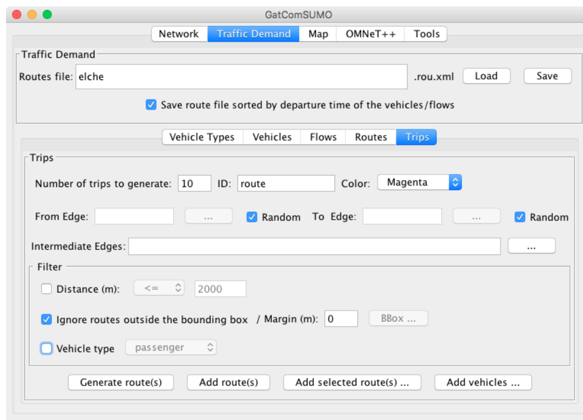


Figure 1.12: Trips definition

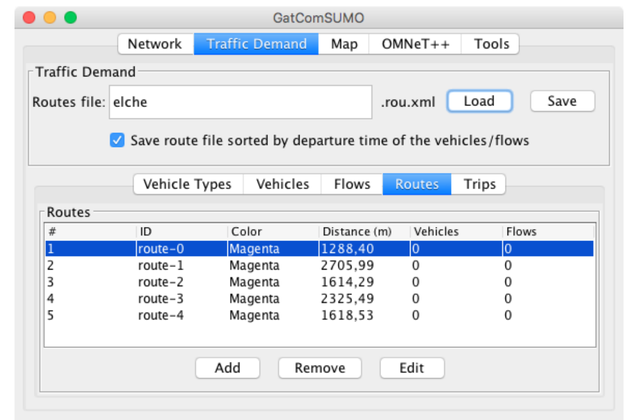


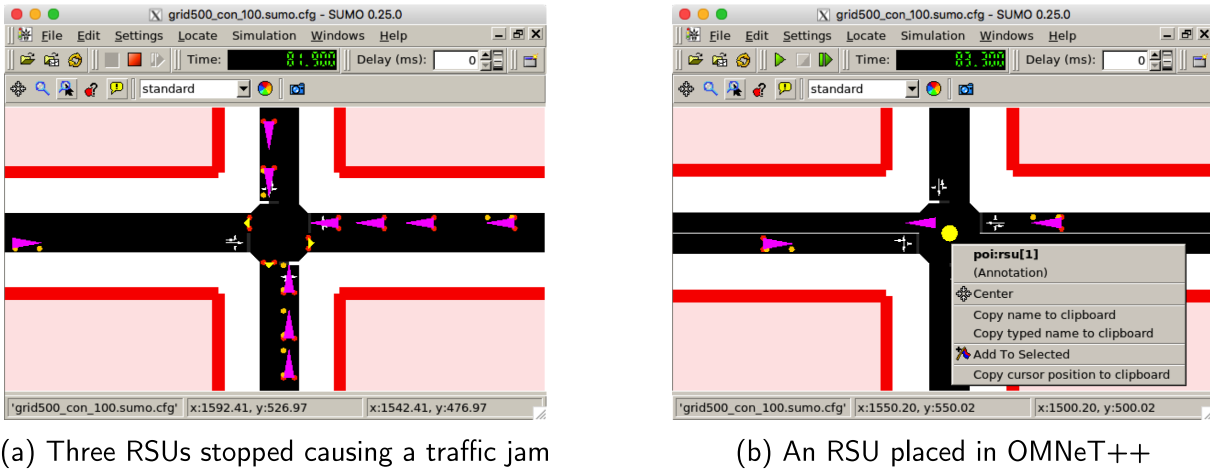
Figure 1.13: Table of defined routes

the minimum playground size to enclose all the selected routes is computed and shown at bottom. Once the desired routes are selected, they can be finally added to the routes table by clicking the [Add route(s)] button (Fig. 1.13). In order to visually check the coherence of each route, the user can select them from the 'Map' tab. If the bounding box filter is selected, no route should leave that area, which is shown as a white rectangle on the map (Fig. 1.9); this helps in keeping the average vehicle density constant in a specific area. An additional utility allows to compute distances between two points, which is useful for validating the length of any street, the bounding box of the network, or the communication range in a wireless network. GatcomSUMO offers the possibility of creating either vehicles or vehicle flows automatically and assigning routes to them. The identifiers, type of vehicle, and color can be specified, as well as the initial departure time of a vehicle, and the increment value for each one.
Finally, all the data described in this section will be saved into a single route file (.rou.xml), that can be previewed in sumo-gui or used to run an OMNeT++ simulation. It is worthy noticeable that the vehicles and vehicle flows will be sorted by their departure time, avoiding the above mentioned warning. The process of writing a route file has been hugely simplified with GatcomSUMO, as the user does not need to write routes by hand, neither to invoke `duarouter` repeatedly.

1.6.3 Placing RSUs

In VANETs scenarios is very common to work with both mobile vehicles and fixed antennas or RSUs (Road-Side Unit). When it is necessary to set the position of a fixed object there are two choices: (a) by defining a 'mobile' car without mobility in SUMO, or (b) by specifying its position in the OMNeT++ configuration file (omnetpp.ini).
The first case consists in defining a car based on a vehicle type defined in the SUMO route file (.rou.xml) whose maximum speed is set to 0; in addition, as it has been said before, a route needs two edges at least (see Fig. 1.4). However, this option has some disadvantages, such as setting the exact RSU position; the object will be located at the beginning of the first edge specified in the assigned route, and oriented to the second one. Furthermore, the vehicle should be kept 'hidden' because it will be considered a vehicle just like all the rest, so, its presence could affect to other mobile vehicles and could cause a traffic jam (Fig. 1.14a).
A better alternative could be to specify the exact coordinates for the RSU in the OMNeT++ configuration file (omnetpp.ini). The following example sets the position `(1500,500)` for the OMNeT++ object `rsu[1]`:

```
*.rsu[1].mobility.x = 1500
*.rsu[1].mobility.y =  500
```



(a) Three RSUs stopped causing a traffic jam

(b) An RSU placed in OMNeT++

Figure 1.14: Placing RSUs (yellow) and cars (magenta)

However, a problem appears at this point since the coordinate systems used in SUMO and OMNeT++ are different. Although both of them use Cartesian coordinates, SUMO considers the origin `(0,0)` in the bottom-left corner and the Y-axis grows upwards, whereas in OMNeT++ the origin is located at the top-left corner, and the Y-axis grows downwards (Fig. 1.15). In addition, SUMO applies an offset to all the coordinates of the network files (.net.xml). As said before, the maps downloaded from OSM use geodetic coordinates (i.e. Longitude/Latitude), which are converted into metric coordinates by `netconvert` SUMO utility by applying a projection to an specific UTM zone in order to minimize the distortion. Next, the UTM coordinates are translated into the origin `(0,0)`, that is, an offset is applied. In this way, the values in the whole file are much lower than the real ones, and these are the values stored in the SUMO network file (.net.xml). At the beginning of the network file, below the `net` root element (Fig. 1.2), there is a child element named `location`. This element contains the `projParameter` parameter, which shows the data of the applied projection: UTM zone and ellipsoid, usually WGS84 (the process of converting geo-coordinates into UTM coordinates is outside of the scope of this work). On the other hand, the value of the `netOffset` parameter contains the offset applied to each UTM pair `(x,y)` in order to do the translation into the final SUMO coordinates.

In the context of this work, the most relevant conversions are those between SUMO and the OMNeT++ network simulator. As it has been said before, this is necessary for placing fixed objects in

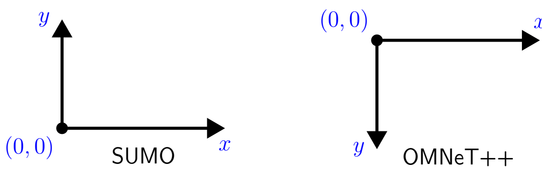


Figure 1.15: Coordinate systems

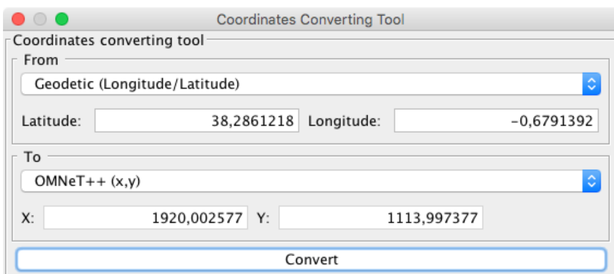


Figure 1.16: Coordinate systems conversion

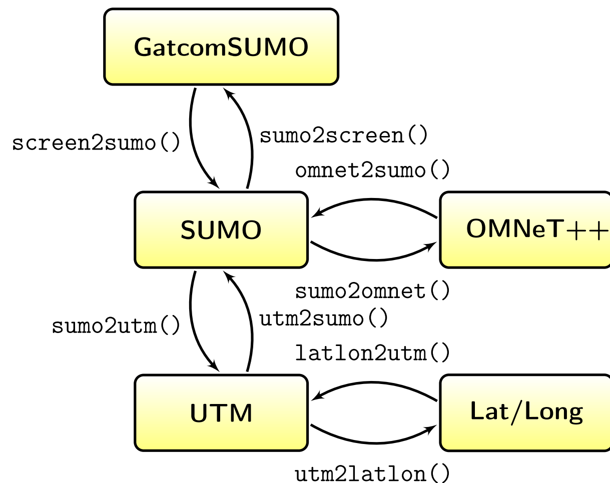


Figure 1.17: Conversion functions

the network like RSUs by specifying coordinates in the OMNeT++ configuration file (omnetpp.ini). As such coordinates cannot be determined from the SUMO graphical view (sumo-gui), it is necessary to convert from one coordinate system to the other. There exist two functions in the Veins framework named `traci2omnet()` and `omnet2traci()` within the `TraCIConnection` class, which do the conversions from SUMO to OMNeT++ and vice versa, respectively. However, both functions are declared as `private` and they cannot be invoked by a user module outside that class. Moreover, the problem is that the user needs to know which coordinates to write in the omnetpp.ini file before running any OMNeT++ code.
GatcomSUMO is able to do all the necessary conversions as explained next. The `convBoundary` parameter of the `location` element is necessary for this task. This parameter is the bounding box or rectangle that encloses the network in SUMO coordinates: `xmin`, `ymin`, `xmax`, and `ymax`. In addition, it is necessary to take into account that the direction of the Y-axis is the opposite, for which it is necessary to compute previously the height of the network. In order to convert SUMO coordinates `(xs,ys)` to OMNeT++ `(xo,yo)` and vice versa, the implemented functions are like the following ones (in pseudo-code):

```
function sumo2omnet(xs,ys) {
   height = convBoundary.ymax - convBoundary.ymin
   xo = xs - convBoundary.xmin
   yo = height - (ys - convBoundary.ymin)
}

function omnet2sumo(xo,yo) {
   height = convBoundary.ymax - convBoundary.ymin
   xs = xo + convBoundary.xmin
   ys = height - (yo + convBoundary.ymin)
}
```

In addition, the Veins class `TraCIScenarioManager` uses a `margin` parameter, which is a value added to the vehicle positions received from SUMO (25 by default), and it should be also considered in the previous functions. Currently, that value is not considered in GatcomSUMO, but this is solved by adding it to the coordinates of each fixed object in the omnetpp.ini file. For example, the following fragment specifies a margin of 25 and defines an iteration variable for adding that value to the position of a fixed RSU named rsu[1]:

```
*.manager.margin = ${margin=25}
*.rsu[1].mobility.x = 1500 + ${margin}
*.rsu[1].mobility.y =  500 + ${margin}
```

Sometimes it would be interesting to convert between real geo-coordinates or UTM, and OMNeT++ or SUMO coordinates. For example, the user could be interested in placing an RSU knowing its Longitude/Latitude or UTM coordinates, or in a selected point in the sumo-gui window; after doing the proper conversion to OMNeT++ coordinates, the user could specify them in the omnetpp.ini file. Although the conversions described before are not complicated, doing them manually is a tedious task and prone to errors. The application allows to convert coordinates in two different ways: (1) by clicking with the mouse on the map (Fig. 1.9), and (2) by typing the exact values to convert by means of a converter (Fig. 1.16). GatcomSUMO works with the SUMO coordinate system, and includes functions to do conversions between all the above mentioned coordinate systems, directly or indirectly (Fig. 1.17). The names of the functions are self-explanatory.
An advantage of placing the fixed objects in OMNeT++ is the possibility of using sequences of values and parallel iteration within the omnetpp.ini file. With this OMNeT++ features it is possible to repeat some experiment varying the position of the fixed objects. In the next example, three positions for the `rsu[0]` object are defined: `(500,500)`, `(600,600)` and `(700,700)`, being `x` and

y two iteration variables that change in parallel:

```
*.rsu[0].mobility.x = ${x=500..700 step 100}
*.rsu[0].mobility.y = ${y=500,600,700 ! x}
```

An additional advantage has to do with the type of the OMNeT++ module that Veins uses to instantiate the vehicle objects in SUMO, as well as their name. Both parameters are specified by setting the parameters `moduleType` and `moduleName` of the `TraCIScenarioManager` object defined in the scenario, respectively. For example:

```
*.manager.moduleType = "es.umh.nodes.car"
*.manager.moduleName = "car"
```

If the specified module code will be in charge of modelling the behaviour of both mobile (i.e. cars) and non-mobile (i.e. RSUs) objects, this could be large and confusing. Moreover, with the previous declaration, all the objects will be named `car[0]`, `car[1]`, `...`, regardless of the type of object. In order to use different module codes and names for each case, the alternative is to use fixed coordinates in the omnetpp.ini file for the fixed objects, as explained before. In this way, the fixed objects could be named as `rsu[0]`, `rsu[1]`, `...`, whereas the mobile vehicles can be defined as usual and they will be controlled by the road traffic simulation.
To sum up, the possibility of setting the position for the fixed objects in the omnetpp.ini instead of setting it in SUMO has the following three advantages:

- To be able to place the non-mobile objects in the exact position in the scenario, and without interfering with the rest of the traffic.
- To be able to define a sequence of values for the position of fixed objects in order to repeat some experiment varying their position.
- To be able to use a module code for fixed objects different from the one used for mobile nodes whose mobility is controlled by SUMO, improving the readability.

As a minor drawback, the objects placed as explained in this section will not be shown in sumo-gui. This could be a problem if the user desires to validate their position or take a screenshot. However, this can be easily solved by using POIs (Point of Interest) for each fixed object. For example, the following snippet creates a small yellow circle as a POI in sumo-gui when that window is open and the object receives the first self-message:

```
/*************  RSUApp.cc  *************/

using namespace Veins;

cModule*                host;
BaseMobility*           mobility;
TraCIScenarioManager*   traciMng;
TraCICommandInterface*  traciCmd;
TraCIColor              color_poi;

bool                    firstTime;

void RSUApp::initialize(int stage) {
  host = findHost();
  mobility = dynamic_cast<BaseMobility*>(host->getSubmodule("mobility"));
  traciMng = TraCIScenarioManagerAccess().get();
  traciCmd = traciMng->getCommandInterface();
  color_poi = TraCIColor::fromTkColor("yellow");
```

```
  firstTime = true;
}

void RSUApp::handleSelfMsg(cMessage *msg) {
  if (firstTime && ev.isGUI()) {
    firstTime = false;
    Coord pos = mobility->getCurrentPosition();
    traciCmd->addPoi(host->getFullName(),"Annotation",color_poi,6,pos);
  }
}
```

Once the POI has been created, it will be shown in the sumo-gui window (Fig. 1.14b), and it could be searched by its name as any other POI by using the option POI within the Locate menu.

1.6.4 GatcomSUMO - Simulation Files

As explained in section 1.4, a VANET simulation requires the files used by the SUMO traffic simulator (.net.xml, .poly.xml, .rou.xml), and a set of additional files needed by the OMNeT++ network simulator. With the 'OMNeT++' tab, the user can generate automatically all such files, instead of copying them from a previous project and changing the name and some other parameters. In the example shown in Fig. 1.18, by clicking the [Generate files] button, the following files will be generated and shown, according with the selected options: elche.sumo.cfg, elche.launchd.xml, config.xml, and a fragment of the omnetpp.ini file (only the part related with the `TraCIScenarioManager` object and a recommended minimum size for the playground are included). GatomSUMO takes into account the different kinds of polygons defined in the network (.poly.xml file) and appends an obstacle type for each one in the config.xml file. Some other files are also generated, like a shell script `run.sh`, useful for running the simulations sequentially or in parallel if a multi-core or a multi-processor system is available.

1.6.5 GatcomSUMO - Tools

The application includes the 'Tools' main tab, which contains several useful converters and calculators (Fig. 1.19). When working with wireless networks in the OMNeT++ network simulator, it is necessary to specify some values in some specific units, and the user has to take his/her pocket calculator, to use a spreadsheet, or to look for some utility in the Internet. The aim of this tab is to have all the necessary utilities in a centralized place.
The main tool deals with the Free-Space Path Loss (FSPL), which needs to be know when dealing with radio communications, for example, for computing the communication range. FSPL is the loss in the signal strength that occurs when a wave is sent through a medium without obstacles. The following formula computes the FSPL in mW:

$$FSPL(mW) = \left(\frac{4 \cdot \pi \cdot d \cdot f}{c}\right)^2 = \left(\frac{4 \cdot \pi \cdot d}{\lambda}\right)^2 \tag{1.1}$$

where:

$FSPL$ = Free-Space Path Loss (mW)
d = distance of the receiver from the transmitter (metres)
f = signal frequency (Hertz)
λ = signal wavelength ($\lambda = c/f$ meters)
c = speed of light in a vacuum ($\sim 3 \cdot 10^8$ m/s)

Since the majority of measurements in this field are made in decibels (dB), it could be necessary to convert between mW and dBm; these conversions use the following formulas:

$$P_{dBm} = 10 \cdot log_{10}(P_{mW}) \tag{1.2}$$

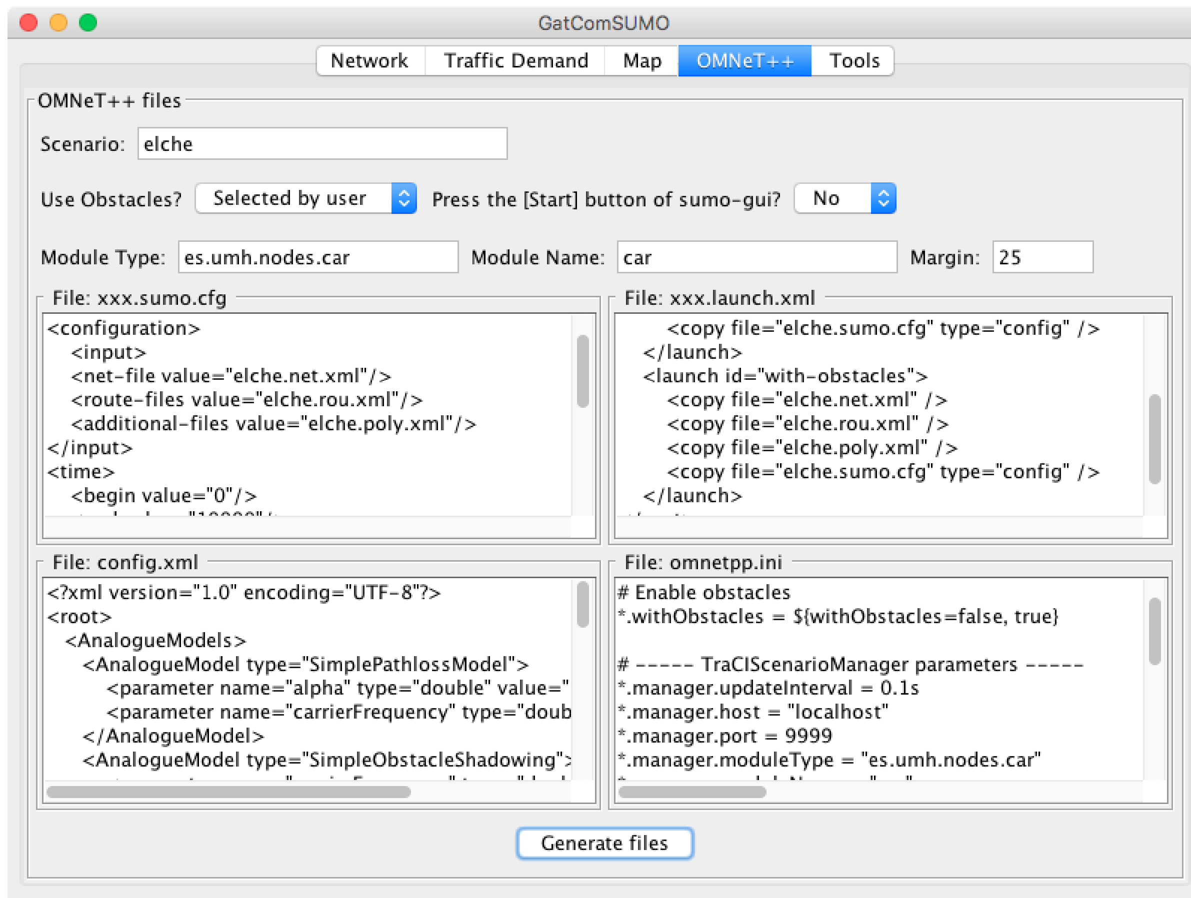


Figure 1.18: OMNeT++ simulation files

$$P_{mW} = 10^{(P_{dBm}/10)} \tag{1.3}$$

By applying (1.2) to (1.1) we can obtain another formula that calculates the FSPL in terms of dB:

$$FSPL(dB) = 20 \cdot log_{10}(d) + 20 \cdot log_{10}(f) - 147.56 \tag{1.4}$$

where:

$FSPL$ = Free-Space Path Loss (dB)
d = distance of the receiver from the transmitter (metres)
f = signal frequency (Hertz)

The final constant value depends on the units used for *d* and *f*; in this case the value -147.56 is needed for meters and Hz. If the distance or the frequency are expressed in other units, the constant value will be different. For example, for meters and GHz, such value is 32.44. GatcomSUMO allows to use any units for *d* (meters, kilometers) and *f* (Hz, KHz, MHz, GHz) by simply selecting them from the dropdown list, and the program computes the proper value accordingly.
In order to calculate the communication range it is necessary to know the power of reception. When a signal reaches the recipient with an intensity above a certain threshold, then it will be able to interpret the packets sent; otherwise, such packages are considered as noise and will be discarded. Signals can also can cause interference and errors in packets if they collide.
The Friis equation for computing the power of reception in linear units (not dB) is given by:

$$P_{rx} = P_{tx} \cdot G_{tx} \cdot G_{rx} \cdot \left(\frac{\lambda}{4 \cdot \pi \cdot d}\right)^2 \tag{1.5}$$

where:

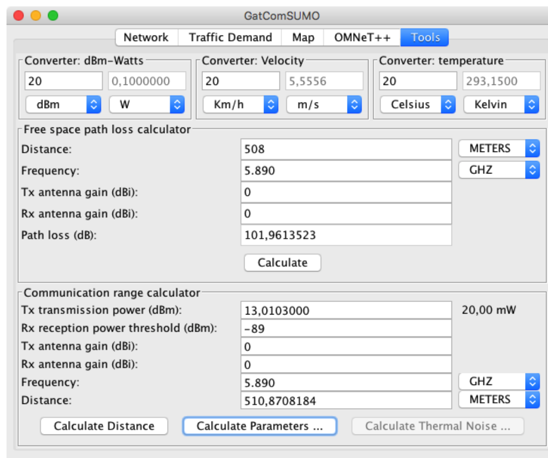


Figure 1.19: Tools

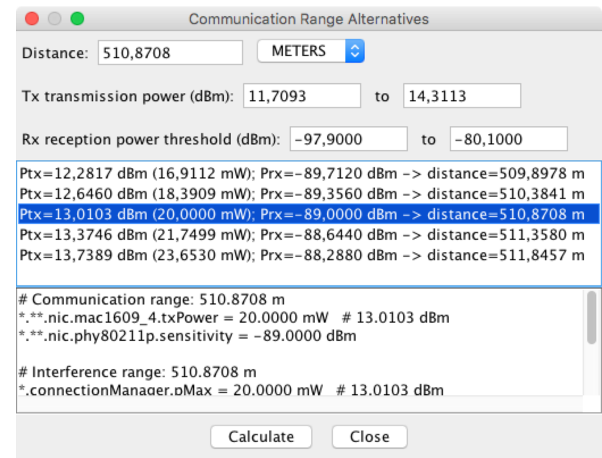


Figure 1.20: Communication range tool

P_{rx} = received power or sensitivity (W)
P_{tx} = transmitter output power (W)
G_{tx} = antenna gain of the transmitter (absolute)
G_{rx} = antenna gain of the receiver (absolute)
λ = signal wavelength ($\lambda = c/f$ meters)
c = speed of light in a vacuum ($\sim 3 \cdot 10^8$ m/s)
f = signal frequency (Hertz)

By solving the previous equation (1.5) for the distance (d), we get the equation for computing the communication range, as used in the `calcInterfDist()` function of the Veins `ConnectionManager` class:

$$d = \left(\frac{P_{tx} \cdot G_{tx} \cdot G_{rx} \cdot \lambda^2}{(4 \cdot \pi)^2 \cdot P_{rx}} \right)^{1/2} \tag{1.6}$$

As an example, the default parameter values in Veins are the following: P_{tx} = 13.0103 dBm ($\sim$ 0.02 W = 20 mW), P_{rx} = -89 dBm ($\sim$ 1.2589254E-12 W), G_{tx} = G_{rx} = 0 dBi ($\rightarrow 1$), and f = 5.890 GHz. With these values, the achieved communication range is: $d = 510.8708184\ m$ ($\sim 0.51\ Km$).

More interesting than the previous calculations is to answer the question of which values are needed for such parameters P_{tx} and P_{rx} in order to achieve a desired communication range. As there is not a single solution, it is necessary to vary each parameter with values ranging between a lower and an upper boundary. The distance is computed for each combination of values as shown before, and if the obtained distance is very close to the desired communication range, that case is added to the list of possible solutions. This is what GatcomSUMO does by just clicking on the [Calculate Parameters] button, which opens the dialog shown in Fig. 1.20. The user only has to select one of the results of the list, and the corresponding Veins parameters will appear with the calculated values in the bottom part, which can be directly copied and pasted to the OMNeT++ configuration file (omnetpp.ini). As can be seen, some parameters are expressed in mW and others in dBm; this is why the converter between W, mW, and dBm was added using the (1.2) and (1.3) equations. There is another converter for velocity units (km/h, m/s), useful for defining vehicle types. Finally, there is a converter between temperature units like Celsius degrees (°C), Fahrenheit degrees (°F), and Kelvin (K), which can be useful for computing the thermal noise, as the parameters need to be expressed in Kelvin.

1.7 Conclusions and Future Works

In this work, an application named GatcomSUMO has been presented, which allows to build the necessary simulation files when planning a simulation of VANETs using SUMO (network and traffic demand), OMNeT++, and Veins. The application generates all the necessary XML files in a user-friendly way in a matter of minutes, and without previous advanced knowledge about SUMO, OMNeT++ or OpenStreetMap. The application acts as a front-end, builds the proper commands according to the selected options in the graphical interface, and invokes the SUMO tools as a back-end. As a result, network files (.net.xml) and routes files (.rou.xml) are generated, among others. This ensures that the syntax and semantic of the files are correct, and error-free when running the simulations with the OMNeT++ network simulator and SUMO. The application can be used as an educational tool too, as the executed commands are shown, as well as the output and error messages. GatcomSUMO makes reproducibility of VANET simulations easier, because networks can be generated or downloaded from a brief description (e.g. coordinates of the bounding box), and the generated routes can be used as mobility traces for other experiments.
Future work includes to improve the map visualization, specifically to draw the different edges of a street, and lanes of each edge. As edges are unidirectional, it is very common that every street is defined with two edges, one for each direction, but currently, both are drawn one over the other. This is a problem when selecting the initial or final edge for a route by clicking on the map. In addition, an edge is drawn only as a poly-line, regardless the number of lanes contained in the streets. Regarding to the traffic demand, more generation models could be supported; currently, only defining routes by hand and randomly generated are supported. This work could be done by coding it into the application, or by invoking some SUMO utilities like `marouter`, `od2trips`, or `jtrrouter`. It might also be interesting to be able to randomly select the type of vehicles according to some probability value when routes and trips are generated; for the moment, it is possible to choose the vehicle type manually, and that type will be used for each vehicle. Nevertheless, the process can be repeated as many times as you want, with a different type each time.
Some of the tools included have been specially developed to be used with Veins, like the parameters obtained with the communication range utility, which can be simply copied and pasted from the application to the omnetpp.ini file. This utility could be generalized to show the parameters for other Wireless Networks frameworks like INET, MiXiM, Castalia, etc. Nevertheless, the values generated are still useful as they are shown currently, but the user has to write them in the proper parameters of the omnetpp.ini file.
GatcomSUMO can be downloaded from the Gatcom web site (`http://atc.umh.es/gatcom/soft/gatcomsumo`), both the source code and a multi-platform JAR file, although they are still under active development. In addition, the application has been compiled to Windows, Linux, and Mac OS X with the native compiler Excelsior JET, mainly for improving the performance during the visualization of the map.

Acknowledgments

This work was partially supported by the Spanish Government under Grant TIN2015-66972-C5-4-R, co-financed by FEDER funds (MINECO/FEDER/UE). The authors would like to thank to the developers of the open-source software used for our research, namely, SUMO (`dlr.de/ts/sumo`), OMNeT++ (`https://omnetpp.org`), and Veins (`http://veins.car2x.org`). We would also like to express our gratitude to the developers of the Java programming language and the Excelsior JET compiler (`https://www.excelsiorjet.com`).

References

[1] Barr, R., Z. J. Hass and R. van Renesse: *JiST/SWANS: Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator*. http://jist.ece.cornell.edu. [Accessed April 20, 2017].

[2] Chové, Etienne, Jocelyn Jaubert, Frédéric Rodrigo et al.: *Osmose*. http://osmose.openstreetmap.fr/es/map/. [Accessed April 20, 2017].

[3] Haklay, Mordechai (Muki) and Patrick Weber: *OpenStreetMap: User-Generated Street Maps*. IEEE Pervasive Computing, 7(4):12–18, October 2008.

[4] Härri, Jérôme, Marco Fiore, Fethi Filali and Christian Bonnet: *Vehicular Mobility Simulation with VanetMobiSim*. Simulation, 87(4):275–300, April 2011.

[5] Krajzewicz, Daniel, Laura Bieker, Jérôme Härri and Robbin Blokpoel: *Simulation of V2X applications with the iTETRIS system*. In *TRA 2012, Transport Research Arena, April 23-26, 2012, Athens, Greece / Also published in Procedia - Social and Behavorial Sciences, Volume 48, 2012*, Athens, GRÈCE, 04 2012.

[6] Krajzewicz, Daniel, Jakob Erdmann, Michael Behrisch and Laura Bieker: *Recent Development and Applications of SUMO - Simulation of Urban MObility*. International Journal On Advances in Systems and Measurements, 5(3&4):128–138, December 2012.

[7] Kurczveil, Tamás and Pablo Alvarez López: *eNetEditor: Rapid prototyping urban traffic scenarios for SUMO and evaluating their energy consumption*. In *SUMO 2015 - Intermodal Simulation for Intermodal Transport*, pages 137–160. Deutsches Zentrum für Luft und Raumfahrt e.V., May 2015.

[8] Lownes, Nicholas E. and Randy B. Machemehl: *VISSIM: A Multi-parameter Sensitivity Analysis*. In *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pages 1406–1413. Winter Simulation Conference, 2006.

[9] Lutnik, Hubert: *JGPSTrackEdit*. https://sourceforge.net/projects/jgpstrackedit/. [Accessed April 20, 2017].

[10] ns-2: *The Network Simulator*. http://www.isi.edu/nsnam/ns/. [Accessed April 20, 2017].

[11] Papaleondiou, L. G. and M. D. Dikaiakos: *TrafficModeler: A Graphical Tool for Programming Microscopic Traffic Simulators through High-Level Abstractions*. In *VTC Spring 2009 - IEEE 69th Vehicular Technology Conference*, pages 1–5, April 2009.

[12] Pediaditakis, Dimosthenis, Yuri Tselishchev and Athanassios Boulis: *Performance and scalability evaluation of the Castalia wireless sensor network simulator*. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, page 53. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010. [Accessed April 20, 2017].

[13] Piórkowski, M., M. Raya, A. Lezama Lugo, P. Papadimitratos, M. Grossglauser and J.-P. Hubaux: *TraNS: Realistic Joint Traffic and Network Simulator for VANETs*. SIGMOBILE Mob. Comput. Commun. Rev., 12(1):31–33, January 2008.

[14] Riley, George F. and Thomas R. Henderson: *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[15] Riverbed Technology: *Riverbed Modeler*. http://www.riverbed.com/products/performance-management-control/network-performance-management/network-simulation.html. [Accessed April 20, 2017].

[16] Scholz, Immanuel, Dirk Stöcker et al.: *JOSM*. https://josm.openstreetmap.de/. [Accessed April 20, 2017].

[17] Schünemann, Björn: *V2X simulation runtime infrastructure VSimRTI: An assessment tool to design smart traffic management systems*. Computer Networks, 55(14):3189 – 3198, 2011. Deploying vehicle-2-x communication.

[18] Schweizer, Joerg: *SUMOPy: An Advanced Simulation Suite for SUMO*. Lecture Notes in Computer Science (LNCS). Springer-Verlag Berlin Heidelberg, November 2014. [Accessed April 20, 2017].

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

[20] Uhrmacher, Adelinde M., Sally Brailsford, Jason Liu, Markus Rabe and Andreas Tolk: *Reproducible Research in Discrete Event Simulation: A Must or Rather a Maybe?* In *Proceedings of the 2016 Winter Simulation Conference*, WSC '16, pages 1301–1315, Piscataway, NJ, USA, 2016. IEEE Press.

[21] Varga, András et al.: *INET Framework*. https://inet.omnetpp.org. [Accessed April 20, 2017].

[22] Varga, András and Rudolf Hornig: *An Overview of the OMNeT++ Simulation Environment*. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[23] Wegener, Axel, Michał Piórkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer and Jean-Pierre Hubaux: *TraCI: An Interface for Coupling Road Traffic and Network Simulators*. In *Proceedings of the 11th Communications and Networking Simulation Symposium*, CNS '08, pages 155–163, New York, NY, USA, 2008. ACM.

[24] Wessel, Karl, Michael Swigulski, Andreas Köpke and Daniel Willkomm: *MiXiM: the physical layer an architecture overview*. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).