Chapter 6
Veins: The Open Source Vehicular Network Simulation Framework

Christoph Sommer, David Eckhoff, Alexander Brummer, Dominik S. Buse, Florian Hagenauer, Stefan Joerer, and Michele Segata

6.1 Introduction

Veins [56] is a model library for (and a toolbox around) OMNeT++, which supports researchers conducting simulations involving communicating road vehicles; either as the main focus of a study (such as Vehicular Ad Hoc Networks - VANETs) or as a component (such as in Intelligent Transportation Systems - ITS). It is distributed as open-source software; as such, it is free to download, adapt, and use.

The model library includes a full stack of simulation models for investigating communicating vehicles and infrastructure; as of Veins 4.7, predominantly cars and trucks using Wireless Local Area Network (WLAN)-based technologies. For this,

C. Sommer (✉) · D. S. Buse · F. Hagenauer
Heinz Nixdorf Institute and Department of Computer Science, Paderborn University, Paderborn, Germany
e-mail: sommer@ccs-labs.org; buse@ccs-labs.org; hagenauer@ccs-labs.org

D. Eckhoff (✉)
TUMCREATE Ltd, Singapore, Singapore
e-mail: david.eckhoff@tum-create.edu.sg

A. Brummer
Computer Networks and Communication Systems, University of Erlangen-Nürnberg, Erlangen, Germany
e-mail: alexander.brummer@fau.de

S. Joerer
Institute of Computer Science, University of Innsbruck, Innsbruck, Austria
e-mail: joerer@ccs-labs.org

M. Segata
Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
e-mail: msegata@disi.unitn.it

A. Virdis, M. Kirsche (eds.), *Recent Advances in Network Simulation*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-12842-5_6

Veins includes a sophisticated model of IEEE 802.11 MAC layer components [12] used by standards such as IEEE Wireless Access in Vehicular Environments (WAVE) (of which a simple simulation model is included), ETSI ITS-G5 (as provided by, e.g., Artery [43] which is described in Chap. 12), or ARIB T-109 [23]. Because Veins is a modular framework, it can equally well be used as the basis for modeling other mobile nodes such as pedestrians, bikes, trains, and Unmanned Aerial Vehicles (UAVs)—or for other communication technologies like Long Term Evolution (LTE) mobile broadband [21] (cf. Sect. 6.4.1) and Visible Light Communication (VLC) [37].

The history of Veins goes back to early 2006 with the first public release being an extension for the INET Framework version 2006-10-20. Because of limitations in the fidelity of wireless channel modeling at the time, for its 1.0 release Veins was ported to be an extension of MiXiM (an alternative, now discontinued library of OMNeT++ simulation models for wireless channel modeling) instead. Veins was then increasingly augmented with new models, e.g., of IEEE 802.11p, IEEE 1609.4, and WAVE, which would later be re-factored all the way down to the physical layer for the 2.0 release. As more refactoring and rewriting was taking place in the channel models, Veins 3.0 became a proper fork of MiXiM, but was kept compatible with mixed simulations incorporating models from the INET Framework. Up to the current 4.7 release, Veins was then continuously streamlined and augmented with more and more of the aforementioned models specific to communicating road vehicles. This release is compatible with OMNeT++ 5 (up to the current version 5.4.1) and SUMO 0.32.0 (the latest release of SUMO; please refer to Sect. 6.2.1 for details on its role for Veins). A full compatibility list is available online.[1]

Veins has become well-established in the domain of Vehicular Ad Hoc Networks (VANETs) and Intelligent Transportation System (ITS). It is employed by both academia and industry around the globe. It serves as the basis of hundreds of publications and contributed to the standardization process of Inter-Vehicle Communication (IVC). Common fields of application include channel access control [17, 58], safety applications [26, 57], privacy [14] and security [44], platooning [49], communication with traffic lights [16], electric vehicle operation [3], as well as traffic optimization [65]. For some of these uses cases there exist dedicated extensions for Veins such as PREXT for location privacy [19], PLEXE for platooning [48], an extension to incorporate a real world driving simulator [2], or a simulation framework for electric vehicles [3].

In this chapter, we give a brief overview of recent developments regarding the internals of Veins (bi-directional coupling, communication stack, antenna characteristics, unit testing, and timer management; in Sect. 6.2), present two practical use cases (platooning and intersection collision avoidance; in Sect. 6.3), and conclude with a brief discussion of two extensions (Veins LTE and Veins_INET) as well as using Veins as a virtual appliance (cf. Sect. 6.4).

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

6.2 Internals

In this section, we explain how the bi-directional coupling works (cf. Sect. 6.2.1) and give details on the implementation of the IEEE 802.11p-based communication stack (cf. Sect. 6.2.2). Discussion on Veins internals continues with the modeling of antenna characteristics (cf. Sect. 6.2.3), followed by a section on how unit testing can help in the development of new simulation models (cf. Sect. 6.2.4), and simplified timer management (cf. Sect. 6.2.5).

6.2.1 Architecture and Bidirectional Coupling

Contrary to what might be expected, Veins does not include custom mobility models of road vehicles. Rather, it has simulations establish a connection to a dedicated road traffic simulator which is running as a separate process, as illustrated in Fig. 6.1. This way, Veins can benefit from the years of research and development by domain experts who have created fully featured tools for road traffic simulation. The road traffic simulator that Veins was designed to interoperate with is Simulation of Urban MObility (SUMO)[2] (though, in theory, any simulator supporting the Traffic Control Interface (TraCI) simulator coupling interface can be used).

SUMO can simulate medium to large road networks of cities, urban areas, highways, and freeways. On those, it can simulate the movement of road vehicles

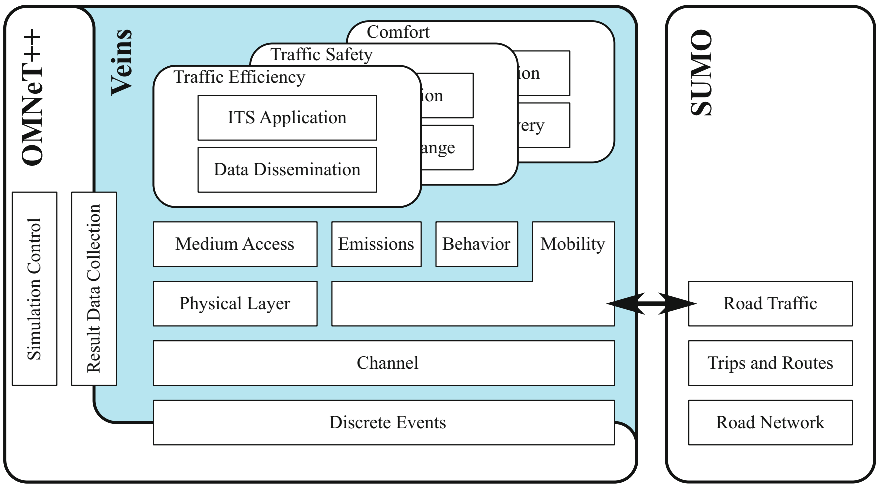


Fig. 6.1 High-level architecture of Veins

[2]SUMO website: http://sumo.dlr.de/.

like cars and trucks, of scooters and bicycles, of pedestrians, and of trains. SUMO supports a wide range of different mobility models (from idealized, lane-discrete models to sub-lane models of mixed car/scooter traffic), a set of different intersection controllers (from simple right of way to demand-actuated traffic lights), and a wide range of road network input formats (from **OpenStreetMap** and **TIGER** to proprietary, specialist Geographic Information System (GIS) formats).

By default, mobility information is polled from SUMO at fixed intervals of, e.g., 100 ms, though adaptive polling is equally well supported by the interface. Execution of the OMNeT++ simulation pauses while SUMO computes mobility information for the desired point in time. The performance impact of this is, however, minimal as SUMO is designed to simulate at least an order of magnitude more mobile nodes than can be afforded in a highly detailed wireless network simulation. As a consequence, in a reasonably complex wireless network simulation, only fractions of percent of simulation time are spent calculating and communicating mobility information.

Whenever SUMO simulates the departure of a mobile node, Veins creates a dedicated simulation module in OMNeT++. Then, as the mobile node moves in SUMO, Veins keeps the corresponding OMNeT++ module updated wrt its position, heading, and speed (along with the status of turn signal indicators and similar miscellaneous data). Similarly, when SUMO simulates the mobile node arriving at its destination, Veins removes the corresponding OMNeT++ module from the simulation. This way, Veins couples node mobility in OMNeT++ to that in SUMO.

This coupling is bi-directional: in addition to the OMNeT++ simulation evolving as dictated by the SUMO simulation, the OMNeT++ simulation can influence the simulated road traffic in SUMO, for example, to have cars choose a different route to their destination in response to received traffic information—or to have a car perform an emergency brake in response to received warnings. This is done by calling methods of the `TraCICommandInterface` and component class instances associated with each mobile node. They are available by obtaining a pointer to the mobility module via `TraCIMobilityAccess().get(getParentModule())` and calling its `getCommandInterface()` or `getVehicleCommandInterface()` method from any simulation model of a mobile node that contains a mobility model of type `TraCIMobility` (a requirement for mobile nodes managed by Veins). These interfaces offer a wealth of methods—from the simple, like `getRoadId` and `newRoute` (for vehicles) to the complex, like `setProgramDefinition` (for a traffic light). Details on this concept are available in the literature [56].

Programmatically, the coupling is performed by instantiating a simulation module of type `TraCIScenarioManager`, which bi-directionally couples OMNeT++ and SUMO. However, the user needs to manually run one SUMO simulation for every OMNeT++ simulation. As an alternative and to ease the management of two simulators running in parallel, Veins also includes tools to automatically set up and run SUMO simulations. This is done by instantiating a subclass of `TraCIScenarioManager` called `TraCIScenarioManagerLaunchd`. It expects the user to have run a command line utility, **sumo-launchd.py**, which waits for incoming network connections from an OMNeT++ simulation and launches one instance of SUMO for each simulation and proxies the connection. Alternatively, another subclass called `TraCIScenarioManagerForker` can be employed, which will directly run a local instance of SUMO when needed. All of these coupling variants are included with Veins.

What is not included with Veins are road traffic scenarios to generate the SUMO traffic from. While, these days, road network data and building positions are easy to come by (thanks to open data sources), information about traffic demand (that is, how typical traffic moves through the road network), traffic light timings, or meta-data like bus and train schedules are much harder to obtain. In the early days of VANET simulation, road traffic scenarios were thus often generated synthetically, e.g., modeling an ideal Manhattan Grid of roads. This had the obvious downside of requiring a lot of skill on the part of the researcher generating the road traffic scenario (lest the simulation test the system under study in unrealistic conditions).

A better choice is to pick one of the well-tested road traffic scenarios that have been made available recently. Examples for the SUMO road traffic simulator are:

- The Bologna "Pasubia" and "Acosta" scenarios [6], depicted in Fig. 6.2a, feature 9k trips each on two areas of 2 km × 1 km each.[3] They can be run individually or as one bigger road traffic scenario and feature traffic driving in a small part of the city core of Bologna, though care must be taken as no building positions are included with the scenario.
- The Bologna "Ringway" scenario [4], depicted in Fig. 6.2b, features 22k trips on an area of 4 km × 3 km.[4] It focuses on road traffic on an arterial road running around a city center. Like the Pasubia and Acosta scenarios, no building positions are included with the scenario.
- The Luxembourg "LuST" scenario [10], depicted in Fig. 6.2c, features 288k trips on an area of 14 km × 11 km.[5] It is the largest and most complete scenario to date and includes a full day of mobility data for a complete city, including the positions of buildings and parking lots.
- The Monaco "MoST" scenario [9] in Fig. 6.2d includes 18k trips in an area of 10 km × 7 km.[6] Still under development, it focuses on multi-modal traffic, comprising added information regarding public transport, bicycles, and pedestrians.

[3]http://sourceforge.net/projects/sumo/files/traffic_data/scenarios/Bologna_small.

[4]http://www.cs.unibo.it/projects/bolognaringway/.

[5]https://github.com/lcodeca/LuSTScenario.

[6]https://github.com/lcodeca/MoSTScenario.

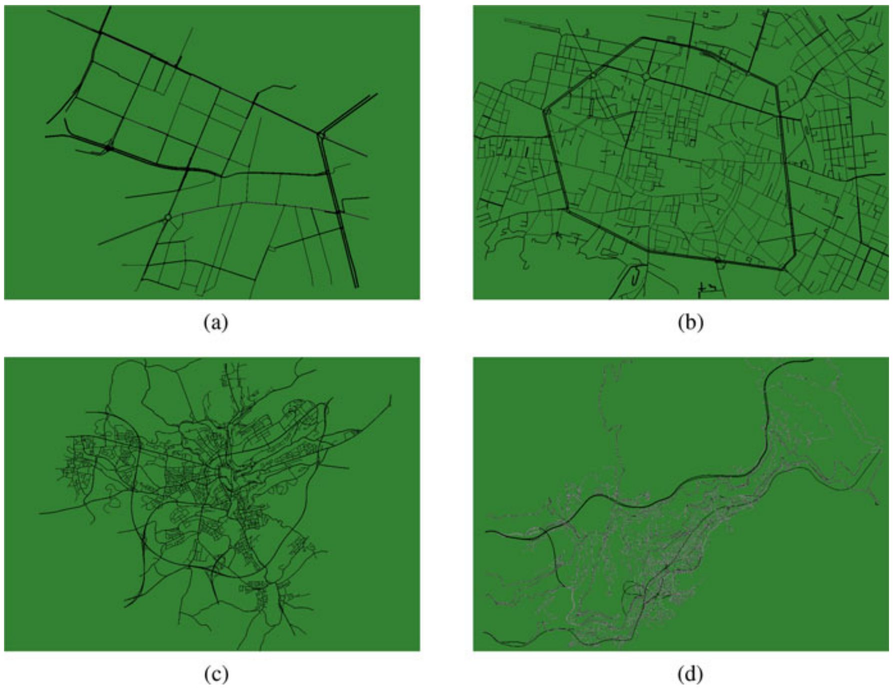


Fig. 6.2 Selection of existing openly available scenarios for SUMO. (**a**) Bologna: Pasubia and Acosta. (**b**) Bologna: Ringway. (**c**) Luxembourg: LuST. (**d**) Monaco: MoST

6.2.2 The MAC and PHY Layer

One of the core features of Veins is the detailed modeling of the lower layers of Inter-Vehicle Communication (IVC). For the evaluation of most IVC applications and networks, a detailed packet-level simulation using accurate models of the evaluated technology is required [15]. For vehicular networks, the technology in question is often IEEE WAVE (or ETSI ITS-G5 in Europe). The core of this family of standards is the IEEE 1609.4 multi-channel operation using the IEEE 802.11p Medium Access Control (MAC) and Physical Layer (PHY). An overview of the stack is given in Fig. 6.3a. While it is possible to implement and integrate each of these layers and standards, Veins puts a focus on the lower layers as these are decisive for the actual channel access and transmission of packets [17]. Other simulation models (not included with Veins, but publicly available, such as ARIB T-109 [23]) can build on this foundation if additional protocol layers of the various protocol stacks of ITS protocols around the world are to be modeled as well.

Figure 6.3b shows the representation of the stack within Veins. Each node, be it a vehicle, a road-side unit, or even a pedestrian or cyclist making use of wireless communications would need to consist of at least an 802.11p Network Interface

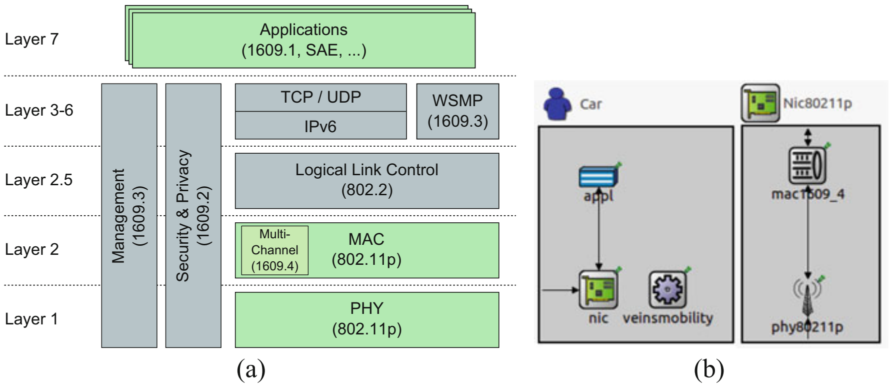


Fig. 6.3 The IEEE WAVE stack and its representation in Veins. (**a**) The IEEE WAVE family of standards. PHY, MAC, and application layer are represented in Veins. (**b**) Layer representation in OMNeT++

Card (NIC) to be able to communicate with other devices. Higher layers (in some stacks: the application layer) are directly connected to this NIC which itself is a compound model consisting of the MAC and the PHY layer. This results in a simple APP-MAC-PHY architecture for each node in Veins. The `veinsmobility` module is responsible for updating the position of the vehicle (see Sect. 6.2.1). In the case of a road-side unit, the mobility would be a constant `BaseMobility`.

In OMNeT++, each module can exchange messages with other modules if they are connected. These messages can be of any type inheriting from `cMessage*`, that is, just plain messages or (encapsulated) packets of any given message format (e.g., Wave Short Messages (WSMs) or Wave Service Advertisements (WSAs)). Inside a node, messages can either be "normal" messages that might be forwarded to layers above or below or control messages to trigger a certain action in the receiving layer. Depending on the type, a different function will be called in the receiving layer. As can be seen in the figure, the physical layer is connected only to the MAC layer and to the outside world.

In the following subsections, we will discuss how messages are generated, processed, forwarded, and received.

6.2.2.1 Medium Access Control and Upper Layers

The Medium Access Control (MAC) layer in the simulation should represent the simulated system as closely as possible (for example, evaluating an IEEE 802.11p system for IVC using a model for IEEE 802.11b can give misleading or even wrong results) [15, 27]. Veins comes with a detailed IEEE 1609.4 and IEEE 802.11p MAC layer that supports multi-channel operation, channel switching (alternate access), transmission of unicast and broadcast messages, and an IEEE 802.11e Enhanced Distributed Channel Access (EDCA) implementation with four different access categories [12]. For a detailed description we refer the interested reader to IEEE 802.11e and IEEE 802.11p [12, 15, 53] and the actual standardization documents [25] and [24]. The level of detail in Veins' MAC and PHY layer implementation allows researchers to conduct various simulation studies, for instance comparing wireless network performance [17, 58], studying the applicability of the wireless network for vehicular cooperative safety [26] (cf. Sect. 6.3.2), or the simulative analysis of platoons [48] (cf. Sect. 6.3.1).

The implementation in Veins follows a different paradigm compared to most other OMNeT++ frameworks. The behavior of the MAC layer can be specified in the form of a state machine, which is a useful method to understand as well as implement the system. Transitions between states are triggered after, e.g., a timeout has expired, the backoff counter has reached zero, a packet arrived, and so on. Indeed, an implementation might choose to directly follow the state diagram. However, the MAC layer has several properties which make such an implementation hard to maintain and read: packets can arrive from an upper layer regardless of the state the MAC layer is in, multiple timers can run in parallel (e.g., for each of the EDCA queues as well as the channel switching time), and the multi-channel operation would require two independent state machines. Not only does this lead to plenty of nested `if` statements in each function (to check which state the system is currently in) which makes extending and understanding the code base challenging, it also has an impact on performance as multiple timers have to be managed in parallel, i.e., inserted into and removed from the event queue.

This prompted the design decision to not rely on a state machine implementation but follow a different, more efficient approach: The MAC layer always tracks the time at which it can send the *next* packet, instead of tracking all the different intervals such as interframe spaces or backoff times, separately. When an event occurs that affects this time, e.g., the channel turns busy or a new packet in a higher priority queue arrives from the upper layer, the timer is canceled or rescheduled and the backoff counters for each EDCA queue are updated. When the channel turns idle again, the time is recomputed and the timer is scheduled again. The result of this design is that Veins will only use one single timer (`nextMacEvent`) when using a single channel MAC layer with broadcast messages only, which is a rather common setup for vehicular networks. Multi-channel operation and unicast packets require additional timers.

Transmitting a Packet The MAC layer expects a `WaveShortMessage` from higher layers (e.g., the application layer) with attached information on which channel it should be sent and a user priority which will be mapped to an EDCA queue. The packet will be queued accordingly and the `nextMacEvent` timer will be updated if necessary. If the channel is busy and the respective EDCA queue has a backoff counter of 0 with the newly arrived packet at the front of the queue, then a backoff procedure is invoked according to the standard.

The core of the MAC layer is the `startContent` function which models the start of contention for the channel and returns the time the next packet can be sent. It iterates through each of the EDCA queues and computes this time based on the queue-specific interframe time (AIFSn $\times$ slot length $+$ SIFS), the current backoff counter, and the last time the channel went idle. If the channel was idle long enough when a new packet arrives from the upper layer, the packet will be sent at the next slot boundary. When the timer expires, the MAC layer sets the channel to busy and calls the `stopContent` function. In this function, the backoff counters of the remaining EDCA queues are updated and Transmit Opportunitys (TXOPs) for ready-to-transmit queues are generated. Then `initiateTransmit` function is invoked which is responsible for returning the actual packet that is supposed to be sent. In the case of an internal collision, that is, when there are two or more packets ready, the lower priority queues will be sent into backoff. The winning packet will be encapsulated with the corresponding MAC header and `controlInfo` (containing transmit power and data rates), and if there is enough time left in the current control or service channel interval, handed to the PHY layer. The `stopContent` function is also invoked when the channel turns busy due to an external transmission. In this case, no TXOPs are generated and the `nextMacEvent` timer is canceled.

Receiving a Packet The role of the MAC layer in the reception of a packet is straightforward. If the PHY layer sends up a `Mac80211Pkt`, the MAC will check whether the destination address is the layer 2 broadcast address or whether it matches its own MAC address. If this is the case, the packet will be decapsulated and the `WaveShortMessage` will be handed to the application layer. When dealing with unicast transmissions, the received packet can be an Acknowledgment (ACK) packet. The reception of an ACK packet marks the successful transmission of a unicast packet, causing the MAC layer to remove it from the respective EDCA queue. If the MAC layer is expecting an ACK packet but has received another packet, then the originally sent packet has to be retransmitted.

The PHY layer also informs the MAC layer of several other events such as successful or unsuccessful reception of a packet, the channel turning busy or idle, erroneous decoding of a packet, and so on. This is achieved by means of control messages. Veins collects various statistics about received packets and failures (split by broadcast and unicast and by cause of loss) as well as about internals of the state machine (e.g., how busy the channel was), giving the researcher methods to evaluate the underlying network in great detail.

6.2.2.2 The Physical Layer and the Wireless Channel

The benefit of packet level simulation is the capability to (more or less) realistically determine for each packet if it can be successfully received. There are several factors affecting the decoding of a packet: the position in space of sender and receiver, the antenna characteristics (see Sect. 6.2.3), whether there is an obstacle blocking the line of sight, and interference from other transmitting nodes. While Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) significantly reduces the chance of two nearby nodes (i.e., they can hear each other) sending at the same time, it does not offer a solution to the hidden terminal problem [22]. All these effects can be captured by Veins. In this section, we will outline the functionality of the PHY.

The described models of the Physical Layer and the wireless channel in Veins are currently based on a fork of MiXiM [64], a discontinued framework which models radio signals as generic n-dimensional objects (power levels expressed in, e.g., time and frequency) and provides a math toolbox to work with them.

It should be noted that, while this is the most flexible way of modeling radio signals, it is also computationally expensive. Thus, Veins has been updated to use a more specific abstraction of radio signals, tailored to the feature set used in common Vehicle-to-Vehicle (V2V) communication (e.g., forcing any radio signal to always have a time and a frequency dimension—never more, never less) and optimized for efficiency. While this has the obvious drawback of not being able to model radio signals in dimensions other than time and space, these adaptations can allow simulations to run faster—in some cases up to two orders of magnitude. While this functionality is not yet available in Veins 4.7, it is available on Github and will be integrated into upcoming releases of Veins.

Analogue Models The connection manager of OMNeT++ maintains a connectivity map to be able to hand transmitted messages to the receiving nodes. Every node inside a configurable *interference range* of a transmitting node will be handed a copy of the transmitted packet. Determining whether this packet is successfully received then lies within the responsibility of the node itself. The setting of an *interference range* is purely an optimization: it defines an artificial range beyond which no radio transmission needs to be considered as interfering. Naturally, it should be set much larger than the maximum range of any successful transmission, as also packets that have a too low receive power (or Received Signal Strength (RSS)) to be decoded can still affect the successful reception of other packets.

The connection manager will hand an airframe at least twice to the PHY layer via the `handleMessage` function: when the receiving starts and when it ends. The first thing that has to be computed is the actual receive power of the frame as this determines whether the channel turns busy or remains idle. This is done by applying the antenna gains (G_t, G_r) and iteratively applying all the configured loss models L in the `filterSignal` function (see Eq. (6.1)).

$$P_r = P_t + G_t + G_r - \sum L \tag{6.1}$$

Common deterministic loss models include the free space path-loss model and the two-ray interference path-loss model [52] that considers the reflected signal from the road that can cause cancellation and amplification of the received signal. A detailed explanation of these models can be found in [13]. To account for fast fading effects, Veins can make use of *Nakagami-m* fading which is a probabilistic method to reflect multi-path propagation in urban environments [59].

The effect of obstacles (e.g., buildings in the scenario description file) is also accounted for by the use of a loss model. Assuming each obstacle is a polygon, then the receive power is reduced based on the number of edges n the signal is intersecting (e.g., walls) and the distance m covered inside of polygons (e.g., inside a building). These values are weighted using parameters β and γ which were calibrated using real world measurements (see Eq. (6.2)). They can be changed according to the material of the obstacle, e.g., brick, concrete, etc.

$$L_{\text{build}} = \beta \cdot n + \gamma \cdot m \tag{6.2}$$

Listing 6.1 shows how to configure a simple chain of analogue models (an XML configuration set as the physical layer's `analogueModels` parameter). In this configuration, each received signal is first passed through a free-space path loss model, then through an obstacle shadowing model.

A comparison of the different models as well as their ability to reproduce real-world measurements [55] is given in Fig. 6.4.

Listing 6.1 Content of the *config.xml* file (part one)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <AnalogueModels>
  <AnalogueModel type="SimplePathlossModel">
   <parameter name="alpha" type="double" value="2.0"/>
   <parameter name="carrierFrequency" type="double" value="5.890e+9"/>
  </AnalogueModel>
  <AnalogueModel type="SimpleObstacleShadowing">
   <parameter name="carrierFrequency" type="double" value="5.890e+9"/>
  </AnalogueModel>
 </AnalogueModels>
</root>
```

The Decider Once all loss models have been applied, the airframe is handed to the `Decider` which is an outsourced class that determines whether packets can be successfully decoded. If the received power is below the configurable Clear Channel

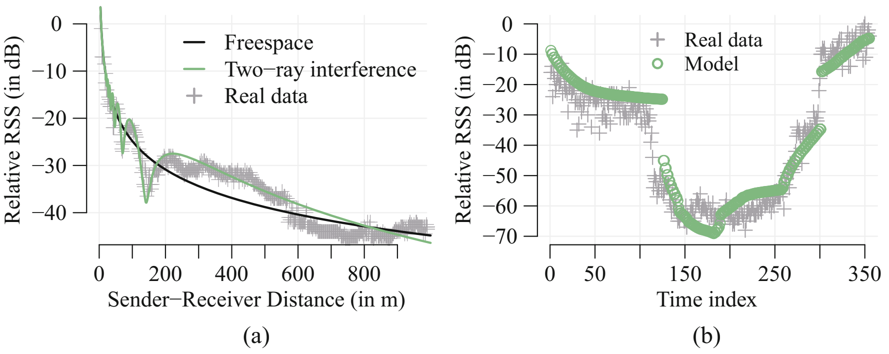


Fig. 6.4 Analogue models and their effect on the Received Signal Strength (RSS) compared to real-world measurements. (**a**) Real-world measurements compared to the free-space model and the two-ray interference model based on [52]. (**b**) Real-world measurements compared to the obstacle model in Eq. (6.2) based on [55]

Assessment (CCA) sensitivity, this packet is unable to set the channel to busy. The MAC layer will not be notified. If the packet is above the CCA threshold, the decider checks whether the node is already transmitting or receiving another packet. In both cases the packet will fail to decode.

The `processSignalEnd` function in the decider is called when the connection manager hands the airframe to the physical layer the last time. It is the task of the decider to finally determine whether the packet is decodable. To this end, it first has to compute the Signal-to-Interference-plus-Noise-Ratio (SINR) as depicted in Eq. (6.3): the receive power of the packet in question i is divided by the power of all interfering packets j and the background noise N.

$$\mathrm{SINR}(i) = \frac{P_i}{N + \sum_{i \neq j} P_j}; \tag{6.3}$$

Once the SINR has been obtained, it can be fed to a bit error model. Depending on the modulation scheme (e.g., Binary Phase Shift Keying (BPSK), Quadrature Phase Shift Keying (QPSK), Quadrature Amplitude Modulation (QAM)), a different equation is applied to calculate the probability of one bit being decoded erroneously. Bit error rates for header and payload are computed separately and then applied to the packet length to derive a packet error rate. Two randomly drawn numbers then decide whether the header and the payload can be decoded successfully.

The packet is handed to the MAC layer or, in the case of an error, a control message is sent. Listing 6.2 shows how to configure the decider model (an XML configuration set as the physical layer's `decider` parameter). Each received signal is then passed through this chain of models.

Listing 6.2 Content of the *config.xml* file (part two)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Decider type="Decider80211p">
    <parameter name="centerFrequency" type="double" value="5.890e9"/>
  </Decider>
</root>
```

6.2.3 *Modeling Antenna Patterns*

Antennas are an integral part of wireless communications as they constitute the interface between the radio device and the transmission medium. Yet, despite the multitude of detailed models for the PHY and MAC layers described before, the impact of antenna patterns has not been taken into account in VANET simulation for a long time—even though the gain (or loss) of an antenna can critically influence the receive power and thus the decodability of a sent message. This dependence is already indicated by Eq. (6.1) (on page 224) with the terms G_t and G_r being related to the sender's and receiver's antenna gain, respectively.

Early work on the impact of antenna patterns on V2V communication [18] demonstrated that the vast majority of messages were received either from the front or from the rear direction of the vehicle. It also demonstrated that, as a consequence, the overall number of received beacons in a typical cooperative awareness simulation was decreased by up to 20% compared to simulations neglecting antenna influence (i.e., assuming isotropic radiators).

A highly configurable model for the consideration of antenna patterns has been added to the Veins framework as of version 4.5.

The power of the sent or received signal depends on several aspects, first of all the type of antenna in use. In the case of an ideal, isotropic antenna, the transmit power is radiated in all directions equally. Omnidirectional antennas, e.g., monopoles, emit the signal power equally in a certain plane. Another category are highly directional antennas, which concentrate the power in one or a few selected directions. These differences in power are usually stated as a dBi value, that is, on a logarithmic scale with respect to an ideal, isotropic radiator.

In the context of vehicular networks, radiation characteristics are further influenced by the vehicles themselves. An important factor is the mounting location, which might be on the roof, at the front, at the rear, on the side mirrors, or even under the car. Example patterns as measured in [31, 34] are depicted in Fig. 6.5. For example, the radiation pattern of a vehicle with patch antennas on the side mirrors (see Fig. 6.5c) exhibits a substantial prevalence towards the front of the car. Moreover, material properties of parts surrounding the antenna can influence the power of a transmitted or received signal. A distinct example is shown in Fig. 6.5a. This radiation pattern is the result of a study by Kwoczek et al. [34] who investigated the consequences of an antenna being mounted next to a panorama glass roof. As can be seen, this leads to a substantial attenuation of up to 20 dBi towards the front of the vehicle as the signal tends to get reflected within the glass roof.

It is quite obvious that such an influence on the signal power can make all the difference in simulation when deciding on the decodability of a packet, which is why the support for antenna patterns has been added to the Veins framework. For this purpose, an object of the newly introduced `Antenna` class is assigned to every vehicle (or more generally: to every module containing a radio). For this, an `Antenna` member is added to the `BasePhyLayer` class, which itself is present in every module capable of wireless communication (see Fig. 6.6). The `Antenna` class can be seen as the superclass for all kinds of specialized antenna implementations and simply returns a factor of 1.0 (representing an isotropic pattern).

This approach facilitates the implementation of various antenna subclasses that differ in the way of computing the specific gain. The subclass capturing one of the most common use cases is `SampledAntenna1D`, which deals with two-dimensional antenna patterns, i.e., only the horizontal plane is considered. In this case the resulting gain depends on one variable, namely the signal's horizontal angle of incidence. The user needs to pick a representative antenna from the included

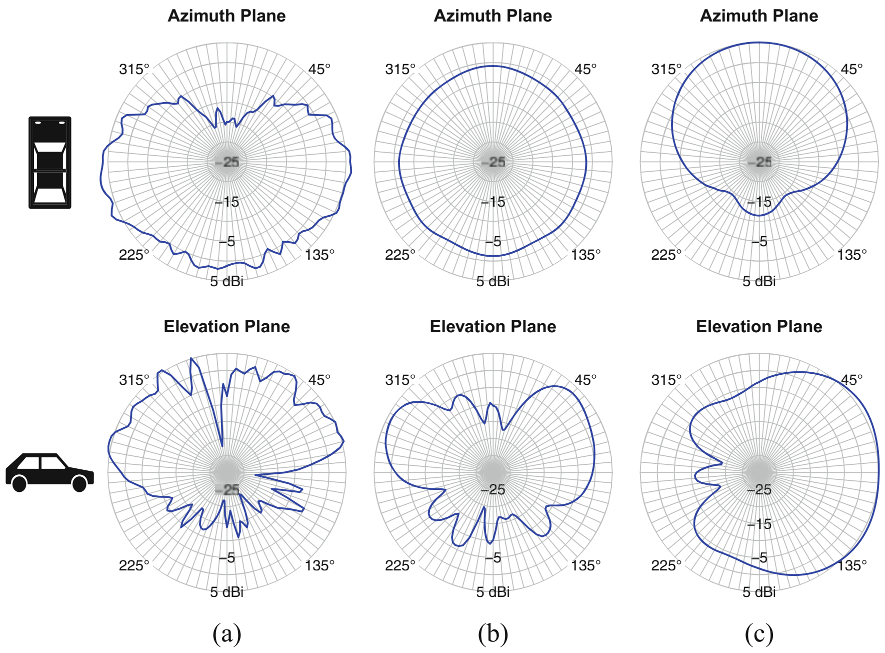


Fig. 6.5 Azimuth and elevation planes of example vehicular antenna patterns (gain in dBi). (**a**) Monopole antenna on glass roof (based on [34]). (**b**) Monopole antenna (based on [31]). (**c**) Patch antenna (based on [31])

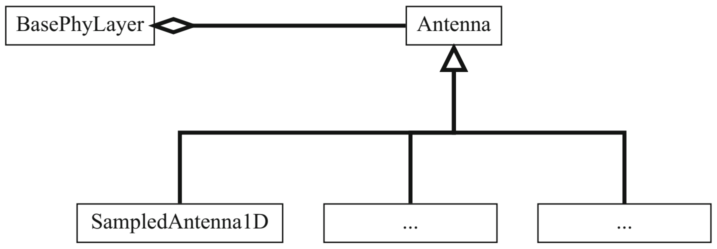


Fig. 6.6 Overview of the newly added antenna classes

database (or provide samples of the radiation pattern at equidistant angles between 0° and 360°).

For the actual gain calculation, the signal direction has to be determined first. As illustrated in Fig. 6.7, this angle of incidence ϕ (also called azimuth angle) depends on the sender's and receiver's position as well as on the orientation of the antenna in question. As all of these parameters are known to the simulation, the azimuth angle ϕ can be determined with the help of the scalar product. Next, the stored antenna gain samples are queried at the determined angle. If the angle of the required gain

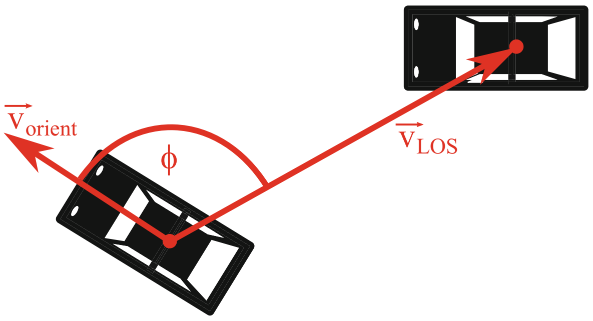


Fig. 6.7 Dependence of the azimuth angle based on Line of Sight (LOS) and orientation vector

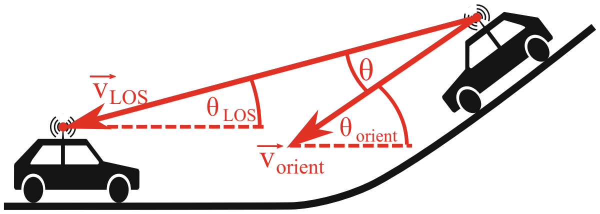


Fig. 6.8 Dependence of the elevation angle based on Line of Sight (LOS) and orientation vector

value is located between two samples, linear interpolation is applied. Finally, the signal power is multiplied by the determined antenna gain factor.

Recent work also examined the influence of 3D antenna patterns in a three-dimensional environment [8]. To this end, another antenna class has been implemented, namely `SampledAntenna2D`. As the name implies, the antenna gain is now dependent on two parameters. Besides the already introduced horizontal (azimuth) angle ϕ, the vertical (elevation) angle θ needs to be determined as well. It can be computed based on the (now three-dimensional) antenna positions and orientation of the ego vehicle (see Fig. 6.8). Only if both angles are known is it possible to specify the signal direction in the three-dimensional space.

Obviously, the user has to provide a 3D antenna pattern in the first place. As a full representation is rarely available and would imply a large number of samples, only the two principal planes are required for our model. The azimuth plane pattern has already been used for the 2D antenna implementation. In addition, the elevation plane pattern becomes necessary. Again, equidistant samples of both patterns of the antenna type to simulate need to be provided by the user. In order to estimate the antenna gain in an arbitrary direction, the 3D antenna pattern interpolation method described by Leonor et al. [36] is applied. It is based on determining the four closest gain values on the principal planes and summing them up weighted proportionally to their contribution. This way, the three-dimensional antenna gain in the required direction can be estimated.

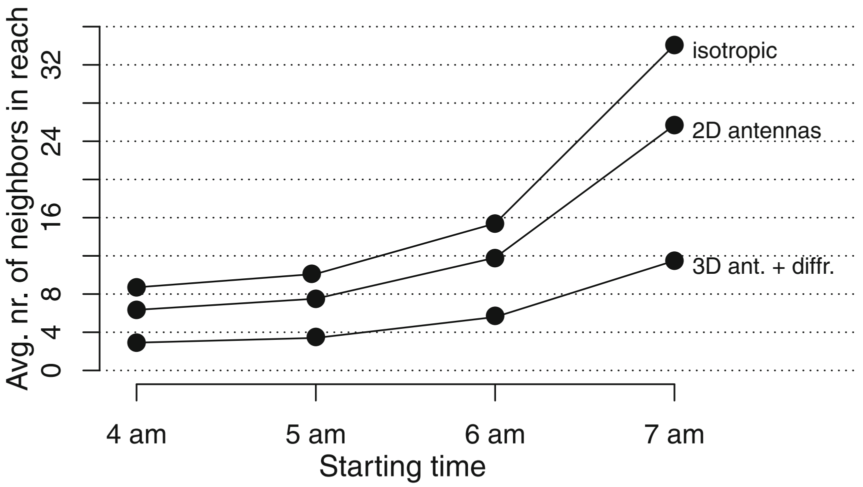


Fig. 6.9 Average number of neighbors in reach when simulating the LuST scenario with and without 2D antenna patterns as well as 3D antenna patterns (including diffraction effects) [8]

As a matter of course, the assignment of 3D antenna patterns only makes sense if the whole environment of the scenario under investigation itself is modeled in a three-dimensional way. This means that the underlying road network has to include z-coordinates and that this additional 3D data also has to be exchanged between SUMO and OMNeT++, where it can be used for the three-dimensional antenna model.

Note, however, that 3D antenna patterns are not the only aspect that needs to be considered for a sufficient three-dimensional simulation of VANET scenarios: Brummer et al. [8] demonstrate that diffraction effects caused by surrounding terrain and other vehicles in the LOS should not be neglected either. Figure 6.9 demonstrates the impact that considering both 3D antenna patterns and terrain has on a simulation measuring the average number of neighbors in reach of a car. It shows substantially differing numbers for the three setups independent of all simulated starting times (and thus traffic densities).

In conclusion, it can be said that antenna characteristics (as well as diffraction effects in the 3D case) should be taken into account for more realistic and reliable results. The means to achieve that are readily available in the Veins framework.

6.2.4 Unit Testing in Veins

Automated testing has become a central element of modern software development. In a world of rapidly changing requirements and short development cycles, a quick and repeatable assurance of code correctness is essential. Automated testing can provide such assurance by running suites of programmed tests. Each test calls a portion of the original code and compares the results (and in some cases side-effects) to *reference values* embedded in the test. If all tests pass and the test suites cover all (or a large enough portion) of the original code, one can be assured that the code behaves as expected. If some tests fail, one can gain hints about which part of the code is not behaving as expected by observing which tests fail and which portion of the code they call.

The whole process of running a test and evaluating its results can nowadays be integrated into software version control and development workflows to support continuous integration.

Veins users usually implement models of algorithms or protocols to conduct research. While this is not the same as releasing a software product to end users, asserting correctness of the software is just as important in this domain. Before being able to rely on data generated from a simulation (or, indeed, publishing findings based on it), the author has to be confident that all models of the simulation behave as expected. Typically, this is done by comparing measurements recorded from the simulated model to reference data obtained from analytical models or from conducted real-world measurements.

This approach treats the model as a single large black box. Only behavior that is observable from the outside is compared to reference data. While this approach is useful to verify the overall correctness of the model, it is hard to cover the model's complete behavior. For example, there may not be enough reference data for all use cases or there may be mechanisms inside the model that are hard to verify from the outside. Finally, manually comparing the model with reference data is a cumbersome process that takes time and may be prone to errors due to its repetitive nature.

Aside from manual result comparison, Veins and OMNeT++ have provided three more automated testing mechanisms for a while now.

The first one is a simple regression testing approach, already described in Chap. 1. After a simulation finishes, OMNeT++ can output its *fingerprint*, a hash value of its defining characteristics (such as its event trace). Later fingerprints can then be used to verify that changes in the code did not affect how the simulation behaves.

The second testing mechanism is to simply run a simulation model that itself contains calls into model code and assertions to check the results. Veins uses this in its `TraCITestApp` to check if some basic interactions with the SUMO traffic simulation lead to expected results. This approach shares the pros and cons of the general usage of assertions within model code: preconditions and postconditions within model code can easily be checked and are straightforward to write. However, the approach is unstructured and not well suited for testing an entire simulation model. Many checks have to be integrated into a single simulation scenario and may depend upon each other. The whole simulation kernel has to be loaded which eventually increases runtime and complexity. Finally, there is no support for established testing tools, e.g., for automation, coverage reporting, or debugging.

The third mechanism is the OMNeT++ **opp_test** tool.[7] It can run tests in a managed environment similar to the execution environment of the OMNeT++ simulation kernel. Message creation and sending as well as result recording and other OMNeT++ utilities are available just like during simulation execution. Such an environment is hard to set up manually when using generic testing facilities. Tests are completely encapsulated into single test files and run by the **opp_test** tool. Correctness can be ensured by observing the successful termination of the simulation and by validating simulation output of file streams, such as result files and standard output or standard error. All of this makes **opp_test** the prime tool for testing OMNeT++, its modules, and code that is tightly integrated with (or relying on) OMNeT++ mechanisms, such as messages and channels. For everything that is not touching the OMNeT++ simulation kernel or library, however, **opp_test** is not the most straightforward tool to use. The test file format introduces unnecessary overhead—and having to find all values to check against in file streams is cumbersome.

Thus, for testing plain C++ code, more generic unit testing frameworks provide a better solution. A very popular example of such frameworks is Catch2.

Catch2 is a powerful C++ unit testing framework that facilitates writing, running, and evaluating unit tests for C++ code.[8] Tests are written in plain C++ (with some macros), compiled into an executable, and run by a built-in runner application. This runner allows control of the way tests are run, e.g., output verbosity and format (e.g., for continuous integration services). It can also limit a run to subsets of the test suite via tags to speed up execution time or automatically spawn a debugger on failing tests. As it is easy to learn, powerful in its capabilities, and available under a nonrestrictive license (i.e., the Boost Software License Version 1.0), it is an ideal framework to test Veins code that does not touch the OMNeT++ kernel or library.

Limiting tests to plain C++ code may appear to be a restriction, but it is actually an opportunity for improving the code design. When implementing models of algorithms and protocols using Veins, ideally only a small fraction of the code has to be written specifically for OMNeT++. Algorithms can easily be expressed as pure C++ functions or classes—and even protocol implementations can be written more cleanly if they do not rely on the concrete messaging model employed by the OMNeT++ simulation kernel. Code written in such a manner contains fewer external dependencies and moving parts. Especially the ownership model of OMNeT++ messages (which heavily relies on passing raw pointers) is contrary to C++ best practices of high-layer application development and a common source of errors. Integration can then happen in a thin layer of code that implements OMNeT++ modules, channels, or messages and, e.g., could be tested with the **opp_test** tool. This approach results in code that is much easier to test and also much easier to debug (as models can be executed without a full simulation environment) and to port to other simulators or platforms at the same time.

[7]https://www.omnetpp.org/doc/omnetpp/manual/.

[8]https://github.com/catchorg/Catch2.

In Catch2, tests are implemented in C++ files (see Listing 6.3) which only have to include a single header file. As discussed, these files are compiled individually and linked together, which also includes a special runner program that is generated by Catch2. The result is a plain binary that can be executed to run the contained tests.

Since Veins 4.7, this process has been automated in the Veins_Catch subproject. The subproject contains a *Makefile* that automatically builds the test binary from C++ files found in its source directory. It dynamically links the file to the shared library compiled from the original Veins code, so that both can be built individually. This also means that the original Veins code is fully independent of the test code. The test code, on the other hand, only needs to include header files from Veins code as if it were a part of Veins itself.

In addition, all the components (including the test runner) are compiled individually and only have to be recompiled if changed. As a result, build times stay short, which benefits frequent testing and development styles like Test-Driven Development (TDD).

In order to run the tests, one only has to execute the **veins_catch** binary produced by the *Makefile* (given that it and Veins itself have been successfully compiled). The binary provides a number of command line switches to control how and which tests are run. For example, `-s` provides detailed output even for successful tests and `-b` spawns a debugger in case of an error or failed test. The names or tags of tests to be run can be given as command line arguments. See the Catch2 documentation or run it with the `-v` switch for more information.

New tests can be added to existing or new C++ files in the *src* directory within the Veins_Catch subproject. Ideally, every unit (e.g., class or set of functions) should get its own file, mirroring the structure of the original Veins code to some degree. Each such test file first has to include the Catch2 header file (*catch/catch.hpp*) and then any headers of Veins components it wants to test against. Include paths are already configured in a way such that test code can include headers from Veins code as if it was a part of Veins itself. However, if new libraries or dependencies are introduced to Veins (in a way that affects header files), the configuration of Veins_Catch has to be adapted in the same way.

Tests can be written in two styles: normal and Behavior-Driven Development (BDD) style. The former is faster to type, the latter is more expressive in terms of debug output and test case structure. In any case, each individual test case (either stated as a `TEST_CASE` or a `SCENARIO`) gets a description text. Within such a test case, one can write arbitrary C++ code to set up the test. Assertions are then added via the `REQUIRE` macro (there is in fact a whole family of macros to cover a wide range of use cases). It is important to always add at least one assertion to each test case, otherwise it might not be run. Sample test cases and invocation of the unit tests are shown in Listings 6.3 and 6.4.

Listing 6.3 Sample test case written in Catch2 in the Veins_Catch subproject

```cpp
#include "catch/catch.hpp"
#include "veins/modules/mobility/traci/TraCICoordinateTransformation.h"

using Veins::TraCICoordinateTransformation;
using Veins::TraCICoord;
using OmnetCoord = TraCICoordinateTransformation::OmnetCoord;

SCENARIO( "coordinates can be transformed", "[netbound]") {
   auto o1 = OmnetCoord(2414.90142, 1578.44161, 0.0);
   auto t1 = TraCICoord(646854.991, 5493242.54);

   GIVEN( "The boundaries from a scenario" ) {
      TraCICoordinateTransformation nb{ {644465.09, 5491786.25},
           {647071.55,5494795.98}, 25 };

      THEN( "omnet coords correctly translate to traci coords" ) {
         auto t2 = nb.omnet2traci(o1);
         REQUIRE( t2.x == Approx(t1.x) );
         REQUIRE( t2.y == Approx(t1.y) );
      }
      THEN( "traci coords correctly translate to omnet coords" ) {
         auto o2 = nb.traci2omnet(t1);
         REQUIRE( o2.x == Approx(o1.x) );
         REQUIRE( o2.y == Approx(o1.y) );
      }
   }
}
```

Listing 6.4 Sample invocation of test cases in the Veins_Catch subproject

```
veins/subprojects/veins\_catch% ./configure
Creating Makefile in veins/subprojects/veins_catch/src...
veins/subprojects/veins\_catch% make
Creating binary: src/veins_catch
veins/subprojects/veins\_catch% ./src/veins_catch
All tests passed (4 assertions in 1 test case)
```

6.2.5 Simple Timer Management

A common use case in many protocols is the handling of timers, that is, doing something and—after a certain time interval elapsed—doing something else, possibly repeatedly. OMNeT++ offers the concept of *Self-Messages* to support this use case: any simulation module may schedule an event to be delivered to itself (using the `scheduleAt` method), annotating its event handler with code to treat this special event as expiration of a timer. Commonly, users create such events in a module's `initialize` method, schedule them in some user-defined method, and handle them in a module's `handleMessage` method.

At the scale (regarding number of timers) needed for many advanced protocols, however, this way of modeling timers has a number of drawbacks for code complexity. Because creation, scheduling, and handling of timeouts is split across multiple methods, data must be communicated in the events themselves (commonly found patterns subclass from `cMessage` to achieve this). Further, boilerplate code needs to be included with every handler to free memory or re-schedule repeated events, depending on whether the timer is a one-shot or a repeating one.

Starting with Veins 4.7, the model library includes a utility class *TimerManager* to ease writing timers. It supports users needing to write timers in two respects:

- it takes care of all memory management associated with OMNeT++ events; and
- it enforces robust, modern coding standards by relying on C++11 lambda constructs (or, indeed, any `std::function`) for passing data to callback handlers.

To use this functionality, all an OMNeT++ module needs to do is create a private instance of the `TimerManager` class and pass received events to it (by introducing a small chunk of code in its `handleMessage` method). Timers can then be introduced by calling the `create` method of this private instance, passing it an object containing a lambda to execute when the callback fires. This lambda can, of course, bind any local variable (or a reference thereto) that needs to be available in the callback.

Listings 6.5 and 6.6 illustrate the use of the `TimerManager` class by way of a simple example. Though (for the single timer taking a single integer value demonstrated in this example) the overhead in terms of code that needs to be written is identical to a solution using raw OMNeT++ events, it is easy to see that this overhead is now simply a constant—independent of how many timers need to be managed by a protocol implementation.

Listing 6.5 Content of the *TimerExample.h* file

```cpp
#include "veins/modules/utility/TimerManager.h"

class TimerExample : public cSimpleModule {
 protected:
   virtual void initialize();
   virtual void handleMessage(cMessage *msg);
   // create a private instance of the TimerManager
   Veins::TimerManager timerManager{this};
};
```

Listing 6.6 Content of the *TimerExample.cc* file

```cpp
Define_Module(TimerExample);

void TimerExample::initialize() {
   int n = intuniform(0, 255);
   // example: remind ourselves about the value of n in 500ms from now
   auto callback = Veins::TimerSpecification([this, n](){
      EV << "value of n was " << n << std::endl;
   });
   timerManager.create(callback.oneshotIn(SimTime(500, SIMTIME_MS)));
}

void TimerExample::handleMessage(cMessage *msg) {
   // allow TimerManager to handle any timer events
   if (timerManager.handleMessage(msg)) return;
   // regular handleMessage follows...
}
```

Naturally, the `TimerManager` instance offers not just a method to create, but also to cancel timers—and timers can be both one-shot and repeating (either in a given time interval or for a given number of repetitions).

6.3 Use Cases

In this section, we present two practical use cases for Veins. We give insights on the simulation of platoons (cf. Sect. 6.3.1) and intersection scenarios (cf. Sect. 6.3.2).

6.3.1 Simulation of Platoons

Cooperative driving and automated car following (or platooning, illustrated in Fig. 6.10), although not a new idea, is now an active research topic due to the ever-increasing demand for highly safe and sustainable transportation. In brief, the idea of platooning is to form road trains of vehicles—where one vehicle leads the group and others autonomously follow it. The *follow distance* should be small, much shorter than the *safety distance* maintained by human drivers. A close following gap improves infrastructure utilization, as it reduces the portion of road wasted for the safety distance. With an improved utilization comes a reduction of traffic congestion, resulting in a more sustainable transportation infrastructure. In addition, a distance in the order of a few meters reduces the air drag, lowering fuel consumption and thus emissions. Finally, autonomously driven vehicles can improve safety: more than 90% of road accidents are due to human errors [11].

Platooning is becoming technologically feasible, as witnessed by the projects working on this topic and realizing successful Field Operational Tests (FOTs), such as SARTRE, PATH, KONVOI, COMPANION, and PROMOTE-CHAUFFEUR [7, 30, 33, 35, 51]. Before the actual market introduction, however, platooning should be tested in large scale settings to understand to which extent platooning technologies and solutions would provide their expected benefits. In this setting (i.e., with tens or hundreds of vehicles) FOTs are simply infeasible. The solution is thus to resort to realistic simulations and this is what PLEXE has been designed for [47, 48].

PLEXE is a Veins extension designed for the analysis of platooning systems from different perspectives. From a low-level perspective, it enables the analysis

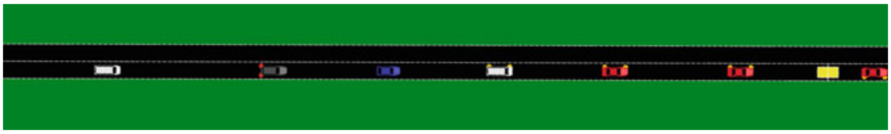


Fig. 6.10 Screenshot of a platoon simulated in Veins

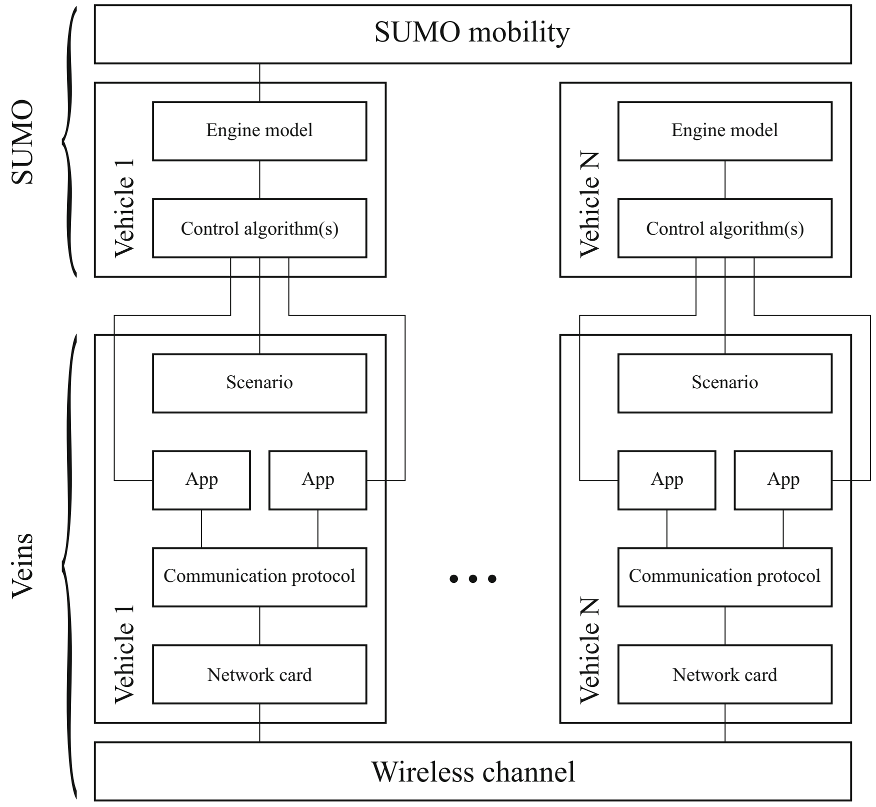


Fig. 6.11 High-level architecture of PLEXE components

of cooperative control systems under realistic vehicle dynamics and network conditions. This is especially useful to understand the impact of network impairments on the performance of the control system, including heterogeneous vehicles in the analysis. From a high-level perspective, PLEXE permits to design, implement, and test platooning maneuvers, as well as to analyze the impact of different strategies on traffic efficiency.

Figure 6.11 shows the high-level architecture of PLEXE. It does not only extend Veins, but also SUMO:

- On the SUMO side, autonomous control algorithms and vehicle dynamics are implemented.
- On the Veins side, users can develop protocols and applications which take high-level decisions on vehicles' behavior.

On the SUMO side of PLEXE, the difference between a "standard" SUMO simulation and a simulation of a *cooperative* driving system is mobility modeling. SUMO is designed for the simulation of transportation systems with a special focus on human traffic. Vehicles behave as dictated by *car-following models* which decide, for each time-step, what a vehicle should do depending on its surrounding environment, including other vehicles, intersections, traffic lights, etc. Standard SUMO models such as the Intelligent Driver Model (IDM) [60] or the Krauss model [32] reproduce mobility patterns which are typical of human driving. In cooperative driving, instead, decisions are taken by an automated system, which clearly behaves in a completely different manner. In this regard, PLEXE implements a new car-following model in SUMO which embeds different control systems—and that can thus behave like a cooperative autonomous vehicle.

More formally, PLEXE gives access to a set of systems called *cruise controllers*. The Cruise Control (CC) controller, as the name suggests, automatically maintains a desired speed set by the driver: this way there is no need to keep the foot on the throttle. This system is only a comfort feature, as the driver is required to manually disengage it when approaching a slower vehicle. The next step in automation, automatic braking, is provided by the Adaptive Cruise Control (ACC), which exploits a radar mounted in the front bumper to maintain a safety gap to the front vehicle, if required.

Although the ACC provides the required functionality, it does not implement platooning in the strict sense. The reason is that, due to the delays introduced by the engine driveline and the radar sensor, it cannot perform close following [41]. The safety distance maintained by an ACC is comparable to safety distances typical of human driving, and it would thus fail in providing the required benefits.

The solution to this problem comes from cooperation, i.e., by sharing information through a wireless link to implement a Cooperative Adaptive Cruise Control (CACC) [1, 20, 38, 40, 42, 45] (how this information is shared via the wireless link is modeled in the Veins side of PLEXE, described later in this section). A CACC can have a huge performance improvement with respect to an ACC as communication overcomes the limitations of sensor-based systems. As an example, exploiting a wireless link, the leader can communicate with all its members simultaneously, while a front-mounted radar is only capable of providing information about the preceding vehicle. In addition, any vehicle can share intended actions which will be executed in the near future: a radar can only sense an event after its occurrence.

In essence, the PLEXE car-following model in SUMO makes it easy for users to implement cruise control algorithms. PLEXE already provides some sample implementations, i.e., the ACC defined in [41] and the CACCs designed in [40, 42, 45]. The software is in continuous development and newly developed control systems are announced on the official website.[9]

In addition to the control algorithms, PLEXE models engine characteristics and vehicle dynamics. The control system computes a desired acceleration which needs to be realized by the vehicle. This process, however, requires a certain amount of time, the *actuation lag*, due to the engine driveline or to the braking system. This can be properly taken into account, increasing the realism of the analysis and

[9]PLEXE website: http://plexe.car2x.org.

the trustworthiness of the results. PLEXE provides two sample implementations: a simple but widely assumed first order lag (i.e., a first order low-pass filter) as well as a realistic engine model which takes into account engine torque curve, gear ratios, vehicle mass, aerodynamic characteristics, etc. Describing these models is out of the scope of this chapter. The interested reader can find a detailed mathematical description of the models (as well as of the control algorithms) in [46].

We now turn to the Veins side of PLEXE. On the Veins side of the simulation, each vehicle has a corresponding network node implementing communication protocols, applications, and scenarios (see Fig. 6.11). Each module can influence the behavior of its corresponding vehicle (or retrieve data about it) using the extension of the TraCI Application Programming Interface (API) provided by PLEXE.

The scenario module implements the high-level behavior of the vehicle. Two basic examples included in the online tutorial are the sinusoidal and the braking scenarios. In the first, the scenario continuously changes the leader speed to analyze the behavior of the control system under disturbance. In the second, the leader instead performs an emergency braking, coming to a complete stop.

Applications influence the behavior of vehicles as scenarios do, but they do so based on the information they receive through wireless communication. The most simple example is feeding the CACC using the data of a member of the same platoon. In this case, depending on whether the information is correctly received or not, the behavior of the vehicle changes (as the CACC computes different control actions). Another use case is the implementation of a maneuver and its corresponding protocol. In the case of a *join* maneuver, for instance, a vehicle might get instructions for joining from the leader of a platoon.

Below the application level we find communication components. In particular, we have communication protocols that implement beaconing strategies. This way it is possible to understand what happens to the control system depending on the employed data dissemination mechanism [49, 50]. As an example, the user can analyze the difference between a static beaconing approach vs. a coordinated one. Even further down the stack, we find the network card and the wireless channel models that are included in the standard Veins release. They provide the necessary level of realism for IEEE 802.11p-based V2V communication.

PLEXE's structure on the Veins side is meant for defining the base concepts and to ease the development process. It also enables users to define their own application/communication structure, providing them with the maximum possible flexibility.

6.3.2 Communication on Intersections

In May 2018, the European Commission announced that it wants to reduce the number of fatalities per year on European roads by 2050 to nearly zero. Beside passive safety measures (e.g., advanced seat belts, improved safety glass) the commission proposed different kinds of active safety measures (often called Advanced Driver

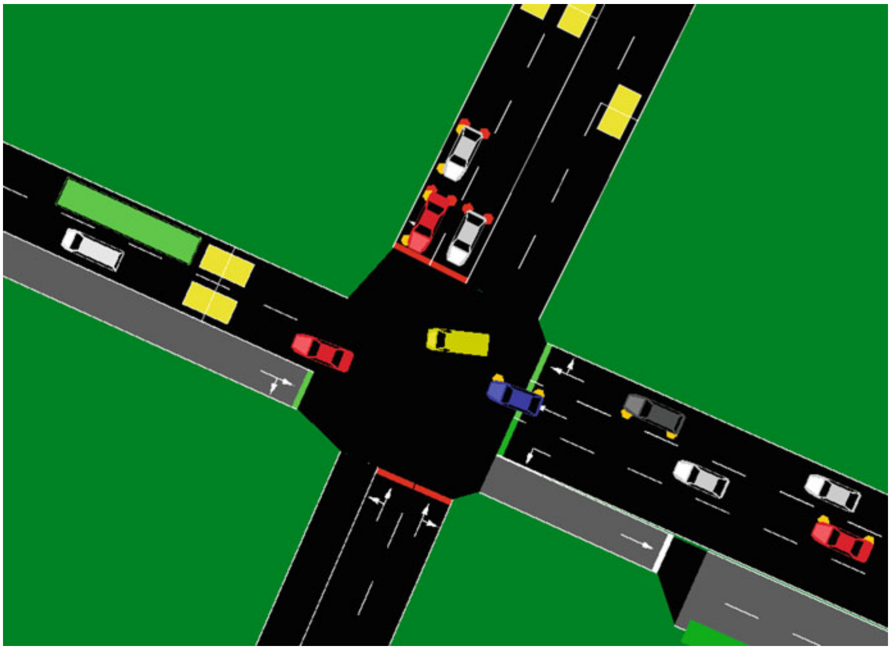


Fig. 6.12 Screenshot of an intersection simulated in Veins

Assistance Systems (ADAS)), which aim to support drivers and prevent accidents. The envisioned safety features of future vehicles include advanced emergency braking, intelligent speed assistance, and lane keeping assistance.

Some of the safety-relevant ADAS do already exist and use various sensor technologies to assess the situation. However, the current systems are limited by their sensors to visual range. Using inter-vehicle communication, sensor data can be distributed among vehicles outside one another's field of view. One prominent example is Intersection Assistance Systems (IAS) which rely on location and movement information. Veins is a natural fit for simulating communication while vehicles are approaching an intersection (illustrated in Fig. 6.12) in a potentially dangerous situation. SUMO, on the other hand, is designed to simulate collision-free traffic, which makes it a less natural fit. Its car-following models are designed to be collision-free, i.e., vehicles approaching an intersection will never have a crash nor get into a potentially dangerous situation. However, starting with SUMO version 0.20.0, it is possible to turn off different safety checks of the car-following models. Hence, simple crash situations can be simulated by letting two vehicles start at the same time and distance to an intersection. In addition, the time when safety checks are disabled can be varied and hence a wider variety of crash situations simulated. This is possible for all implemented car following models.

Several measurements on how drivers approach intersections can be found in the literature [5]. A comparison of existing car-following models (e.g., the Krauss model [32] or the IDM [60]) quickly reveals that the IDM better reflects human behavior when approaching an intersection [28]. Note also that default simulation time steps (in the magnitude of seconds) for data exchange between SUMO and OMNeT++ will not allow to sufficiently model such complex situations. Depending on the vehicular safety application under investigation, simulation time steps between 1 and 100 ms will be reasonable. For a detailed analysis of the simulation time step, we refer the reader to the literature [26].

In the following, insights on how Veins can be used to research safety metrics and situation-aware communication for IAS are shared.

Other than typical metrics of network behavior (like latency or load) and typical metrics of road traffic behavior (like emissions or travel time [54]), the primary metrics for traffic intersection must assess criticality. Metrics that assess the criticality of driving situations are called *safety metrics*. The criticality of a situation can only be estimated when driving information of all involved vehicles is available. The information to estimate the risk depends heavily on the situation, but might include: the exact geographical position, the driving direction, the speed, the acceleration, the planned route, or even typical driving behavior of the current driver. Please note that most of these parameters can be accessed in Veins directly or by an extension of the data exchange interface (TraCI) between SUMO and OMNeT++.

Finally, it is of course important to detect crash situations at intersections. This feature is implemented in Veins, which also enables the result recording of interesting simulation data directly in OMNeT++.

In the following, a closer look on intersection scenarios is presented, i.e., a possibility to estimate the likelihood of a crash at an intersection is explained. For a detailed description, we refer the reader to the literature [29]. The considered information for two vehicles A and B, which are approaching an intersection, is as follows:

- distances d_A and d_B reflecting the distance to the intersection of trajectories,
- speeds v_A and v_B, and
- the maximum acceleration $a_{\max}$ and the maximum deceleration (negative) $a_{\min}$.

The values of $a_{\min}$ and $a_{\max}$ would, of course, be different for each vehicle, but the vehicle-dependent indices are omitted for simplicity.

The intersection collision probability can be estimated by considering all possible driver behaviors (called trajectories) of approaching vehicles. A trajectory is a feasible function of time that satisfies the constraints:

$$\mathcal{T}_A(t_0) = d_A, \quad \dot{\mathcal{T}}_A(t_0) = v_A, \quad a_{\min} \leq \ddot{\mathcal{T}}_A(t) \leq a_{\max}. \tag{6.4}$$

All possible future trajectories are denoted as $\mathbb{T}_A$ and defined by $\mathbb{T}_A = \bigcup \mathcal{T}_A$. Of course, this set depends on the current distance d_A and speed v_A as each trajectory does. In addition, it is limited by the two trajectories applying constant maximum acceleration $a_{\max}$ and constant maximum deceleration $a_{\min}$.

A crash between vehicles A and B happens if the bounding boxes (defined by the length and width of the vehicles) overlap. This is used to define a function

$\mathrm{coll}\,(\mathcal{T}_A, \mathcal{T}_B)$, which returns 0 if no crash happens and 1 if a crash happens for the given trajectories.

The intersection collision probability $\mathcal{P}_C$ depends on the probability that two trajectories are chosen which lead to a crash. This probability function is denoted as $p(\mathcal{T}_A, \mathcal{T}_B)$. Therefore, the intersection collision probability $\mathcal{P}_C$ can be calculated by integrating over all possible trajectories and summing up the probabilities as follows:

$$\mathcal{P}_C = \int_{\mathbb{T}_B} \int_{\mathbb{T}_A} p(\mathcal{T}_A, \mathcal{T}_B)\,\mathrm{coll}\,(\mathcal{T}_A, \mathcal{T}_B)\ \mathrm{d}\mathcal{T}_A\,\mathrm{d}\mathcal{T}_B. \tag{6.5}$$

Aside from serving as an output metric of simulations, this metric can also be used to improve communication on intersection scenarios, as we describe subsequently.

Figure 6.3a on page 221 shows that Veins already provides all necessary lower layers for evaluating communication strategies. Therefore, one can directly start designing the application layer, e.g., a message dissemination algorithm, which determines parameters like the content of messages or the interval of message generation. The content of the message may include position, speed, acceleration, and heading, but also neighbor information (last received message sequence number or time) might be helpful for advanced communication strategies.

The message generation interval was subject to extensive research during the past decade (e.g., [58]). Several *congestion control mechanisms* have been proposed to keep the channel load in a reasonable and efficient range. To improve communication reliability in dangerous situations, safety metrics can be used to alter the message dissemination interval alongside congestion control mechanisms.

The intersection collision probability can be used to realize situation-aware communication for intersections. Basically, each vehicle can calculate its intersection collision probability when receiving a message from another vehicle. If the probability exceeds a certain threshold, the vehicle will temporarily lower its message dissemination interval accordingly. Hence, vehicles in a dangerous situation are trying to communicate more frequently, whereas others will automatically adapt their message intervals (which, in turn, helps to keep the channel load balanced).

Finally, proposed communication strategies (such as situation-aware communication) need to be evaluated. Basically, a detailed analysis of message arrival times (which can be recorded in OMNeT++) is sufficient. The following three metrics represent a basis for evaluating communication strategies of safety applications (refer to [26] for details):

- *Last Before Unavoidable*: the last message received and the point in time before a crash becomes unavoidable is of particular concern.
- *Worst-Case Update Lag*: the update lag measures the time between two consecutive messages. Obviously, the most critical update lag is the longest during a certain time interval before a crash happens (called worst-case update lag).
- *Unsafe Time*: when a certain update lag is required by an application, it can help to sum up all times where the update lag was not maintained.

6.4 Extensions

In this section, we discuss how to use Veins in simulations involving LTE networks (see Sect. 6.4.1) and in simulations involving regular Internet-centric protocols, that is, with models included in the INET Framework (see Sect. 6.4.2). A discussion of Instant Veins for classroom use and quick deployment in general (see Sect. 6.4.3) concludes the chapter.

6.4.1 Using LTE Models in Veins (Veins LTE)

Combining multiple networking technologies for ITS is called *Heterogeneous Vehicular Networking*. In the context of vehicular networks these networks are often LTE- and IEEE 802.11p-based, i.e., a cellular and an short range communication network. LTE is already widely deployed, mainly for use in mobile phones—but new cars regularly come equipped with a Subscriber Identity Module (SIM). In the EU, as of April 2018, all new cars need to be equipped with the *eCall* system used to automatically call emergency services if an accident occurs. Due to the centralized nature of cellular networks, scheduling can be used to handle situations with an overloaded channel. Nevertheless, there are disadvantages: vehicles are not considered in currently deployed cellular networks, especially when it comes to periodic beaconing. Since 2017, LTE-Vehicle-to-Everything (V2X) is standardized in a first version in LTE Release 14 as an extension to LTE Device-to-Device (D2D). The standard added two additional D2D modes which specifically focus on vehicular networking, one of them requiring an evolved Node B (eNB) (Mode 3) while the other one works in a distributed manner (Mode 4). Nonetheless, there are various points of discussion and an update to LTE-V2X is included in LTE Release 15, which is scheduled for release in 2018. If the infrastructure cannot cope with the additional load of cars using the cellular network (mode 4 is only used when there is no eNB in transmission range), there is the question who pays for the necessary upgrades. Not only the infrastructure improvements need to be paid for, someone needs to finance the usage of the cellular networks. Currently this is mostly included in the price of the cars, but if more cars come equipped with cellular technology this might change. Overall cellular networks are an alternative to WLAN-based networks when it comes to vehicular networking. Nevertheless, they have their own disadvantages, so research has been conducted to use both technologies. Additionally, research is conducted on other alternatives such as VLCs and Bluetooth.

The basic idea of heterogeneous vehicular networking is to use the strengths of one networking technology to overcome the weakness of the other when used in a certain application scenario. Take long-range communication in vehicular networks as an example. Transmitting data to a distant node in an IEEE 802.11p-based network needs a connected network from the source car to the destination. If the network is not fully connected, the data might get lost or a large delay due to the use of store-carry-forward is induced. When using cellular networks, this is not the case as long as the cars are in range of a base station, i.e., an eNB, which can handle the transmission via the backbone. Similarly, IEEE 802.11p-based networks allow to have a simpler (and potentially faster) communication between cars close to each other compared to using a cellular network where every message potentially needs to traverse the backbone. Furthermore, heterogeneous technologies can be a *fall back* mechanism if one of them is not available. This might prove useful in the initial deployment phases of connected vehicles where IEEE 802.11p will not be used widely while LTE infrastructure exists already.

When using Veins alone, cellular networks can only be simulated in a very rudimentary way by using direct communication with a small amount of delay. Therefore, various solutions exist which support and provide more complete heterogeneous simulations:

- *SimuLTE*: a framework for OMNeT++ which recently introduced experimental support to combine it with Veins [61–63].
- *Veins LTE*: a framework integrating Veins and SimuLTE resulting in a toolbox to develop algorithms exploiting both IEEE 802.11p and LTE channels [21].
- *Artery*: a framework for simulation of ETSI ITS-G5 protocols which among various others includes Veins and SimuLTE [43].

As an example of a framework providing support for heterogeneous vehicular networks, we briefly introduce the first one with support for vehicles, i.e., *Veins LTE* and its features [21] (a discussion of SimuLTE and Artery can be found in Chaps. 5 and 12, respectively). Veins LTE combines short-range communication (Veins providing IEEE 802.11p) with cellular communication (SimuLTE providing LTE).

SimuLTE is, as the name gives away, a simulation model library for LTE. It currently provides support for the major parts of LTE including base stations (eNodeBs), mobile nodes (UEs), a (nearly) complete data plane, multiple example applications, an extensive MAC layer implementation, the backbone in the form of the X2 interface [39], and various basic scheduling algorithms. The downsides of SimuLTE are that it focuses on the user plane and only covers a rudimentary control plane as well as only basic handover between base stations. Both simulation model libraries, SimuLTE and Veins, are based on OMNeT++, which allows to integrate them with each other. The focus of the integration was to include cars as nodes into the cellular network. While instantiating models from both libraries at the same time is easy due to them both using OMNeT++, there are certain issues with their network models, which have proven to be incompatible. This is especially true for the treatment of mobility. On the one hand, SimuLTE did expect a fully set up network (including all moving nodes) and did not allow nodes to enter or leave at runtime. On the other hand, Veins relies on nodes dynamically entering and leaving the simulation to simulate realistic traffic conditions. To successfully integrate Veins with SimuLTE, the whole LTE stack on User Equipment (UE) and eNB side was

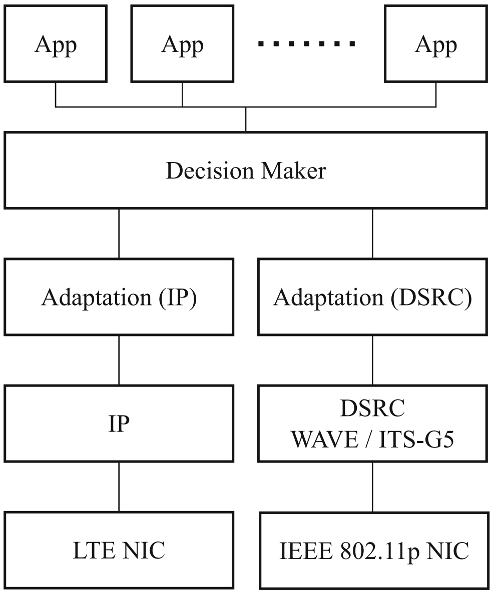


Fig. 6.13 The heterogeneous networking stack introduced in Veins LTE [21]

modified to accommodate the addition of new vehicles and the correct removal of them during runtime.

The overall architecture of Veins LTE can be seen in Fig. 6.13. To make the development of new heterogeneous algorithms easier, a new layer was introduced—the *Decision Maker*. Residing between the application layer and the two network stacks, it adds the possibility to provide a scheduler spanning both the IEEE 802.11p and the LTE network stack. If the application has set a specific network technology, the corresponding stack is used by this module, even if the chosen network is currently not available. If no such network is set by the application, this layer allows a developer to add a decider module, which decides the network layer on which to send the packet. Such a scheduler can, for instance, choose the target network stack based on the channel load or make this decision based on the distance between sender and receiver. Furthermore, this is useful to test an algorithm with barely any configuration overhead both in an IEEE 802.11p and in an LTE setting. Below this layer is the adaption layer, which adds the necessary parameters to the heterogeneous message in order to make it possible to send it via the chosen stack. An application only needs to set the most basic parameters (e.g., destination, payload) and the rest is added or adapted by the decision maker layer. After applying the necessary attributes to the messages, they are sent via the selected networking stack.

These features, especially the integration of two networking technologies and the decision layer, allow a user of Veins LTE to focus on the development of the algorithm rather than on the underlying network.

6.4.2 Using INET Framework Models in Veins (Veins_INET)

Often, Veins simulations need to be combined with simulations of common Internet protocols. Conversely, systems employing Internet protocols like those of cloud services, backbone networks, or Mobile Ad Hoc NETworks (MANETs) often need to be simulated with nodes carried in road traffic. One is the domain of Veins, the other is the domain of the INET Framework, the prime OMNeT++ model library for Internet protocol simulation (see Chap. 2).

Veins hence includes an extension, `Veins_INET`, which allows models of the INET Framework to use Veins as a mobility model. Because many other simulation model libraries, in turn, rely on INET for modeling node movement, this extension also allows any of these simulation model libraries to model nodes in road traffic.

The extension is included as a *subproject*, that is, as a separate simulation project but contained in the source tree of Veins.

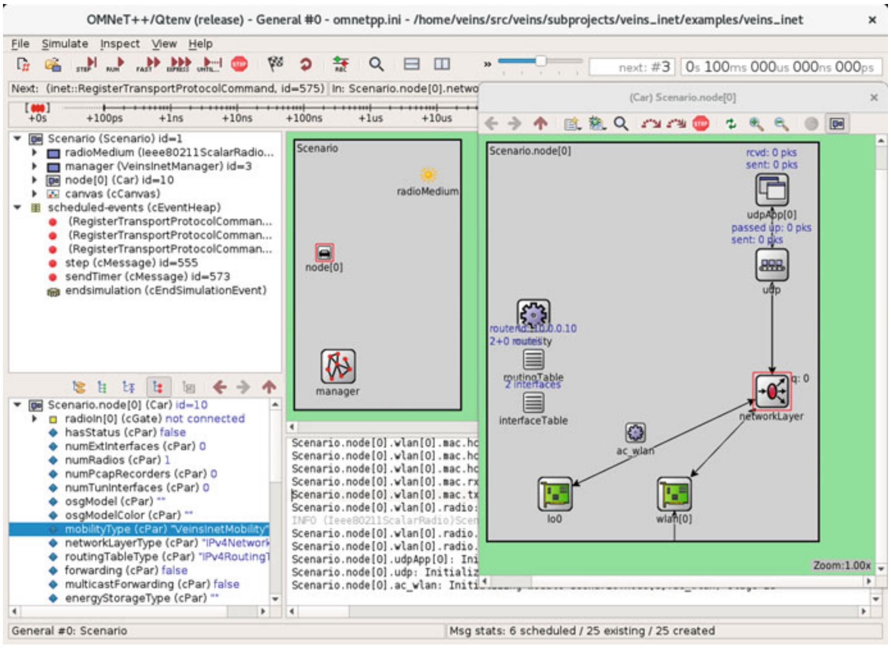
All that is needed is to have the target simulation project use the model libraries of all of Veins, the INET Framework, and Veins_INET. In the OMNeT++ Integrated Development Environment (IDE), this is achieved by importing all three projects into the workspace and changing the target simulation's project settings to use all three as *referenced* projects. On the command line, this is achieved by supplying the corresponding `-I`, `-L`, and `-l` switches to **opp_makemake**—as well as the corresponding `-l` and `-n` switches to **opp_run**.

In such simulations, instantiating a module `VeinsInetManager` in the network will take care of connecting to a SUMO road traffic simulation, instantiating one simulation module per road traffic participant in the SUMO simulation, and updating the modules' position information as the simulation executes (as detailed in Sect. 6.2.1). Care must only be taken that modules intended to represent road traffic participants contain `VeinsInetMobility` as their INET Framework mobility module (e.g., by configuring this in the *omnetpp.ini* file).

Figure 6.14 illustrates such a combined simulation. Note the presence of a `VeinsInetManager` module in the network (named "manager") and a mobility module of type `VeinsInetMobility` (named "mobility") in the module representing a car.

6.4.3 Instant Veins

Many moving parts comprise a typical Veins simulation: (1) the road traffic simulation tool SUMO; (2) the OMNeT++ simulation kernel, (3) the simulation model under study, and (4) all model libraries it is based on. For example, a simulation model of vehicles communicating with a cloud service reachable via LTE will typically rely not just on Veins, but also on INET (for Internet protocols, see Chap. 2), SimuLTE (for LTE simulation models, see Chap. 5), as well as Veins_INET (for linking them together, see Sect. 6.4.2).


```
OMNeT++/Qtenv (release) - General #0 - omnetpp.ini - /home/veins/src/veins/subprojects/veins_inet/examples/veins_inet
Next: (inet::RegisterTransportProtocolCommand, id=575) In: Scenario.node[0].netwo

Scenario (Scenario) id=1
  radioMedium (Ieee80211ScalarRadio...
  manager (VeinsInetManager) id=3
  node[0] (Car) id=10
  canvas (cCanvas)
scheduled-events (cEventHeap)
  (RegisterTransportProtocolComman...
  (RegisterTransportProtocolComman...
  (RegisterTransportProtocolComman...
  step (cMessage) id=555
  sendTimer (cMessage) id=573
  endsimulation (cEndSimulationEvent)

Scenario.node[0] (Car) id=10
  radioIn[0] (cGate) not connected
  hasStatus (cPar) false
  numExtInterfaces (cPar) 0
  numRadios (cPar) 1
  numPcapRecorders (cPar) 0
  numTunInterfaces (cPar) 0
  osgModel (cPar) ""
  osgModelColor (cPar) ""
  mobilityType (cPar) "VeinsInetMobility"
  networkLayerType (cPar) "IPv4Network
  routingTableType (cPar) "IPv4RoutingT
  forwarding (cPar) false
  multicastForwarding (cPar) false
  energyStorageType (cPar) ""

Scenario.node[0].wlan[0].mac.hc
Scenario.node[0].wlan[0].mac.hc
Scenario.node[0].wlan[0].mac.hc
Scenario.node[0].wlan[0].mac.rx
Scenario.node[0].wlan[0].mac.tx
Scenario.node[0].wlan[0].radio:
INFO (Ieee80211ScalarRadio)Scen
Scenario.node[0].wlan[0].radio.
Scenario.node[0].wlan[0].radio.
Scenario.node[0].udpApp[0]: Ini
Scenario.node[0].udp: Initializ
Scenario.node[0].ac_wlan: Initi

General #0: Scenario          Msg stats: 6 scheduled / 25 existing / 25 created
```

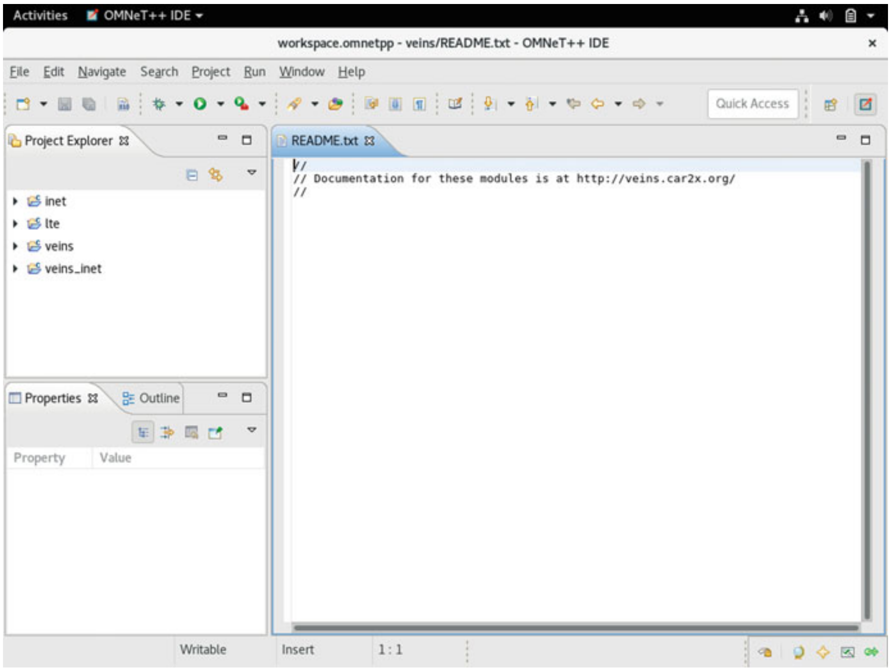
Fig. 6.14 Screenshot of the sample simulation of Veins_INET running in the OMNeT++ GUI

Interested users have to download all of these components, compile them, and configure them for linking into a mixed simulation. In addition, care must be taken that the software versions of these tools are closely aligned, so that they remain interoperable.

This is a common source of error or delay for newcomers who want to quickly try out a novel tool—and a source of frustration for the teacher who needs to oversee the installation and deployment on hundreds of student machines every course.

Veins is therefore also made available as a virtual appliance, *Instant Veins*, which can be installed with a single click—and run independent of the operating system of the target machine. Its only prerequisite is pre-installed virtualization software, such as the open-source tool **Oracle VM VirtualBox** or any other tool that can read the Open Virtual Appliance (*.ova*) file format, such as the popular **VMware Workstation Player**. Instant Veins already contains compatible versions of all of Veins, the INET Framework, and Veins_INET to link the two (and, as a special download, also of SimuLTE)—along with OMNeT++ and SUMO.

On most machines, all that is needed is to double-click the downloaded *.ova* virtual appliance file to import it into the user's virtualization tool, from where it can then be launched directly—though some machines might have a slightly more involved installation procedure for *.ova* files, e.g., requiring the user to confirm opening the file first. After booting the virtual appliance and logging in, all needed tools can be started from the graphical shell (by clicking their respective launch



```
//
// Documentation for these modules is at http://veins.car2x.org/
//
```

Fig. 6.15 Screenshot of the Instant Veins virtual appliance, showing the OMNeT++ IDE after clicking on the OMNeT++ launch icon

icon). For example, after clicking on the OMNeT++ launch icon, a user is soon presented with the usual OMNeT++ IDE, which already has a workspace open that includes all four mentioned simulation libraries—ready to run (Fig. 6.15).

Instant Veins is built on fully open-source tools, most importantly Debian GNU/Linux as its base (taking care to only include re-distributable software with the base installation). This makes Instant Veins particularly useful in the classroom: Aside from getting students up and running within as little as a minute, the virtual appliance file can be freely shared with and among students.

Acknowledgements The authors are grateful to the community surrounding Veins, the many people who keep contributing their time and smarts to its continuous improvement. We particularly acknowledge the research labs at Univ. Paderborn, Univ. Erlangen-Nuremberg, Univ. Trento, TUMCREATE Singapore, Univ. Sydney, UCLA, Univ. Innsbruck, Univ. Luxembourg, TH Ingolstadt, Fraunhofer, TU Berlin, Carnegie Mellon University, and the German Aerospace Center.

The author D. Eckhoff was financially supported by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

References

1. Ali, A., Garcia, G., Martinet, P.: The flatbed platoon towing model for safe and dense platooning on highways. IEEE Intell. Transp. Syst. Mag. **7**(1), 58–68 (2015). https://doi.org/10.1109/MITS.2014.2328670
2. Aramrattana, M., Larsson, T., Jansson, J., Nåbo, A.: A simulation framework for cooperative intelligent transport systems testing and evaluation. Transport. Res. F: Traffic Psychol. Behav. (2017). https://doi.org/10.1016/j.trf.2017.08.004
3. Bedogni, L., Bononi, L., Di Felice, M., D'Elia, A., Mock, R., Morandi, F., Rondelli, S., Salmon Cinotti, T., Vergari, F.: An integrated simulation framework to model electric vehicles operations and services. IEEE Trans. Veh. Technol. **65**(8) (2015). https://doi.org/10.1109/TVT.2015.2453125
4. Bedogni, L., Gramaglia, M., Vesco, A., Fiore, M., Härri, J., Ferrero, F.: The Bologna ringway dataset: improving road network conversion in SUMO and validating urban mobility via navigation services. IEEE Trans. Veh. Technol. **64**(12), 5464–5476 (2015). https://doi.org/10.1109/TVT.2015.2475608
5. Berndt, H., Wender, S., Dietmayer, K.: Driver braking behavior during intersection approaches and implications for warning strategies for driver assistant systems. In: IEEE Intelligent Vehicles Symposium (IV'07), pp. 245–251. IEEE, Istanbul (2007). https://doi.org/10.1109/IVS.2007.4290122
6. Bieker, L., Krajzewicz, D., Morra, A.P., Michelacci, C., Cartolano, F.: Traffic simulation for all: a real world traffic scenario from the city of Bologna. In: SUMO User Conference 2014, pp. 19–26. Deutsches Zentrum für Luft - und Raumfahrt e.V., Berlin (2014). https://doi.org/10.1007/978-3-319-15024-6_4
7. Bonnet, C., Fritz, H.: Fuel consumption reduction in a platoon: experimental results with two electronically coupled trucks at close spacing. In: Future Transportation Technology Conference. SAE, Costa Mesa (2001)
8. Brummer, A., German, R., Djanatliev, A.: On the necessity of three-dimensional considerations in vehicular network simulation. In: 14th IEEE/IFIP Conference on Wireless on demand Network Systems and Services (WONS 2018), Isola 2000, pp. 75–82. IEEE, Isola (2018). https://doi.org/10.23919/WONS.2018.8311665
9. Codecá, L., Härri, J.: Towards multimodal mobility simulation of C-ITS: the monaco SUMO traffic scenario. In: 9th IEEE Vehicular Networking Conference (VNC 2017), pp. 97–100. IEEE, Torino (2017). https://doi.org/10.1109/VNC.2017.8275627
10. Codeca, L., Frank, R., Engel, T.: Luxembourg SUMO traffic (LuST) scenario: 24 hours of mobility for vehicular networking research. In: 7th IEEE Vehicular Networking Conference (VNC 2015). IEEE, Kyoto (2015). https://doi.org/10.1109/VNC.2015.7385539
11. Dávila, A., Nombela, M.: Sartre - safe road trains for the environment reducing fuel consumption through lower aerodynamic drag coefficient. In: 25th SAE Brasil International Congress and Display. SAE Brasil, São Paulo (2011)
12. Eckhoff, D., Sommer, C.: A multi-channel IEEE 1609.4 and 802.11p EDCA model for the Veins framework. In: 5th ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2012): 5th ACM/ICST International Workshop on OMNeT++ (OMNeT++ 2012), Poster Session. ACM, Desenzano (2012)
13. Eckhoff, D., Sommer, C.: Simulative performance evaluation of vehicular networks. In: Chen, W. (ed.) Vehicular Communications and Networks: Architectures, Protocols, Operation and Deployment, pp. 255–274. Woodhead Publishing, Sawston (2015). https://doi.org/10.1016/B978-1-78242-211-2.00012-X
14. Eckhoff, D., Sommer, C.: Readjusting the privacy goals in vehicular ad-hoc networks: a safety-preserving solution using non-overlapping time-slotted pseudonym pools. Elsevier Comput. Commun. **122**, 118–128 (2018). https://doi.org/10.1016/j.comcom.2018.03.006

15. Eckhoff, D., Sommer, C., Dressler, F.: On the necessity of accurate IEEE 802.11p models for IVC protocol simulation. In: 75th IEEE Vehicular Technology Conference (VTC2012-Spring), pp. 1–5. IEEE, Yokohama (2012). https://doi.org/10.1109/VETECS.2012.6240064
16. Eckhoff, D., Halmos, B., German, R.: Potentials and limitations of green light optimal speed advisory systems. In: 5th IEEE Vehicular Networking Conference (VNC 2013), pp. 103–110. IEEE, Boston (2013). https://doi.org/10.1109/VNC.2013.6737596
17. Eckhoff, D., Sofra, N., German, R.: A performance study of cooperative awareness in ETSI ITS G5 and IEEE WAVE. In: 10th IEEE/IFIP Conference on Wireless on demand Network Systems and Services (WONS 2013), pp. 196–200. IEEE, Banff (2013). https://doi.org/10.1109/WONS.2013.6578347
18. Eckhoff, D., Brummer, A., Sommer, C.: On the impact of antenna patterns on VANET simulation. In: 8th IEEE Vehicular Networking Conference (VNC 2016), pp. 17–20. IEEE, Columbus (2016). https://doi.org/10.1109/VNC.2016.7835925
19. Emara, K.: Poster: PREXT: privacy extension for veins VANET simulator. In: 8th IEEE Vehicular Networking Conference (VNC 2016), Poster Session. IEEE, Columbus (2016). https://doi.org/10.1109/VNC.2016.7835979
20. Giordano, G., Segata, M., Blanchini, F., Lo Cigno, R.: A joint network/control design for cooperative automatic driving. In: 9th IEEE Vehicular Networking Conference (VNC 2017), pp. 167–174. IEEE, Torino (2017)
21. Hagenauer, F., Dressler, F., Sommer, C.: A simulator for heterogeneous vehicular networks. In: 6th IEEE Vehicular Networking Conference (VNC 2014), Poster Session, pp. 185–186. IEEE, Paderborn (2014). https://doi.org/10.1109/VNC.2014.7013339
22. Hassan, M.I., Vu, H.L., Sakurai, T.: Performance analysis of the IEEE 802.11 MAC protocol for DSRC safety applications. IEEE Trans. Veh. Technol. **60**(8), 3882–3896 (2011). https://doi.org/10.1109/TVT.2011.2162755
23. Heinovski, J., Klingler, F., Dressler, F., Sommer, C.: A simulative analysis of the performance of IEEE 802.11p and ARIB STD-T109. Elsevier Comput. Commun. **122**, 84–92 (2018). https://doi.org/10.1016/j.comcom.2018.03.016
24. IEEE: IEEE standard for Wireless Access in Vehicular Environments (WAVE) - multi-channel operation. Std 1609.4-2016. IEEE, Piscataway (2016)
25. IEEE: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Std 802.11-2016. IEEE, Piscataway (2016)
26. Joerer, S.: Improving intersection safety with inter-vehicle communication. Phd thesis (dissertation), University of Innsbruck (2016)
27. Joerer, S., Dressler, F., Sommer, C.: Comparing apples and oranges? Trends in IVC simulations. In: 9th ACM International Workshop on Vehicular Internetworking (VANET 2012), pp. 27–32. ACM, Low Wood Bay (2012). https://doi.org/10.1145/2307888.2307895
28. Joerer, S., Segata, M., Bloessl, B., Lo Cigno, R., Sommer, C., Dressler, F.: To crash or not to crash: estimating its likelihood and potentials of Beacon-based IVC systems. In: 4th IEEE Vehicular Networking Conference (VNC 2012), pp. 25–32. IEEE, Seoul (2012). https://doi.org/10.1109/VNC.2012.6407441
29. Joerer, S., Segata, M., Bloessl, B., Lo Cigno, R., Sommer, C., Dressler, F.: A vehicular networking perspective on estimating vehicle collision probability at intersections. IEEE Trans. Veh. Technol. **63**(4), 1802–1812 (2014). https://doi.org/10.1109/TVT.2013.2287343
30. Jootel, P.S.: SAfe Road TRains for the Environment. Final project report, SARTRE project (2012)
31. Kornek, D., Schack, M., Slottke, E., Klemp, O., Rolfes, I., Kürner, T.: Effects of antenna characteristics and placements on a vehicle-to-vehicle channel scenario. In: IEEE International Conference on Communications (ICC 2010), Workshops. IEEE, Capetown (2010). https://doi.org/10.1109/ICCW.2010.5503935
32. Krauß, S., Wagner, P., Gawron, C.: Metastable states in a microscopic model of traffic flow. Phys. Rev. E **55**(5), 5597–5602 (1997). https://doi.org/10.1103/PhysRevE.55.5597

33. Kunze, R., Ramakers, R., Henning, K., Jeschke, S.: Organization and operation of electronically coupled truck platoons on German motorways. In: Automation, Communication and Cybernetics in Science and Engineering 2009/2010, pp. 427–439. Springer, Berlin (2011)
34. Kwoczek, A., Raida, Z., Láčík, J., Pokorný, M., Puskely, J., Vágner, P.: Influence of car panorama glass roofs on Car2car communication. In: 3rd IEEE Vehicular Networking Conference (VNC 2011), Poster Session, pp. 246–251. IEEE, Amsterdam (2011). https://doi.org/10.1109/VNC.2011.6117107
35. Larson, J., Liang, K.Y., Johansson, K.H.: A distributed framework for coordinated heavy-duty vehicle platooning. IEEE Trans. Intell. Transp. Syst. **16**(1), 419–429 (2015). https://doi.org/10.1109/TITS.2014.2320133
36. Leonor, N.R., Caldeirinha, R.F.S., Sánchez, M.G., Fernandes, T.R.: A three-dimensional directive antenna pattern interpolation method. IEEE Antennas Wirel. Propag. Lett. **15**, 881–884 (2016). https://doi.org/10.1109/LAWP.2015.2478962
37. Memedi, A., Tsai, H.M., Dressler, F.: Impact of realistic light radiation pattern on vehicular visible light communication. In: IEEE Global Telecommunications Conference (GLOBECOM 2017). IEEE, Singapore (2017). https://doi.org/10.1109/GLOCOM.2017.8253979
38. Milanés, V., Shladover, S.E., Spring, J., Nowakowski, C., Kawazoe, H., Nakamura, M.: Cooperative adaptive cruise control in real traffic situations. IEEE Trans. Intell. Transp. Syst. **15**(1), 296–305 (2014). https://doi.org/10.1109/TITS.2013.2278494
39. Nardini, G., Virdis, A., Stea, G.: Modeling X2 backhauling for LTE-advanced and assessing its effect on CoMP coordinated scheduling. In: 1st International Workshop on Link- and System Level Simulations (IWSLS 2016). IEEE, Vienna (2016). https://doi.org/10.1109/IWSLS.2016.7801582
40. Ploeg, J., Scheepers, B., van Nunen, E., van de Wouw, N., Nijmeijer, H.: Design and experimental evaluation of cooperative adaptive cruise control. In: IEEE International Conference on Intelligent Transportation Systems (ITSC 2011), pp. 260–265. IEEE, Washington (2011). https://doi.org/10.1109/ITSC.2011.6082981
41. Rajamani, R.: Vehicle Dynamics and Control, 2nd edn. Springer, Cham (2012)
42. Rajamani, R., Tan, H.S., Law, B.K., Zhang, W.B.: Demonstration of integrated longitudinal and lateral control for the operation of automated vehicles in platoons. IEEE Trans. Control Syst. Technol. **8**(4), 695–708 (2000). https://doi.org/10.1109/87.852914
43. Riebl, R., Günther, H.J., Facchi, C., Wolf, L.: Artery - extending veins for VANET applications. In: 4th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS 2015). IEEE, Budapest (2015). https://doi.org/10.1109/MTITS.2015.7223293
44. Riebl, R., Monz, M., Varga, S., Maglaras, L., Janicke, H., Al-Bayatti, A.H., Facchi, C.: Improved security performance for VANET simulations. In: 4th IFAC Symposium on Telematics Applications (TA 2016), vol. 49, pp. 233–238. Elsevier, Porto Alwegre (2016). https://doi.org/10.1016/j.ifacol.2016.11.173
45. Santini, S., Salvi, A., Valente, A.S., Pescapè, A., Segata, M., Lo Cigno, R.: A consensus-based approach for platooning with inter-vehicular communications and its validation in realistic scenarios. IEEE Trans. Veh. Technol. **66**(3), 1985–1999 (2017). https://doi.org/10.1109/TVT.2016.2585018
46. Segata, M.: Safe and efficient communication protocols for platooning control. Ph.D. thesis (dissertation), University of Innsbruck (2016)
47. Segata, M.: Platooning in SUMO: an open source implementation. In: SUMO User Conference 2017, pp. 51–62. DLR, Berlin (2017)
48. Segata, M., Joerer, S., Bloessl, B., Sommer, C., Dressler, F., Lo Cigno, R.: PLEXE: a platooning extension for Veins. In: 6th IEEE Vehicular Networking Conference (VNC 2014), pp. 53–60. IEEE, Paderborn (2014). https://doi.org/10.1109/VNC.2014.7013309
49. Segata, M., Bloessl, B., Joerer, S., Sommer, C., Gerla, M., Lo Cigno, R., Dressler, F.: Towards communication strategies for platooning: simulative and experimental evaluation. IEEE Trans. Veh. Technol. **64**(12), 5411–5423 (2015). https://doi.org/10.1109/TVT.2015.2489459

50. Segata, M., Dressler, F., Lo Cigno, R.: Jerk beaconing: a dynamic approach to platooning. In: 7th IEEE Vehicular Networking Conference (VNC 2015), pp. 135–142. IEEE, Kyoto (2015). https://doi.org/10.1109/VNC.2015.7385560
51. Shladover, S.: PATH at 20 – history and major milestones. In: IEEE Intelligent Transportation Systems Conference (ITSC 2006), pp. 22–29. Toronto (2006). https://doi.org/10.1109/ITSC.2006.1706710
52. Sommer, C., Dressler, F.: Using the right two-ray model? A measurement based evaluation of PHY models in VANETs. In: 17th ACM International Conference on Mobile Computing and Networking (MobiCom 2011), Poster Session. ACM, Las Vegas (2011)
53. Sommer, C., Dressler, F.: Vehicular Networking. Cambridge University Press, Cambridge (2014). https://doi.org/10.1017/CBO9781107110649
54. Sommer, C., Krul, R., German, R., Dressler, F.: Emissions vs. travel time: simulative evaluation of the environmental impact of ITS. In: 71st IEEE Vehicular Technology Conference (VTC2010-Spring), pp. 1–5. IEEE, Taipei (2010). https://doi.org/10.1109/VETECS.2010.5493943
55. Sommer, C., Eckhoff, D., German, R., Dressler, F.: A computationally inexpensive empirical model of IEEE 802.11p radio shadowing in urban environments. In: 8th IEEE/IFIP Conference on Wireless on Demand Network Systems and Services (WONS 2011), pp. 84–90. IEEE, Bardonecchia (2011). https://doi.org/10.1109/WONS.2011.5720204
56. Sommer, C., German, R., Dressler, F.: Bidirectionally coupled network and road traffic simulation for improved IVC analysis. IEEE Trans. Mob. Comput. **10**(1), 3–15 (2011). https://doi.org/10.1109/TMC.2010.133
57. Sommer, C., Eckhoff, D., Dressler, F.: IVC in cities: signal attenuation by buildings and how parked cars can improve the situation. IEEE Trans. Mob. Comput. **13**(8), 1733–1745 (2014). https://doi.org/10.1109/TMC.2013.80
58. Sommer, C., Joerer, S., Segata, M., Tonguz, O.K., Lo Cigno, R., Dressler, F.: How shadowing hurts vehicular communications and how dynamic beaconing can help. IEEE Trans. Mob. Comput. **14**(7), 1411–1421 (2015). https://doi.org/10.1109/TMC.2014.2362752
59. Torrent-Moreno, M., Schmidt-Eisenlohr, F., Füßler, H., Hartenstein, H.: Effects of a realistic channel model on packet forwarding in vehicular ad hoc networks. In: IEEE Wireless Communications and Networking Conference (WCNC 2006), pp. 385–391. IEEE, Las Vegas (2006). https://doi.org/10.1109/WCNC.2006.1683495
60. Treiber, M., Hennecke, A., Helbing, D.: Congested traffic states in empirical observations and microscopic simulations. Phys. Rev. E **62**(2), 1805–1824 (2000)
61. Virdis, A., Stea, G., Nardini, G.: SimuLTE - a modular system-level simulator for LTE/LTE-A networks based on OMNeT++. In: 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2014). Vienna (2014)
62. Virdis, A., Nardini, G., Stea, G.: Modeling unicast device-to-device communications with SimuLTE. In: 2016 1st International Workshop on Link- and System Level Simulations (IWSLS), pp. 1–8. IEEE, Vienna (2016)
63. Virdis, A., Stea, G., Nardini, G.: Simulating LTE/LTE-advanced networks with SimuLTE. In: Obaidat, S.M., Ören, T., Kacprzyk, J., Filipe, J. (eds.) Simulation and Modeling Methodologies, No. 402. Advances in Intelligent Systems and Computing, pp. 83–105. Springer, Cham (2016)
64. Wessel, K., Swigulski, M., Köpke, A., Willkomm, D.: MiXiM – the physical layer: an architecture overview. In: 2nd ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2009): 2nd ACM/ICST International Workshop on OMNeT++ (OMNeT++ 2009). ACM, Rome (2009)
65. Zardosht, B., Beauchemin, S.S., Bauer, M.A.: A predictive accident-duration based decision-making module for rerouting in environments with V2V communication. Elsevier J. Traffic and Transp. Eng. (2017). https://doi.org/10.1016/j.jtte.2017.07.007