Practical Manual of

# SUMO/MATLAB/VEINS/INET/OMNET++ PROGRAMMING AND INTERFACING

for V2X Simulation with Standard Protocols

TECHNICAL REPORT

Author: Tamás Ormándi [ormandi.tamas[at]edu.bme.hu] v1.4
`http://traffic.bme.hu`

Budapest University of Technology and Economics
Department of Control for Transportation and Vehicle Systems

November 14, 2021

# Contents

## 1 Introduction

This documentation provides helpful information about the following topics:

- Usage of OMNeT++ network simulator with Veins and INET frameworks
- Implementation of the ITS Facilities Layer (simulation of standard CAM/DENM messages) in OMNeT++
- Using SUMO in multi-client mode together with OMNeT++ and Matlab/Simulink
- Interfacing Matlab/Simulink with OMNeT++ using Named Shared Memory
- Implementing Road Side Units in the Veins-INET project
- Extension with the Simu5G framework with TraCI interface access

It also provides a step-by-step guide for the installation of the aforementioned software to be able to run a co-simulation and provides example codes. In addition, some common (and undocumented) bugs and best practices are included in this documentation.

## 2 Software Installation

Software needed:

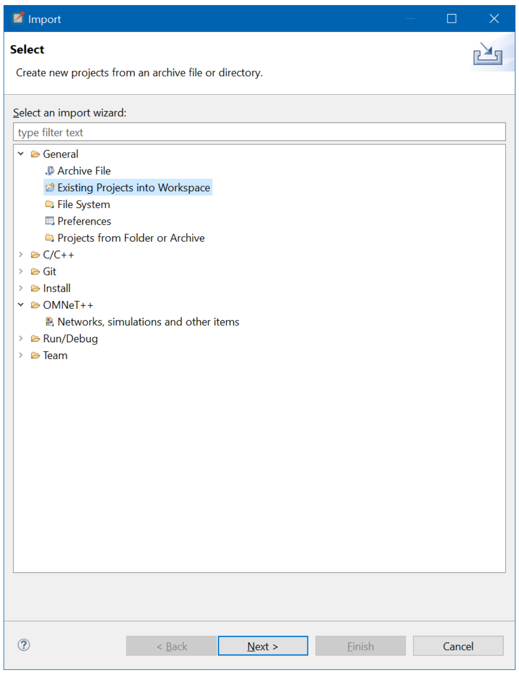
1. SUMO Traffic Simulator (Version: 1.7.0) `https://www.eclipse.org/sumo/`
2. OMNeT++ (Version: 5.6.2) `https://omnetpp.org/download/`
3. Veins framework (Version: 5.0) `https://veins.car2x.org/download/`
4. INET framework (Version: 4.2.2) `https://inet.omnetpp.org/Download.html`
5. Simu5G framework (Version: 1.1.0) `https://github.com/Unipisa/Simu5G/archive/refs/tags/v1.1.0.zip`

Using different versions can cause errors, which are hard to fix. Software should be downloaded and unpacked to the recommended paths.
⚠ The path should not contain any special characters or whitespaces!

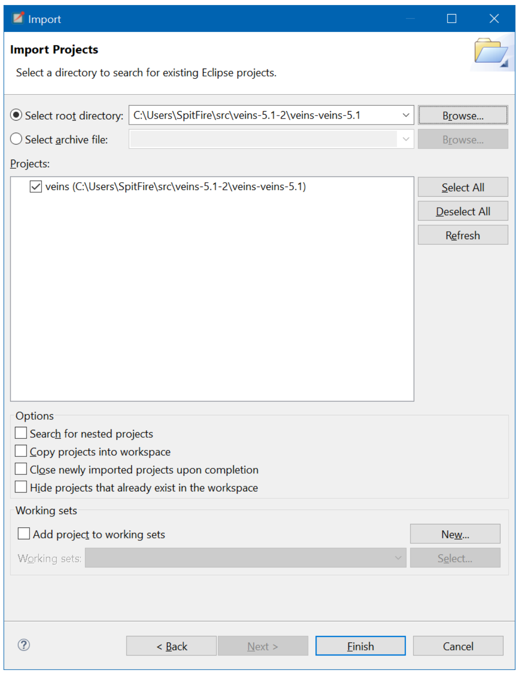
1. C:\Users\user\src\sumo-1.7.0
2. C:\Users\user\src\omnetpp-5.6.2
3. C:\Users\user\src\veins-5.0

Following the Veins tutorial steps, first SUMO has to be installed. Then OMNeT++ can be downloaded. To install OMNeT++, it has to be unpacked to its path and the *mingwenv.cmd* has to be executed. It opens a console. First the software has to be built by using the console, by executing command *./configure* and then the command *make*. If everything went right, the *C\Users\user\src\omnetpp-5.6.2\bin\omnetpp* build should be ready.

To start OMNeT++ *mingwenv.cmd* has to be started and the command *omnetpp* has to be executed. OMNeT++ lets the user choose the workspace at every start but it can be used as the default samples.


Import
Select
Create new projects from an archive file or directory.
Select an import wizard:
type filter text
General
Archive File
Existing Projects into Workspace
File System
Preferences
Projects from Folder or Archive
C/C++
Git
Install
OMNeT++
Networks, simulations and other items
Run/Debug
Team
< Back
Next >
Finish
Cancel

(a) Import type.



Select root directory: C:\Users\SpitFire\src\veins-5.1-2\veins-veins-5.1
Select archive file:
Projects:
☑ veins (C:\Users\SpitFire\src\veins-5.1-2\veins-veins-5.1)
☐ Search for nested projects
☐ Copy projects into workspace
☐ Close newly imported projects upon completion
☐ Hide projects that already exist in the workspace
☐ Add project to working sets
Working sets:

(b) Loading the veins folder.

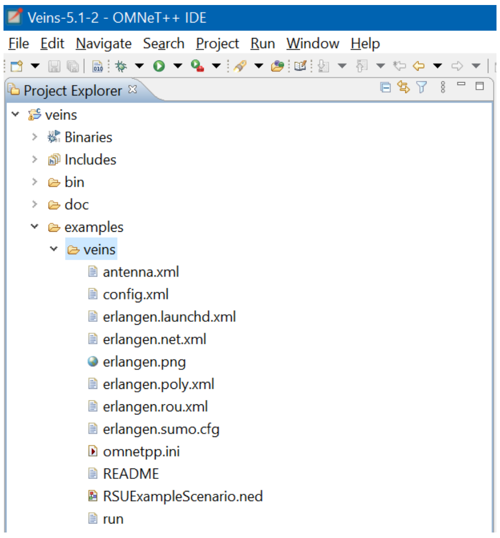
Figure 1: Importing an existing project.

After Veins is downloaded and unpacked to the correct path,it can be imported to OMNeT++ by using *File > Import > General: Existing Projects into Workspace* and select the folder of veins by browsing it.

## 3 OMNeT++ Settings

### 3.1 Running the example simulation

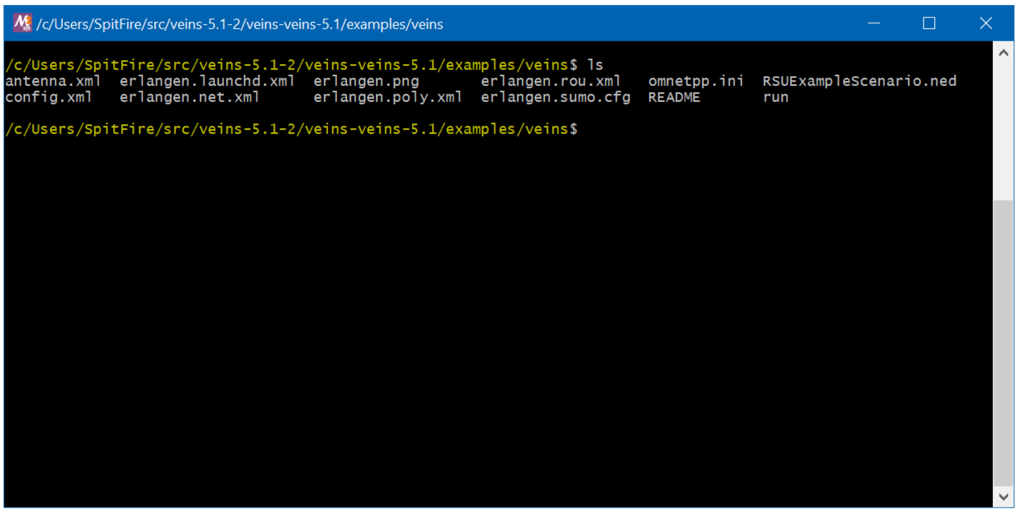
After importing the Veins example project successfully to OMNeT++, navigate to the veins folder under examples in the Project Explorer.



veins
Binaries
Includes
bin
doc
examples
veins
antenna.xml
config.xml
erlangen.launchd.xml
erlangen.net.xml
erlangen.png
erlangen.poly.xml
erlangen.rou.xml
erlangen.sumo.cfg
omnetpp.ini
README
RSUExampleScenario.ned
run

Figure 2: Veins example folder.

The configuration files of the project are located here. To start the example, the *mingwenv* console has to be used again to navigate to the veins example folder. It can be done by the default way, using the windows console commands to change folders using command *cd C:/Users/User/src/veins-5.1/examples/veins*. By executing the *ls* command, a list of files can be seen to make sure its the right folder with the files of the example project.



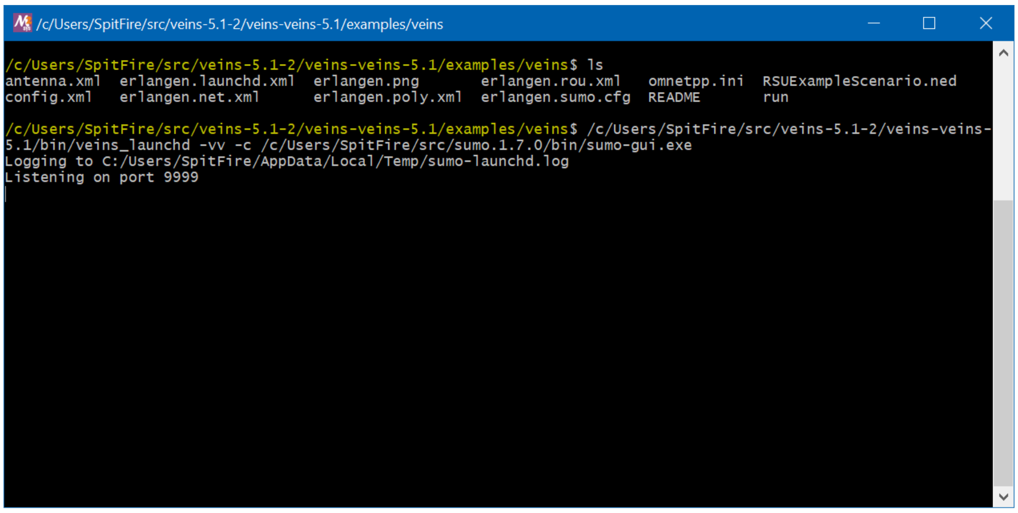
```
/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins
/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins$ ls
antenna.xml  erlangen.launchd.xml  erlangen.png       erlangen.rou.xml   omnetpp.ini  RSUExampleScenario.ned
config.xml   erlangen.net.xml      erlangen.poly.xml  erlangen.sumo.cfg  README       run

/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins$
```

Figure 3: Console in the folder with list of files.

## 3.2 Example with Veins Launchd

The default example runs with the help of a python script called launchd. To execute the example, the command */c/Users/User/src/veins-5.1-2/bin/veins_launchd -vv -c /c/Users/User/src/sumo.1.7.0/bin/sumo-gui.exe* should be run.

(Note: sumo.exe can be used instead of sumo-gui.exe, but then SUMO will run without the graphical interface.)



```
/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins
/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins$ ls
antenna.xml  erlangen.launchd.xml  erlangen.png       erlangen.rou.xml   omnetpp.ini  RSUExampleScenario.ned
config.xml   erlangen.net.xml      erlangen.poly.xml  erlangen.sumo.cfg  README       run

/c/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins$ /c/Users/SpitFire/src/veins-5.1-2/veins-veins-
5.1/bin/veins_launchd -vv -c /c/Users/SpitFire/src/sumo.1.7.0/bin/sumo-gui.exe
Logging to C:/Users/SpitFire/AppData/Local/Temp/sumo-launchd.log
Listening on port 9999
```

Figure 4: Running the launchd script.

It will start the Launchd python script. If the text *Listening on port 9999* appears in the console, the simulation is ready to go. The simulation can be run by building the example in OMNeT++.

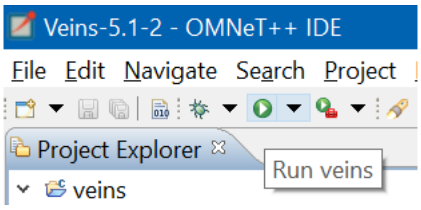
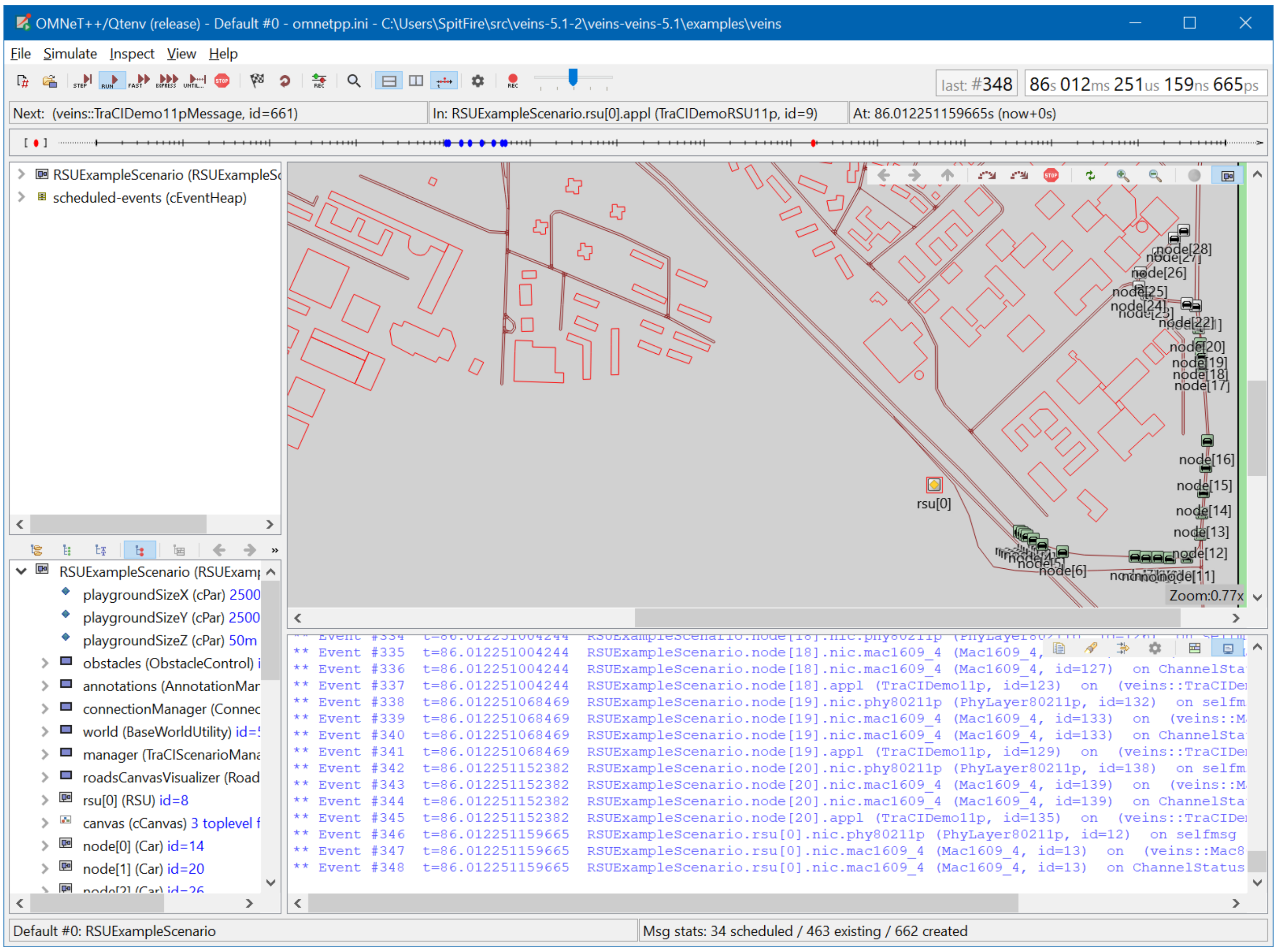


Figure 5: Building and running Veins example.

After it is built, the simulation can be started in a new window, which is the QT environment.


OMNeT++/Qtenv (release) - Default #0 - omnetpp.ini - C:\Users\SpitFire\src\veins-5.1-2\veins-veins-5.1\examples\veins
File Simulate Inspect View Help
last: #348
86s 012ms 251us 159ns 665ps
RSUExampleScenario (RSUExampleSc
scheduled-events (cEventHeap)
rsu[0]
Zoom:0.77x
RSUExampleScenario (RSUExamp
playgroundSizeX (cPar) 2500
playgroundSizeY (cPar) 2500
playgroundSizeZ (cPar) 50m
obstacles (ObstacleControl)
annotations (AnnotationMar
connectionManager (Connec
world (BaseWorldUtility) id=
manager (TraCIScenarioMana
roadsCanvasVisualizer (Road
rsu[0] (RSU) id=8
canvas (cCanvas) 3 toplevel
node[0] (Car) id=14
node[1] (Car) id=20

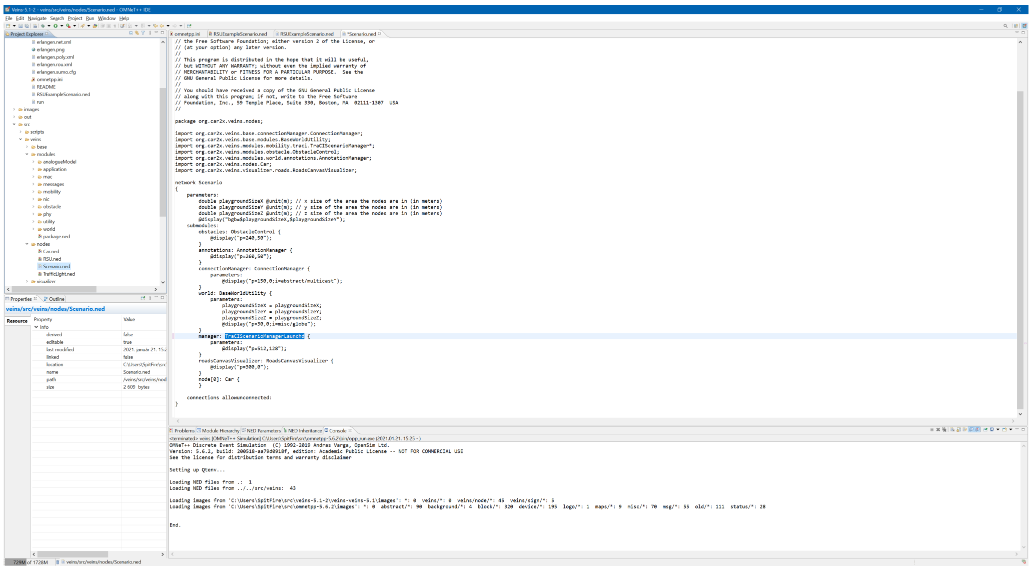
```
Next:  (veins::TraCIDemo11pMessage, id=661)
In: RSUExampleScenario.rsu[0].appl (TraCIDemoRSU11p, id=9)
At: 86.012251159665s (now+0s)

** Event #335  t=86.012251004244  RSUExampleScenario.node[18].nic.mac1609_4 (Mac1609_4,
** Event #336  t=86.012251004244  RSUExampleScenario.node[18].nic.mac1609_4 (Mac1609_4, id=127)  on ChannelSta
** Event #337  t=86.012251004244  RSUExampleScenario.node[18].appl (TraCIDemo11p, id=123)  on  (veins::TraCIDe
** Event #338  t=86.012251068469  RSUExampleScenario.node[19].nic.phy80211p (PhyLayer80211p, id=132)  on selfm
** Event #339  t=86.012251068469  RSUExampleScenario.node[19].nic.mac1609_4 (Mac1609_4, id=133)  on  (veins::M
** Event #340  t=86.012251068469  RSUExampleScenario.node[19].nic.mac1609_4 (Mac1609_4, id=133)  on ChannelSta
** Event #341  t=86.012251068469  RSUExampleScenario.node[19].appl (TraCIDemo11p, id=129)  on  (veins::TraCIDe
** Event #342  t=86.012251152382  RSUExampleScenario.node[20].nic.phy80211p (PhyLayer80211p, id=138)  on selfm
** Event #343  t=86.012251152382  RSUExampleScenario.node[20].nic.mac1609_4 (Mac1609_4, id=139)  on  (veins::M
** Event #344  t=86.012251152382  RSUExampleScenario.node[20].nic.mac1609_4 (Mac1609_4, id=139)  on ChannelSta
** Event #345  t=86.012251152382  RSUExampleScenario.node[20].appl (TraCIDemo11p, id=135)  on  (veins::TraCIDe
** Event #346  t=86.012251159665  RSUExampleScenario.rsu[0].nic.phy80211p (PhyLayer80211p, id=12)  on selfmsg
** Event #347  t=86.012251159665  RSUExampleScenario.rsu[0].nic.mac1609_4 (Mac1609_4, id=13)  on  (veins::Mac8
** Event #348  t=86.012251159665  RSUExampleScenario.rsu[0].nic.mac1609_4 (Mac1609_4, id=13)  on ChannelStatus

Default #0: RSUExampleScenario
Msg stats: 34 scheduled / 463 existing / 662 created
```

Figure 6: The simulation.

## 3.3 Veins Launchd and its exclusion from the simulation

The Veins Launchd script is basically a Proxy TCP server, which establishes the communication between OMNeT++ and SUMO. It has its own port handling and it can handle multiple connections for parallel simulations. This script is not needed, if only one instance of the simulation is needed and it has to be controlled by Matlab/Simulink with OMNeT++ simultaneously.

To get rid of this python script, the Scenario.ned file has to be modified. OMNeT++ relies on its own NED (Network Description) files and on C++ files. Open the Scenario.ned file, which is located in the Project Explorer in *src/veins/nodes/*. Find the line which defines the manager of the simulation, and modify *TraCIScenarioManagerLaunchd* to *TraCIScenarioManager*.

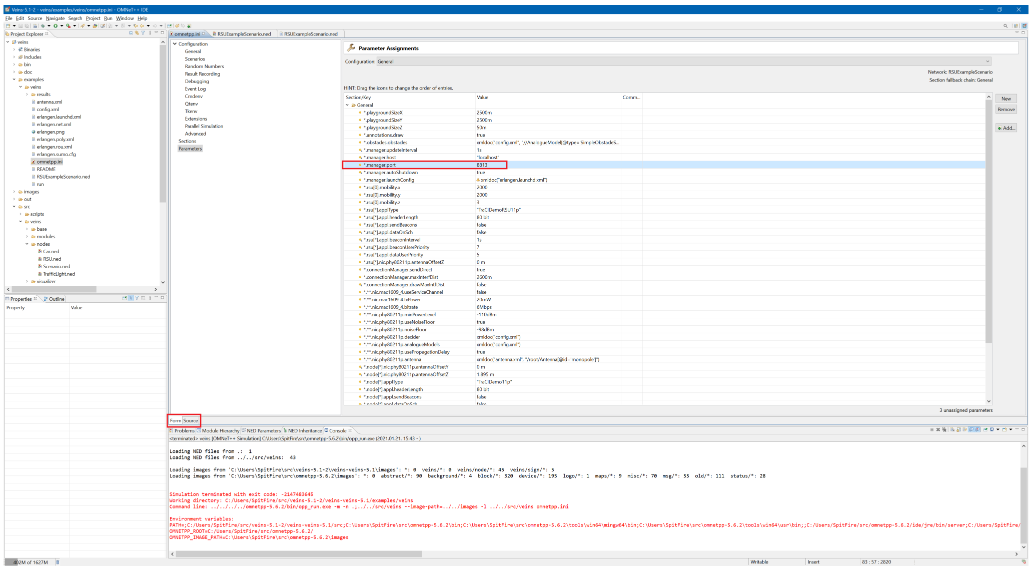


```
// the Free Software Foundation; either version 2 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
//

package org.car2x.veins.nodes;

import org.car2x.veins.base.connectionManager.ConnectionManager;
import org.car2x.veins.base.modules.BaseWorldUtility;
import org.car2x.veins.modules.mobility.traci.TraCIScenarioManager*;
import org.car2x.veins.modules.obstacle.ObstacleControl;
import org.car2x.veins.modules.world.annotations.AnnotationManager;
import org.car2x.veins.nodes.Car;
import org.car2x.veins.visualizer.roads.RoadsCanvasVisualizer;

network Scenario
{
    parameters:
        double playgroundSizeX @unit(m); // x size of the area the nodes are in (in meters)
        double playgroundSizeY @unit(m); // y size of the area the nodes are in (in meters)
        double playgroundSizeZ @unit(m); // z size of the area the nodes are in (in meters)
        @display("bgb=$playgroundSizeX,$playgroundSizeY");
    submodules:
        obstacles: ObstacleControl {
            @display("p=240,50");
        }
        annotations: AnnotationManager {
            @display("p=260,50");
        }
        connectionManager: ConnectionManager {
            parameters:
                @display("p=150,0;i=abstract/multicast");
        }
        world: BaseWorldUtility {
            parameters:
                playgroundSizeX = playgroundSizeX;
                playgroundSizeY = playgroundSizeY;
                playgroundSizeZ = playgroundSizeZ;
                @display("p=30,0;i=misc/globe");
        }
        manager: TraCIScenarioManagerLaunchd {
            parameters:
                @display("p=512,128");
        }
        roadsCanvasVisualizer: RoadsCanvasVisualizer {
            @display("p=300,0");
        }
        node[0]: Car {
        }

    connections allowunconnected:
}
```

```
<terminated> veins [OMNeT++ Simulation] C:\Users\SpitFire\src\omnetpp-5.6.2\bin/opp_run.exe (2021.01.21. 15:25 - )
OMNeT++ Discrete Event Simulation  (C) 1992-2019 Andras Varga, OpenSim Ltd.
Version: 5.6.2, build: 200518-aa79d0918f, edition: Academic Public License -- NOT FOR COMMERCIAL USE
See the license for distribution terms and warranty disclaimer

Setting up Qtenv...

Loading NED files from .:  1
Loading NED files from ../../src/veins:  43

Loading images from 'C:\Users\SpitFire\src\veins-5.1-2\veins-veins-5.1\images': *: 0  veins/*: 0  veins/node/*: 45  veins/sign/*: 5
Loading images from 'C:\Users\SpitFire\src\omnetpp-5.6.2\images': *: 0  abstract/*: 90  background/*: 4  block/*: 320  device/*: 195  logo/*: 1  maps/*: 9  misc/*: 70  msg/*: 55  old/*: 111  status/*: 28


End.
```

Figure 7: TraCIScenarioManager setup.

The *TraCIScenarioManager* is a C++ script which is responsible for the TraCI connection directly between SUMO and OMNeT++. To set up the TraCI connection, open *omnetpp.ini* in */examples/veins/*. Choosing the Parameters option, a lot of settings can be set up for the simulation. It contains the TraCI settings, like host, port etc.. Below the main window containing the parameters, the editor can be switched to source mode, so editing can be done by code writing or by using the GUI form.



Parameter Assignments

Configuration: General

Network: RSUExampleScenario
Section fallback chain: General

HINT: Drag the icons to change the order of entries.

| Section/Key | Value | Comm... |
|---|---|---|
| General | | |
| *.playgroundSizeX | 2500m | |
| *.playgroundSizeY | 2500m | |
| *.playgroundSizeZ | 50m | |
| *.annotations.draw | true | |
| *.obstacles.obstacles | xmldoc("config.xml", "//AnalogueModel[@type='SimpleObstacleS... | |
| *.manager.updateInterval | 1s | |
| *.manager.host | "localhost" | |
| *.manager.port | 8813 | |
| *.manager.autoShutdown | true | |
| *.manager.launchConfig | xmldoc("erlangen.launchd.xml") | |
| *.rsu[0].mobility.x | 2000 | |
| *.rsu[0].mobility.y | 2000 | |
| *.rsu[0].mobility.z | 3 | |
| *.rsu[*].applType | "TraCIDemoRSU11p" | |
| *.rsu[*].appl.headerLength | 80 bit | |
| *.rsu[*].appl.sendBeacons | false | |
| *.rsu[*].appl.dataOnSch | false | |
| *.rsu[*].appl.beaconInterval | 1s | |
| *.rsu[*].appl.beaconUserPriority | 7 | |
| *.rsu[*].appl.dataUserPriority | 5 | |
| *.rsu[*].nic.phy80211p.antennaOffsetZ | 0 m | |
| *.connectionManager.sendDirect | true | |
| *.connectionManager.maxInterfDist | 2600m | |
| *.connectionManager.drawMaxIntfDist | false | |
| *.**.nic.mac1609_4.useServiceChannel | false | |
| *.**.nic.mac1609_4.txPower | 20mW | |
| *.**.nic.mac1609_4.bitrate | 6Mbps | |
| *.**.nic.phy80211p.minPowerLevel | -110dBm | |
| *.**.nic.phy80211p.useNoiseFloor | true | |
| *.**.nic.phy80211p.noiseFloor | -98dBm | |
| *.**.nic.phy80211p.decider | xmldoc("config.xml") | |
| *.**.nic.phy80211p.analogueModels | xmldoc("config.xml") | |
| *.**.nic.phy80211p.usePropagationDelay | true | |
| *.**.nic.phy80211p.antenna | xmldoc("antenna.xml", "/root/Antenna[@id='monopole']") | |
| *.node[*].nic.phy80211p.antennaOffsetY | 0 m | |
| *.node[*].nic.phy80211p.antennaOffsetZ | 1.895 m | |
| *.node[*].applType | "TraCIDemo11p" | |
| *.node[*].appl.headerLength | 80 bit | |
| *.node[*].appl.sendBeacons | false | |

3 unassigned parameters

```
<terminated> veins [OMNeT++ Simulation] C:\Users\SpitFire\src\omnetpp-5.6.2\bin/opp_run.exe (2021.01.21. 15:43 - )

Loading NED files from .:  1
Loading NED files from ../../src/veins:  43

Loading images from 'C:\Users\SpitFire\src\veins-5.1-2\veins-veins-5.1\images': *: 0  veins/*: 0  veins/node/*: 45  veins/sign/*: 5
Loading images from 'C:\Users\SpitFire\src\omnetpp-5.6.2\images': *: 0  abstract/*: 90  background/*: 4  block/*: 320  device/*: 195  logo/*: 1  maps/*: 9  misc/*: 70  msg/*: 55  old/*: 111  status/*: 28


Simulation terminated with exit code: -2147483645
Working directory: C:/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/examples/veins
Command line: ../../../../omnetpp-5.6.2/bin/opp_run.exe -m -n .;../../src/veins --image-path=../../images -l ../../src/veins omnetpp.ini

Environment variables:
PATH=;C:/Users/SpitFire/src/veins-5.1-2/veins-veins-5.1/src;C:\Users\SpitFire\src\omnetpp-5.6.2\bin;C:\Users\SpitFire\src\omnetpp-5.6.2\tools\win64\mingw64\bin;C:\Users\SpitFire\src\omnetpp-5.6.2\tools\win64\usr\bin;;C:/Users/SpitFire/src/omnetpp-5.6.2/ide/jre/bin/server;C:/Users/SpitFire/
OMNETPP_ROOT=C:/Users/SpitFire/src/omnetpp-5.6.2/
OMNETPP_IMAGE_PATH=C:\Users\SpitFire\src\omnetpp-5.6.2\images
```

Figure 8: Omnetpp.ini Parameters and option to switch between editing modes.

By setting the port, it is possible to run SUMO only with the OMNeT++ simulation together as a Single Client using the TraCI interface without the complex Launchd TCP proxy.

## 4 Multi Client Setup

This section provides information about interfacing OMNeT++ with Simulink. With the help of this method, Simulink and OMNeT++ can control SUMO together through the TraCI interface.

To make it possible to run SUMO together with Simulink and OMNeT++ and other clients, SUMO must be initialized in multi-client mode. It can be done by using the - *-num-clients <number of clients as INT>* command for SUMO start. After SUMO is started, its TraCI server will wait for the given number of clients to connect before start. In case of multi-client setup, SUMO needs a setOrder command from every client, to be able to identify them later during the simulation. Every client will be able to issue TraCI commands to SUMO and read / write data. If the setOrder command is missing from a client, SUMO will stop working.
⚠ Important: TraCI setOrder command must be sent before any other TraCI commands from every client or SUMO will stop with an error!

TraCI4Matlab does not have setOrder implemented, so it must be written by the user. For this, the setOrder.m function has to be created in the +traci folder. The setOrder function:

```
function [result] = setOrder(Order)
%% setOrder command for SUMO Multi-client mode.


global connections message
import traci.constants



% Check connection
if ~isempty(connections)
    if isKey(connections,'') && ~isempty(connections(''))
        % Build the setOrder command
        % Prepare the message to be sent to the SUMO server
        message.queue = [message.queue uint8(sscanf(constants.CMD_SETORDER,
            '%x'))];
        message.string = [message.string uint8([1+1+8 sscanf(constants.
            CMD_SETORDER,'%x')]) ...
             traci.packInt64(Order)];

        % Send the message
        result = traci.sendExact();
    end
end
```

This function will send the TraCI command *0x03* which will set the order of the client. The function's argument is an integer, which represents the order number of the client. The function can be called in a Matlab script using Traci4Matlab like this:

```
function SUMO_Starter(NetworkName, StepLength, Port)

    %Connect to SUMO
    javaaddpath('traci4matlab.jar')
    scenarioPath = char(NetworkName);
    sumoHome = 'C:\Users\SpitFire\src\sumo-1.8.0\';

    % VEINS - INET
    system(['C:\Users\SpitFire\src\sumo-1.8.0\bin\sumo-gui.exe' ' -c '
        scenarioPath ' --num-clients' ' 2' ' --remote-port ' ...
    num2str(Port) ' --step-length ' char(string(StepLength)) ' --scale 1' '
         --collision.mingap-factor 0'  ' --collision.action remove' ' --
        collision.check-junctions' ...
      ' --summary results.xml' ' --start &']);

    %Initialize the simulation
    traci.init();
    %Set order #1
    traci.setOrder(1);
    disp("setOrder -DONE");

end
```

In OMNeT++ in the C++ TraCI library, setOrder is not implemented either, so it must be also implemented. First, the function has to be defined inside *TraCICommandInterface.h* like this:

```
void setOrder(uint32_t order);
```

After declaration, the function can be developed inside *TraCICommandInterface.cc*:

```
// Set SUMO TraCI order
void TraCICommandInterface::setOrder(uint32_t order)
{
    EV_INFO << "Set SUMO order..." << endl;
    const uint8_t CMD_SETORDER = 3;
    TraCIBuffer orderbuf = TraCIBuffer();
    connection.query(CMD_SETORDER, orderbuf << order);
    ASSERT(orderbuf.eof());
    EV_INFO << "setOrder(): done!" << endl;
}
```

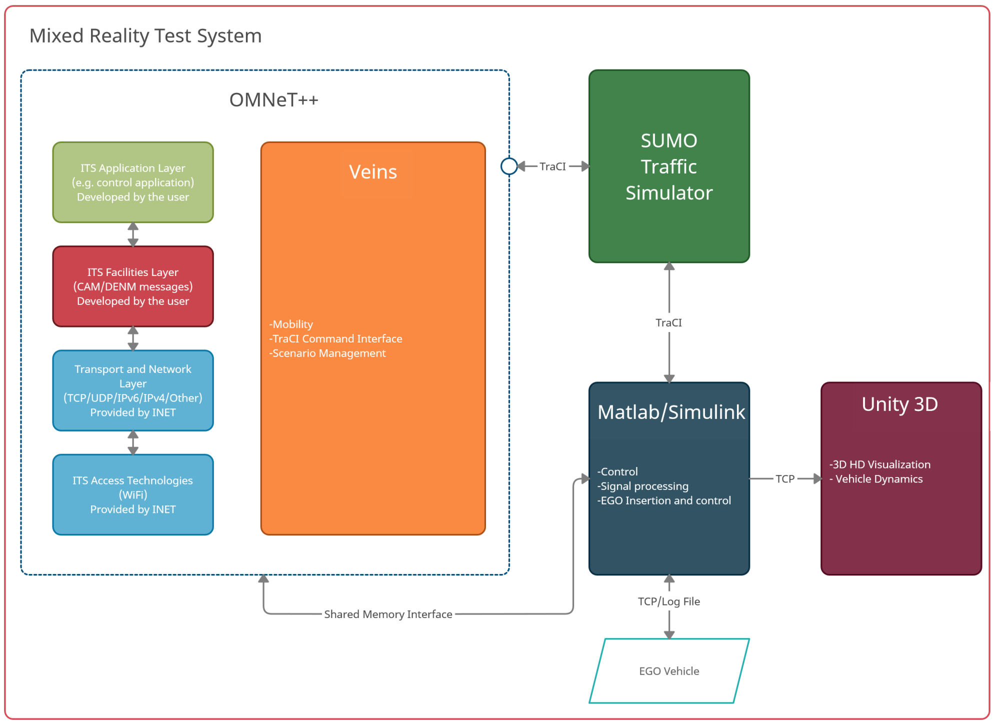
The function has to be called in *TraCIScenarioManager.cc* inside a loop. As setOrder must be called only once, a boolean variable is defined and used to call it only once. The function must be placed at the start of *void TraCIScenarioManager::init_traci()* function, before the api version function call, like this:

```
bool orderSet = false;

void TraCIScenarioManager::init_traci()
{

    auto* commandInterface = getCommandInterface();
    {
        // setOrder
        if(!orderSet)
        {
        commandInterface->setOrder(2);
        orderSet = true;

        }

        auto apiVersion = commandInterface->getVersion();
        EV_DEBUG << "TraCI server \"" << apiVersion.second << "\" reports
            API version " << apiVersion.first << endl;
        commandInterface->setApiVersion(apiVersion.first);
    }
    .
    .
    .
```

If SUMO got every order from its clients, it will start the simulation. SUMO will wait for every client to send their simulation step command to execute the step. If the first client sent the command, SUMO will freeze that client from executing another step, until the second client sends its own step command.

## 5 Simulation time settings

This section provides information about time settings, which provided a real-time simulation (without communication animations). These time settings are used in case of a mixed reality test system, where an EGO vehicle (a real vehicle or a real vehicle's log) can be inserted in the simulation with a 3D visualization (Unity 3D game engine).



Mixed Reality Test System
OMNeT++
ITS Application Layer (e.g. control application) Developed by the user
ITS Facilities Layer (CAM/DENM messages) Developed by the user
Transport and Network Layer (TCP/UDP/IPv6/IPv4/Other) Provided by INET
ITS Access Technologies (WiFi) Provided by INET
Veins
-Mobility
-TraCI Command Interface
-Scenario Management
TraCI
SUMO Traffic Simulator
TraCI
Matlab/Simulink
-Control
-Signal processing
-EGO Insertion and control
TCP
Unity 3D
-3D HD Visualization
- Vehicle Dynamics
TCP/Log File
EGO Vehicle
Shared Memory Interface

Figure 9: Mixed reality test system architecture with OMNeT++.

The provided settings may not result in a real-time simulation, depending on the running system's hardware (Used system: i9 CPU, 32GB RAM). Also, the amount of vehicle nodes is highly affecting the simulation speed.

Running OMNeT++ with Veins and INET frameworks together with a mixed reality test system using Unity 3D as a visualizer, a correct time setting is required to achieve a smooth realtime simulation (Note, that the simulation of higher number of vehicles will definitely slow down the simulation at a certain number of vehicles, meaning in some cases it is simply impossible to achieve a realtime simulation.). The following table contains every important time settings:

| **Parameter** | **Matlab** | **Simulink** | **OMNeT++ (omnetpp.ini)** | **SUMO CFG** |
|---|---|---|---|---|
| CMD_SIMSTEP time: | 0.01s | - | - | - |
| Simulink solver: | - | Fixed step: 0.01s | - | - |
| Simulation time resolution: | - | - | ps | - |
| *.manager.updateInterval: | - | - | 0.01s | - |
| Simulation mode: | - | - | Fast | - |
| step-length value: | - | - | - | 1 |

Table 1: Time setup without EGO vehicle.

OMNeT++'s *Simulation time resolution* can be found in the *omnetpp.ini* under General settings. The **.manager.updateInterval* can be found in the same file under parameters. Simulation mode can be chosen after the OMNeT++ project is built and the simulation should be started in the QT environment (Options are: Step/Run/Fast/Express/Until). These settings provide a smooth simulation run between used software without an EGO vehicle insertion.

If an EGO vehicle is inserted in SUMO with the help of the mixed reality system (tested from a log file), then the settings must be changed for a realtime simulation.

| **Parameter** | **Matlab** | **Simulink** | **OMNeT++ (omnetpp.ini)** | **SUMO CFG** |
|---|---|---|---|---|
| CMD_SIMSTEP time: | 0.1s | - | - | - |
| Simulink solver: | - | Fixed step: 0.1s | - | - |
| Simulation time resolution: | - | - | ps | - |
| *.manager.updateInterval: | - | - | 0.1s | - |
| Simulation mode: | - | - | Fast | - |
| step-length value: | - | - | - | 1 |

Table 2: Time setup with EGO vehicle.

## 6 Simulink - OMNeT++ interface (Inter-task communication)

This section provides an interface for communication between Simulink and OMNeT++. It is useful for creating control tasks in Simulink and share information between software, however it is not necessarily needed for V2X simulations.

Interfacing Simulink with OMNeT++ faced some major problems using TCP/UDP protocols (software will hang eachoter). Instead of them, a so called Named Memory Share can be used. For a flexible solution and to avoid synchronization problems, two shared memory blocks are created. Each software will have its own memory block with the read/write permissions. They will share these with the other software. The shared information will be written in memory in a form of a JSON message.

In Simulink an S-Function is created with the following C++ code:

```
#define BUF_SIZE 256
TCHAR szName[] = TEXT("Global\\Omnet");
TCHAR szName2[] = TEXT("Global\\Matlab");
HANDLE hMapFile;
HANDLE hMapFile2;
TCHAR szMsg[] = TEXT("Init text.");
```

```

// Reading shared memory from OMNeT++
int shareMemory()
{
        LPCTSTR pBuf;

                hMapFile = OpenFileMapping(
                        FILE_MAP_ALL_ACCESS,    // read/write access
                        FALSE,                  // do not inherit the name
                        szName);                // name of mapping object

                if (hMapFile == NULL)
                {
                        _tprintf(TEXT("Could not open file mapping object
                           (%d).\n"),
                                GetLastError());
                        return 1;
                }
                init = true;



        pBuf = (LPTSTR)MapViewOfFile(hMapFile, // handle to map object
                FILE_MAP_ALL_ACCESS,  // read/write permission
                0,
                0,
                BUF_SIZE);

        if (pBuf == NULL)
        {
                _tprintf(TEXT("Could not map view of file (%d).\n"),
                        GetLastError());

                CloseHandle(hMapFile);

                return 1;
        }

        std::string message = pBuf;
        cout << "Data from Omnet shared memory: " << message << "\n" <<
            endl;

        UnmapViewOfFile(pBuf);

        CloseHandle(hMapFile);

        return 0;
}

// Writing Simulink's own data in a separate memory block
int shareMemory2()
{
        loop_round = rand() % 1000;
        json msg;
        msg["Root"]["Test"] = loop_round;
        std::string smsg = msg.dump();
        for (int j = 0; j < sizeof(smsg); j++)
        {
                szMsg[j] = smsg[j];
        }

        hMapFile2 = CreateFileMapping(
                INVALID_HANDLE_VALUE,    // use paging file
                NULL,                    // default security
                PAGE_READWRITE,          // read/write access
                0,                       // maximum object size (high-order DWORD)
                BUF_SIZE,                // maximum object size (low-order DWORD)
                szName2);                 // name of mapping object
        if (hMapFile2 == NULL)
        {

                std::cout << "CreateFile Error:" << GetLastError() << std::endl;
                return 1;
        }

        LPCTSTR pBuf2;
        pBuf2 = (LPTSTR)MapViewOfFile(hMapFile2,   // handle to map object
                FILE_MAP_ALL_ACCESS, // read/write permission
                0,
                0,
                BUF_SIZE);

        if (pBuf2 == NULL)
        {
                std::cout << "Map File Error:" << GetLastError() << std::endl;

                CloseHandle(hMapFile2);

                return 1;
        }


        CopyMemory((PVOID)pBuf2, szMsg, (sizeof(szMsg) * sizeof(std::string)));
        std::cout << "Loop Round:" << loop_round << std::endl;

        UnmapViewOfFile(pBuf2);

        //CloseHandle(hMapFile);

        return 0;
}
```

In OMNeT++ the same code can be used.

⚠ Important: To be able to use the memory share function, OMNeT++ and Matlab must be started with system administrator privileges!

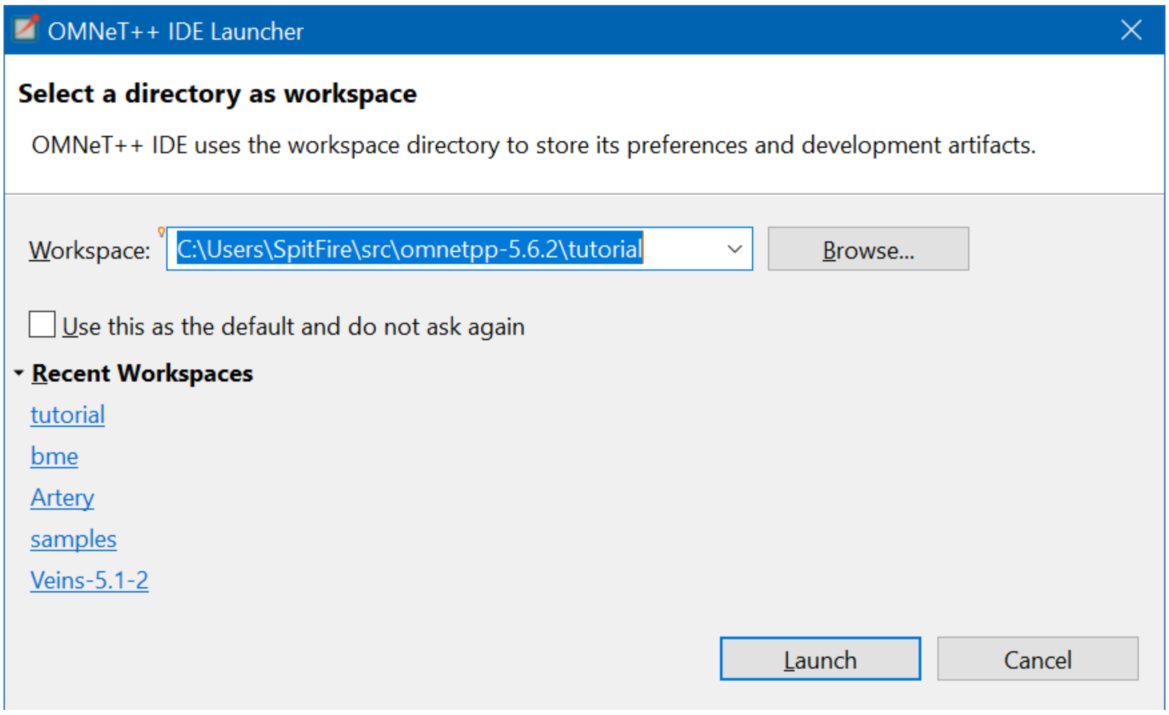
# 7 Creating an own project using Veins and INET together

This section provides the steps of creating an own project with Veins and INET frameworks working together.

The INET framework contains the protocols and the application layer to communicate through the "wlan0" interface with UDP protocol in multicast mode. It can be used to simulate V2X communication. Veins contains only the MAC (Media Access Control) and the physical layer of the communication. Combining these two frameworks will provide a more accurate simulation of the communication.

## 7.1 Creating own project in OMNeT++

First an empty project is created, where the whole project can be developed. For this, OMNeT++ should be opened and in the launcher window, a new workspace should be created (e.g., "tutorial").


OMNeT++ IDE Launcher

**Select a directory as workspace**

OMNeT++ IDE uses the workspace directory to store its preferences and development artifacts.

Workspace: C:\Users\SpitFire\src\omnetpp-5.6.2\tutorial

Figure 10: Creating new workspace.

After this, OMNeT++ will ask to install the INET Framework. It should not be installed, because the right version should be installed manually. INET should be downloaded now, and unpacked anywhere. The inet4 folder should be copied to the workspace. The existing project should be imported to OMNeT++ (like in Section 2). If the modifications of the Veins framework were followed (excluding the launchd script), the entire Veins (Veins-5.0) folder has to be copied to the workspace. This project has to be imported too in OMNeT++.

Now a custom project can be created in OMNeT++. Using File -> New -> OMNeT++ Project. A project name should be given (e.g., tutorial) and the default location should be used with C++ Development Support. Press Next and select "Empty project with 'src' and 'simulations' folders." Click Finish. The Project Explorer should look like in Fig. 11.

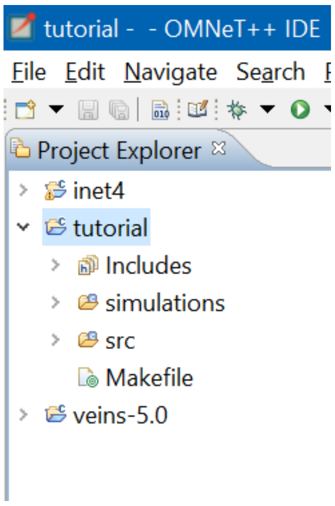


Figure 11: Created project.

## 7.2 Referencing Veins and INET for the custom project

To connect the two projects, both has to be referenced in the newly created empty project. For this, right click on the "tutorial" folder in the Project Explorer and choose Properties. Go to Project References and select both inet4 and veins-5.0 just like in Fig. 12.

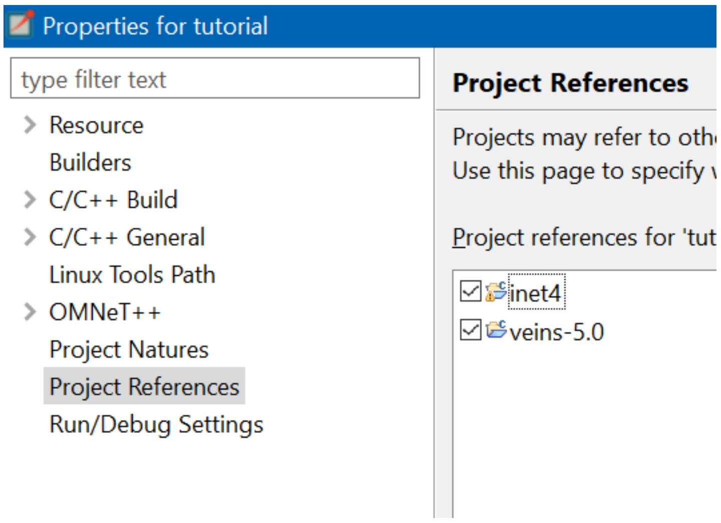
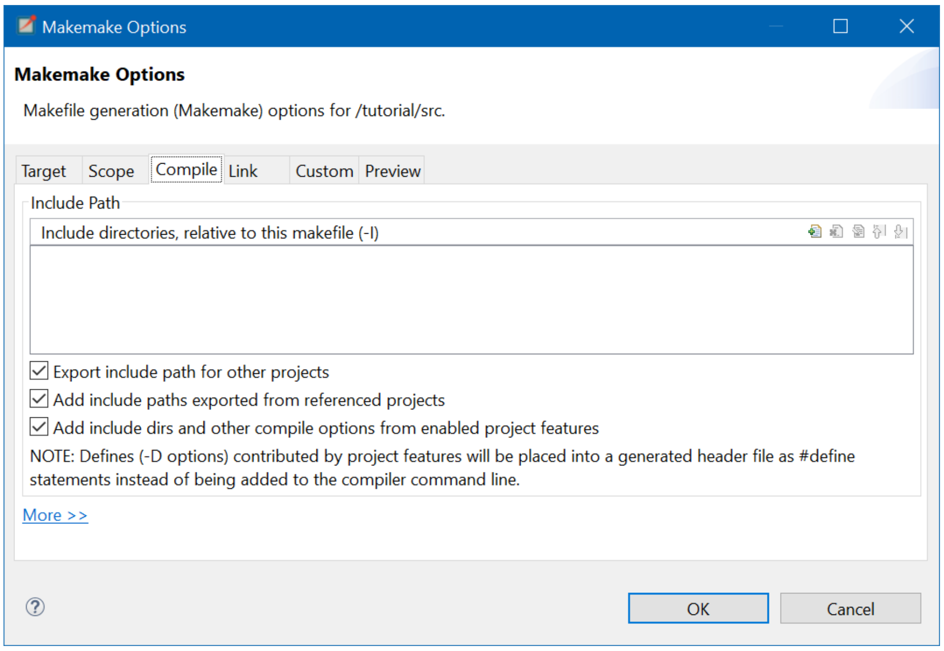


Figure 12: Referencing.

Open the OMNeT++ option, select Makemake and double click on the src:makemake(deep, recurse) –> tutorial (executable) folder. This will open up the Makemake options. Now the Compile tab should be chosen and the following options should be selected:

1. Export include path for other projects

2. Add include paths exported from referenced projects
3. Add include dirs and other compile options from enabled project features


Makemake Options

**Makemake Options**

Makefile generation (Makemake) options for /tutorial/src.

Include Path

Include directories, relative to this makefile (-I)

☑ Export include path for other projects

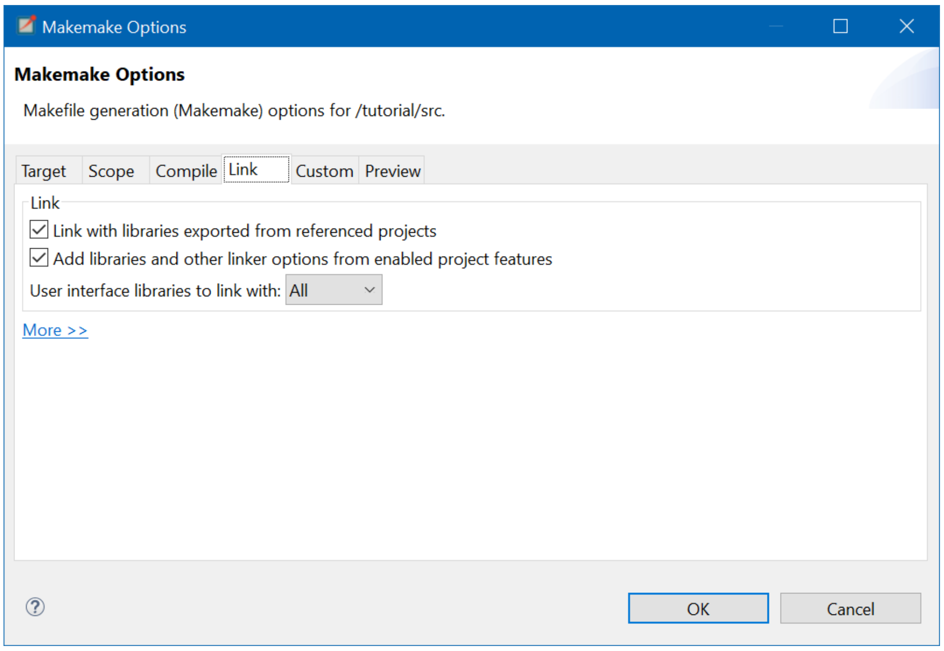
☑ Add include paths exported from referenced projects

☑ Add include dirs and other compile options from enabled project features

NOTE: Defines (-D options) contributed by project features will be placed into a generated header file as #define statements instead of being added to the compiler command line.

Figure 13: Compile

Now go to the Link tab and enable the "Add libraries and other linker options from enabled project features".


Makemake Options

**Makemake Options**

Makefile generation (Makemake) options for /tutorial/src.

Link

☑ Link with libraries exported from referenced projects

☑ Add libraries and other linker options from enabled project features

User interface libraries to link with: All

Figure 14: Link.

In the Custom tab the following code fragment should be added:

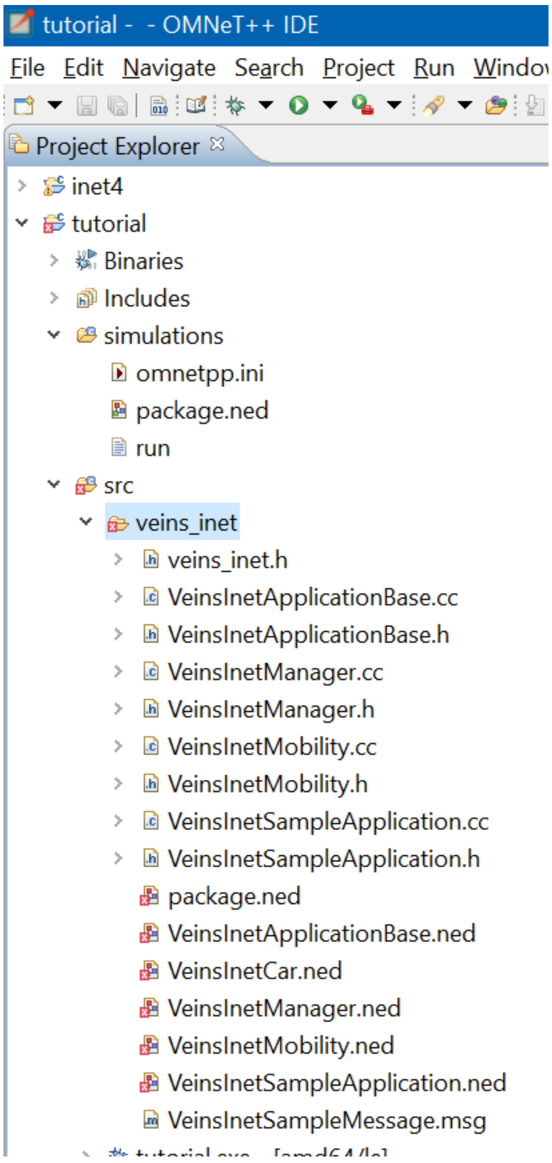
```
# Use the new message compiler introduced in OMNeT++ 5.3
#
MSGC:=$(MSGC) --msg6

ifeq ($(PLATFORM),win32.x86_64)
  #
  # on windows we have to link with the ws2_32 (winsock2) library as it is
      no longer added
  # to the omnetpp system libraries by default (as of OMNeT++ 5.1)
  #
  LIBS += -lws2_32
  DEFINES += -DINET_EXPORT
  ENABLE_AUTO_IMPORT=-Wl,--enable-auto-import
  LDFLAGS := $(filter-out $(ENABLE_AUTO_IMPORT), $(LDFLAGS))

endif
```

Press Ok to save and close the window and then press Apply and Close. Make sure that the veins folder has its reference with inet4!

If any missing include file errors arise during building (e.g. *fatal error: 'veins/veins.h' file not found*), then the project must be built with a custom makefile. The steps of setting up the project with a custom makefile, can be found in section 8.1.

## 7.3 Veins_INET subproject

Now the veins_inet subproject is needed in the custom project. For this, the "veins_inet" folder should be copied (located in: */veins-5.0/subprojects/veins_inet/src/* ) to *tutorial/src*. After the copy, some NED file errors will arise.



```
tutorial - - OMNeT++ IDE
File  Edit  Navigate  Search  Project  Run  Windo
Project Explorer
> inet4
v tutorial
  > Binaries
  > Includes
  v simulations
      omnetpp.ini
      package.ned
      run
  v src
    v veins_inet
      > veins_inet.h
      > VeinsInetApplicationBase.cc
      > VeinsInetApplicationBase.h
      > VeinsInetManager.cc
      > VeinsInetManager.h
      > VeinsInetMobility.cc
      > VeinsInetMobility.h
      > VeinsInetSampleApplication.cc
      > VeinsInetSampleApplication.h
        package.ned
        VeinsInetApplicationBase.ned
        VeinsInetCar.ned
        VeinsInetManager.ned
        VeinsInetMobility.ned
        VeinsInetSampleApplication.ned
        VeinsInetSampleMessage.msg
```

Figure 15: NED Errors.

This is because the custom project has different ned package locations now. To fix it, ned file has to be opened and the locations must be fixed. Opening the first one ("package.ned"). The line "package org.car2x.veins.subprojects.veins_inet;" defines the package route. It must be changed to "package tutorial.veins_inet". This step has to be repeated in every NED file which has this error. In the last one ("VeinsInetSampleApplication.ned"), the line "import org.car2x.veins.subprojects.veins_inet.inet.VeinsInetApplicationBase;", must be changed to "import tutorial.veins_inet.VeinsInetApplicationBase;" beside the package line.

Now, copy the "veins_inet" folder from "veins-5.0/subprojects/veins_inet/examples" to "tutorial/simulations". It will produce NED errors again. It should be fixed with the same method: "package org.car2x.veins.subprojects.veins_inet.example;" should get its new location: "tutorial.simulations.veins_inet;". Scenario.ned should look like this:

```
package tutorial.simulations.veins_inet;

import inet.physicallayer.ieee80211.packetlevel.Ieee80211ScalarRadioMedium;
import tutorial.veins_inet.VeinsInetCar;
import tutorial.veins_inet.VeinsInetManager;
```

In omnetpp.ini the "org.car2x.veins.subprojects." should be also changed to "tutorial.". The VeinsInetCar.ned (located in "tutorial/src/veins_inet") has a submodule part. It must be cleared, so the file should look like this:

```
package tutorial.veins_inet;

import inet.networklayer.configurator.ipv4.HostAutoConfigurator;
import inet.node.inet.AdhocHost;

//
// Wireless-enabled Host
//
module VeinsInetCar extends AdhocHost
{
    parameters:
        @display("i=device/cellphone");
        ipv4.configurator.networkConfiguratorModule = "";

    submodules:
}
```

After getting rid of NED errors, the Scenario Manager has to be modified. It can be done, by changing the TraCIScenarioManagerLaunchd references to TraCIScenarioManager in the *VeinsInetManager.ned* file. Also the following line has to be removed from the omnetpp.ini file:

```
*.manager.launchConfig = xmldoc("square.launchd.xml")
```

This line was the configuration for the TraCIScenarioManagerLaunchd, so it is not needed anymore. If this is done, the simulation should be ready to be built and run.

## 8 Implementing ASN1 Standard CAM messages

The Standard Cooperative Awareness Messages are defined in ETSI EN 302 637-2 V1.4.1 (Link) by an ASN1 Specification (see: Annex A (normative): ASN.1 specification of CAM). Abstract Syntax Notation One (ASN.1) is a standard interface description language for defining data structures that can be serialized and deserialized in a cross-platform way. To create a standard CAM message structure with its encoding and decoding rules, this ASN1 specification should be used. Creating the structure and the rules in C++ compatible C would be an enormous job (the generated structure will contain 663 .c/.h files), so it is easier to use some other solutions.

At the time of writing this documentation, the newest CAM standard is version 1.4.1. To generate the CAM structure in C, an ASN1C Compiler can be used Link. This documentation does not go into details of the usage of the compiler, please see its own documentation.

Instead of using the compiler, the generated structures can be downloaded with Vanetza (Vanetza is an open-source implementation of the ETSI C-ITS protocol suite.) [Link], which is the preferred way because the ASN1C compiler can be tricky to be used. Furthermore Vanetza contains not only the ETSI CAM standard generated, but standards like EN302637-3v131-DENM, ISO TS 19091 CPM, Etsi Ts 102941 Messages etc.

## 8.1 Building a Standalone Library for CAM messages

As Vanetza comes with the precompiled ASN1 standards, it seems easy to use the generated structures. Unfortunately (at least on Windows), Omnet++ compiles only .cc files. As the structure is generated in C++ compatible C code, it will not be compiled by the Omnet++ IDE, which uses *opp_makemake*. If header files are included, it will produce undefined reference errors, because it will only see the function prototypes and not the function definitions.

To build a library, which can be included in the custom Omnet++ build, a Makefile must be created and Omnet++'s MingW console must be used. First a build folder must be created. Create a new folder inside it and call it "sources". Copy every file to this folder from the folders *vanetza-master > vanetza > asn1 > its* and from *vanetza-master > vanetza > asn1 > support*. Delete the following files from the source folder:

1. GeneralizedTime.c
2. GeneralizedTime.h
3. UTCTime.c
4. UTCTime.h

Two files must be also modified by including a library to avoid some compilation errors. Add the following line in files *INTEGER.c* and *RELATIVE-OID.c*:

```
#include <inttypes.h>
```

Now in the root of the build folder, create a file named *Makefile* without any extensions and put the following makefile code in it:

```
# Makefile for ASN1C CAM Standalone Static Library
# Set up the compiler to use: (OMNeT++ is using clang++. For compatibility,
    we use clang to compile.)
CC = clang

# Setup of compiler flags:
#  -g    adds debugging information to the executable file
#  -Wall turns on most, but not all, compiler warnings
CFLAGS = -g -Wall

# Variable for source directory
SOURCEDIR = sources

# Get sources from the SOURCEDIR with .c extension
SOURCES := $(shell find $(SOURCEDIR) -name '*.c')

# All header files (it is not necessery, because clang will search the
    header files in the same folder, where .c files are)
HEADERS := $(shell find $(SOURCEDIR) -name '*.h')

# Build directory
BUILD = lib

# Creating a list of the objects, which will be created from the source .c
    files
OBJ = $(SOURCES:.c=.o)

# ASN1C Library to be created -> We have to create a Standalone Library
```

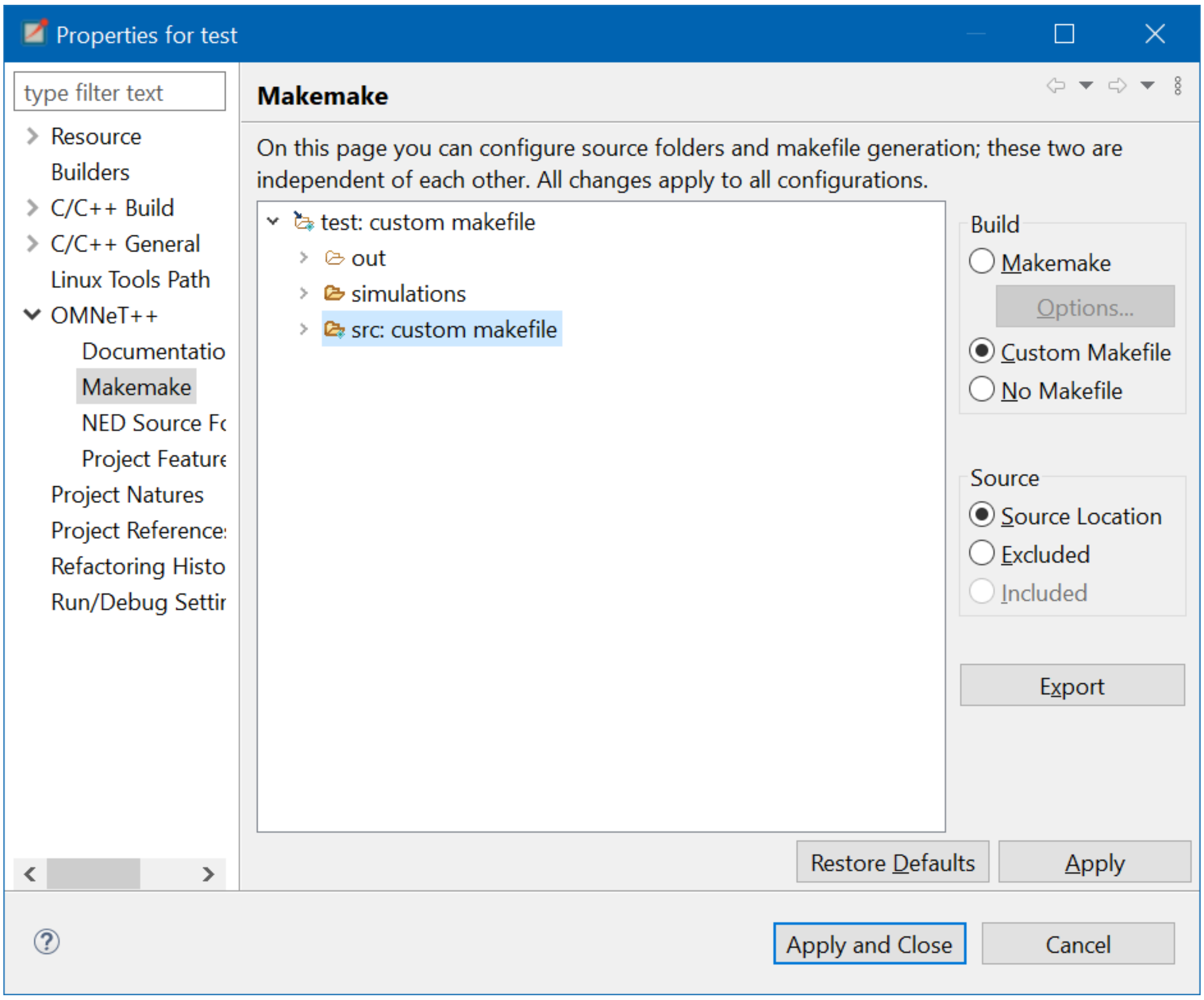
```
STATIC = libasn1c.a

# Linking object files into our library
# Note: The Make program works in a dependency order.
# It means, that even we write the linking commands before compilation, it
    depends on the OBJ variable.
# If objects are not existing, it will run the compilation first. -> $(
    STATIC) depends on $(OBJ)
$(STATIC): $(OBJ)
        @echo "[Link (Static)]"
        @ar rcs $@ $^


# Here comes the compilation.
# The command: clang -c -g -Wall <dependency file> -o <target reference> -
    std=gnu11
# -std is the language mode, where gnu11 is C11 language standard with GNU
    extensions
# $< is a macro that refers to the first dependency
# $@ is a macro that refers to the target
# Use -DASN_EMIT_DEBUG=1 if we need debug information from encoding/
    decoding functions
.c.o:
        @echo [Compiling] $<
        @$(CC) -c $(CFLAGS) $< -o $@ -std=gnu11 -DASN_EMIT_DEBUG=1


# A phony target is one that is not really the name of a file; rather it is
     just a name for a recipe to be executed when you make an explicit
    request.
# There are two reasons to use a phony target: to avoid a conflict with a
    file of the same name, and to improve performance.
.PHONY: .c.o clean

# Command 'make clean' will erase the generated objects and the linked
    library
clean:
        rm -f $(OBJ) *~ core tags *.bak Makefile.bak libgeniePi.*
        rm -f $(STATIC)
```

Now open the MingW console (from the omnetpp install folder, mingwenv.cmd) and navigate to the build folder's root. Type and run the command: *make* and it should compile the source files and create the libasn1c.a library file to the root of the build folder (objects will be generated next to the source files in sources folder). If everything goes right, it will only produce some warnings but no errors. If it is needed to clear the created object files, the command *make clean* can be used in the console.

*Note: clang version 5.0.1 was used.* To check your clang version, type *clang -v* in MingW.

To use the library, copy the *libasn1c.a* into *YOUR-OMNETPP-INSTALL-FOLDER/tools/win64/libs/*.

Now open Omnet++, right click on the project folder (called tutorial) in the Project Explorer and open Properties. Under OMNeT++ select Makemake. Select the src folder and choose the option Custom Makefile. Click Apply and close.


Properties for test

Makemake

On this page you can configure source folders and makefile generation; these two are independent of each other. All changes apply to all configurations.

test: custom makefile
- out
- simulations
- src: custom makefile

Build: ( ) Makemake [✓ Custom Makefile] ( ) No Makefile

Source: [✓ Source Location] ( ) Excluded ( ) Included

Figure 16: Using custom Makefile.

If there were any missing include file errors before, then the custom makefile has to have the following lines:

```
# C++ include paths (with -I)
# INET should be added by previously used steps, we have to add veins
  manually if we had build errors
INCLUDE_PATH = -I$(INET4_PROJ)/src -I. -I$(VEINS_PROJ)/src -I.

# Other makefile variables (-K)
# The makefile should also contain these variables automatically, but if it
  's not there, add them.
INET4_PROJ=../../inet4
VEINS_PROJ=C:/Users/Username/src/veins-5.0 # Path to your veins folder
```

The static library must be built with the custom OMNeT++ project. For this, the custom makefile is used, which was automatically generated while the project was built for the first time. Now it must be modified.

Open the make file in any text editor which is located in: *WORKSPACE-FOLDER/tutorial/src/*. (it's a file called Makefile without any extensions). Find the *LIBS* variable and put the full path of the static library in it, like this:

```
# Additional libraries (-L, -l options)
LIBS = $(LDFLAG_LIBPATH)$(INET4_PROJ)/src $(LDFLAG_LIBPATH)$(VEINS_PROJ)/
    src C:/Users/USER/src/omnetpp-5.6.2/tools/win64/libs/libasn1c.a -lINET$(
    D) -lveins$(D)
```

CAM.h can be now included from the build folder and the CAM structure can be used.

## 8.2 Usage of the CAM structure

This section provides information about the usage of the built CAM structure.

After the static library is built and the main header file of the CAM structure (CAM.h) is included, it can be used in the custom project. In a Veins_Inet project, the communications' application layer can be programmed in VeinsInetSampleApplication.cc located under *src/veins_inet/*.

As CAM messages will be encoded in UPER encoding, a buffer will be needed which can hold this data. Some functions must be defined too in VeinsInetSampleApplication.h, which will be used later. The file should look like this:

```
#pragma once

#include "veins_inet/veins_inet.h"
#include "veins_inet/VeinsInetApplicationBase.h"

class VEINS_INET_API VeinsInetSampleApplication : public veins::
    VeinsInetApplicationBase {

protected:
    bool haveForwarded = false;

protected:
    virtual bool startApplication() override;
    virtual bool stopApplication() override;
    virtual void processPacket(std::shared_ptr<inet::Packet> pk) override;


public:
    VeinsInetSampleApplication();
    ~VeinsInetSampleApplication();
    void generateCAM(); // Function for CAM generation and sending
    std::pair<double,double> getVehicleLatLong(); // Function for getting
        SUMO vehicle GeoPosition
    uint8_t msg_buffer[1024]; //Buffer for CAM
};
```

OMNeT++ has its own solution for creating messages/packets. It already has a sample .msg in the veins_inet folder, called VeinsInetSampleMessage.msg. This file should be opened and a buffer should be defined, which will be transmitted between nodes with the encoded CAM data. The content of the file should look like this:

```
import inet.common.INETDefs;
import inet.common.packet.chunk.Chunk;

//
// Example message definition
//
class VeinsInetSampleMessage extends inet::FieldsChunk
```

```
{
    string roadId; // This is from the example, but we can leave it here
    uint8_t messageBuffer[1024]; // Buffer for the CAM data
}
```

OMNeT++ will automatically generate a file called VeinsInetSampleMessage_m.cc file and the corresponding header too. These generated files should not be changed, because OMNeT++ will rebuild them and overwrite them anyway. If the buffer is defined, the solution can be built with Ctrl + B and the message header can be included in VeinsInetSampleApplication.cc. Also include the math library.

```
#include "veins_inet/VeinsInetSampleMessage_m.h"
#include <math.h>
```

Now a function *generateCAM()* can be created in VeinsInetSampleApplication.cc. To initialize the CAM structure, the *CAM_t ** type has to be defined and the memory must be allocated for it by initializing a new instance of it. The encoder return value variable must be also defined. The message buffer what we defined in the header should be cleared.

```
// Function to fill CAM data for every node and encode it in UPER then send
   it in every second.
void VeinsInetSampleApplication::generateCAM()
{
    // Memory allocation for CAM_t
    CAM_t *cam = new CAM_t();

    // encoder return value
    asn_enc_rval_t erv;

    // Clear msg_buffer just to make sure it's empty
    memset(msg_buffer, 0, sizeof(msg_buffer));

}
```

Now the CAM structure can be accessed through the *CAM_t *cam* pointer. For example to set the station ID in the CAM message, the following can be done:

```
/* CAM STRUCTURE */
// HEADER
cam->header.stationID = 1;
```

Be aware, that the structure hierarchy contains some pointers inside. If a value has to be assigned to these fields, which are defined as pointers, the memory must be allocated for them first. For example:

```
// LATERAL ACCELERATION
// First the memory allocation
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.lateralAcceleration = (
    LateralAcceleration *) calloc(1, sizeof(LateralAcceleration));
// Assigning a value to the field
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.lateralAcceleration->
    lateralAccelerationValue = -2;
```

In this hierarchy *cam->cam.camParameters.highFrequencyContainer.choice .basicVehicleContainerHighFrequency.lateralAcceleration*, where .laterAcceleration is the pointer, the memory allocation must be done with a type cast (for example: *(LateralAcceleration *)*), because calloc's type is always *void**. To find out the type which should be casted for calloc, right click on the name of the pointer and *Open Declaration*, where the structure and the corresponding type can be seen, what is needed. In case of .laterAcceleration:

```
/* BasicVehicleContainerHighFrequency */
typedef struct BasicVehicleContainerHighFrequency {
        Heading_t         heading;
        Speed_t   speed;
        DriveDirection_t          driveDirection;
        VehicleLength_t  vehicleLength;
        VehicleWidth_t   vehicleWidth;
        LongitudinalAcceleration_t       longitudinalAcceleration;
        Curvature_t      curvature;
        CurvatureCalculationMode_t       curvatureCalculationMode;
        YawRate_t        yawRate;
        AccelerationControl_t    *accelerationControl;    /* OPTIONAL */
        LanePosition_t  *lanePosition;  /* OPTIONAL */
        struct SteeringWheelAngle       *steeringWheelAngle;     /* OPTIONAL
            */
        struct LateralAcceleration      *lateralAcceleration; // <--- This
           is the type needed in this case!
        struct VerticalAcceleration     *verticalAcceleration;  /* OPTIONAL
            */
        PerformanceClass_t      *performanceClass;      /* OPTIONAL */
        struct CenDsrcTollingZone       *cenDsrcTollingZone;     /* OPTIONAL
            */

        /* Context for parsing across buffer boundaries */
        asn_struct_ctx_t _asn_ctx;
} BasicVehicleContainerHighFrequency_t;
```

There are some pointers which contain another pointer inside of them (for example a buffer). Then the memory has to be also allocated for the buffer. For example, the AccelerationControl field is a BIT_STRING type, which means it is a serialized binary data container. In this case the memory has to be allocated for the buffer and then the value can be assigned like this:

```
// ACCELERATION CONTROL
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.accelerationControl = (
    AccelerationControl_t *) calloc(1, sizeof(AccelerationControl_t));
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.accelerationControl->size = 1;
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.accelerationControl->buf = (uint8_t
    *) calloc(1,sizeof(uint8_t));
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.accelerationControl->buf[0] = 0
    b0100000;
```

Some fields have predefined enumerators, which should be used by the standard. Find these by opening the fields declaration and then open the declaration of the field's type. The enumerators can be seen corresponding to that type. For example in the case of the AccelerationConfidence field, the enumerator definitions are this:

```
/* Dependencies */
typedef enum AccelerationConfidence {
        AccelerationConfidence_pointOneMeterPerSecSquared        = 1,
        AccelerationConfidence_outOfRange        = 101,
        AccelerationConfidence_unavailable       = 102
} e_AccelerationConfidence;
```

And they can be assigned by just calling them:

```
cam->cam.camParameters.highFrequencyContainer.choice.
    basicVehicleContainerHighFrequency.lateralAcceleration->
    lateralAccelerationConfidence =
    AccelerationConfidence_pointOneMeterPerSecSquared;
```

After filling the data structure, the CAM message can be serialized. For this the support functions from ASN1C should be used which are also built in the static library. For an UPER encode the following function can be used:

```
erv = asn_encode_to_buffer(0, ATS_UNALIGNED_BASIC_PER, &asn_DEF_CAM, cam,
    msg_buffer, sizeof(msg_buffer));
if(erv.encoded == -1)
{
    std::cout << "ERROR: CAM Serialization failed!\n";
}else
{
    std::cout << "\n";
    for(int i = 0; i < sizeof(msg_buffer); i++)
    {
            // Print out buffer in uppercased hexadecimal
            std::cout << std::hex << std::uppercase << +msg_buffer[i] << "
                ";
    }
    std::cout << std::nouppercase << std::dec << "\n";
    std::cout << "Encoded bytes: " << erv.encoded << "\n";
}
```

Now *msg_buffer* should contain the UPER encoded data. It's time to load this data in an OMNeT++ message, exactly in the *uint8_t messageBuffer[1024];* buffer which is already created with the .msg file. For this, first a payload variable has to be created which will hold the data and contain a shared intrusive pointer. This means, that it will have a pointer at the start and at the end of the variable. With these pointers, OMNeT++ can easily create packets from the payload by adding and removing certain headers (e.g. UDP headers, MAC addresses etc.). A timestamp has to be added, the Chunk length has to be set, the buffer hasto be filled and a packet from the payload must be created before it can be sent. After the packet is sent the CAM structure and the buffers have to be cleared. Example code:

```
// Creating the payload, which is a shared pointer
auto payload = makeShared<VeinsInetSampleMessage>();

// Timestamp adds the newest generation time to the payload (or if it
    already has one, it removes the old one)
timestampPayload(payload);

// Defining payload Chunk length
payload->setChunkLength(B(1024));

// Fill the payload message buffer with the UPER serialized CAM message
for(int k = 0; k < sizeof(msg_buffer); k++)
{
    payload->setMessageBuffer(k, msg_buffer[k]);
}

// Creating the packet
auto packet = createPacket("camPacket");

// Insert the payload at the end pointer of the packet
packet->insertAtBack(payload);
sendPacket(std::move(packet));

// Memory release and clearing buffer
ASN_STRUCT_FREE(asn_DEF_CAM, cam);
memset(msg_buffer, 0, sizeof(msg_buffer));
cam = NULL;
delete cam;
```

To send the message a timer is needed:

```
// Creating an interval for calling "callback"
timerManager.create(veins::TimerSpecification(callback).interval(SimTime(1,
     SIMTIME_S)), "CAM");
```

This timer will call the callback named *callback* with an interval of 1 second.

The CAM generation code should be written inside of the callback like this:

```
auto callback = [this]()
{
 /*CAM generation, serialization and packet creation/sending code comes
     here*/
}
```

During the simulation, every node will create an instance of VeinsInetSampleApplication. This means, that the created code will run on every SUMO vehicle. The *generateCAM()* function must be called in *VeinsInetSampleApplication::startApplication()*. A complete example code with the packet processing and some node corresponding settings:

```
bool VeinsInetSampleApplication::startApplication()
{
    /* Generating CAM message for every node in the simulation. Creating a
        payload and a packet and send it. */
    generateCAM();

    return true;
}

void VeinsInetSampleApplication::processPacket(std::shared_ptr<inet::Packet
    > pk)
{
    auto payload = pk->peekAtFront<VeinsInetSampleMessage>();
    getParentModule()->getDisplayString().setTagArg("i", 1, "green");
    EV_INFO << "Received packet: " << payload << endl;

    // DEBUG: Received encoded message
    for(int i = 0; i < 1024; i++)
    {
        std::cout << std::hex << std::uppercase << +payload->
            getMessageBuffer(i);
```

```cpp
    }
    std::cout << std::nouppercase << std::dec << "\n";

    if (haveForwarded) return;

    auto packet = createPacket("relay");
    packet->insertAtBack(payload);
    sendPacket(std::move(packet));

    haveForwarded = true;
}

// Function to fill CAM data for every node and encode it in UPER then send it in every second.
void VeinsInetSampleApplication::generateCAM()
{
    // Send CAM message from node[0], other nodes will forward the message
    if(getParentModule()->getIndex() == 0)
    {
        auto callback = [this]()
            {
                // Set node icon to red, indicating CAM generation
                getParentModule()->getDisplayString().setTagArg("i", 1, "red");
                /* CAM Message Structure */

                // Memory allocation for CAM_t
                CAM_t *cam = new CAM_t();

                // encoder return value
                asn_enc_rval_t erv;

                // Clear msg_buffer just to make sure it's empty
                memset(msg_buffer, 0, sizeof(msg_buffer));

                /* Get vehicle parameters */
                // node ID
                int nodeID = getParentModule()->getIndex();
                std::cout << "Start CAM generation...\n";

                /* CAM STRUCTURE */
                // HEADER
                cam->header.stationID = nodeID;
                cam->header.protocolVersion = 1;
                cam->header.messageID = ItsPduHeader__messageID_cam;

                // GenDeltaTime
                cam->cam.generationDeltaTime = 11409;

                // BASIC CONTAINER
                cam->cam.camParameters.basicContainer.stationType = StationType_passengerCar;
                // Print out serialized CAM MSG
                asn_fprint(stdout, &asn_DEF_CAM, cam);

                /* UPER ENCODE TO buffer uint8_t msg_buffer[SIZE] */
                std::cout << "Serialize...\n";

                erv = asn_encode_to_buffer(0, ATS_UNALIGNED_BASIC_PER, &asn_DEF_CAM, cam, msg_buffer, sizeof(msg_buffer));
                if(erv.encoded == -1)
                {
                    std::cout << "ERROR: CAM Serialization failed!\n";
                }else
                {
                    std::cout << "\n";
                    for(int i = 0; i < sizeof(msg_buffer); i++)
                    {
                            // Print out buffer in uppercased hexadecimal
                            std::cout << std::hex << std::uppercase << +msg_buffer[i] << " ";
                    }
                    std::cout << std::nouppercase << std::dec << "\n";
                    std::cout << "Encoded bytes: " << erv.encoded << "\n";
                }

                // Creating the payload, which is a shared pointer
                auto payload = makeShared<VeinsInetSampleMessage>();

                // Timestamp adds the newest generation time to the payload (or if it already has one, it removes the old one)
                timestampPayload(payload);

                // Defining payload Chunk length
                payload->setChunkLength(B(1024));

                // Fill the payload message buffer with the UPER serialized CAM message
                for(int k = 0; k < sizeof(msg_buffer); k++)
                {
                    payload->setMessageBuffer(k, msg_buffer[k]);
                }

                // Creating the packet
                auto packet = createPacket("camPacket");

                // Insert the payload at the end pointer of the packet
                packet->insertAtBack(payload);
                sendPacket(std::move(packet));

                // Memory release and clearing buffer
                ASN_STRUCT_FREE(asn_DEF_CAM, cam);
                memset(msg_buffer, 0, sizeof(msg_buffer));
                cam = NULL;
                delete cam;
            };
        // Creating an interval for calling "callback"
        timerManager.create(veins::TimerSpecification(callback).interval(SimTime(1, SIMTIME_S)), "CAM");
    }
}
```

## 8.3 Extending TraCICommandInterface

The Veins framework has a TraCICommandInterface which contains some functions, which can get some parameters from SUMO through the TraCI interface. To fill CAM message data, these functions must be extended. Open the header file located in: *veins/src/veins/modules/mobility/traci/TraCICommandInterface.h*. Declare new functions as public:

```
double getVehicleSpeed();
double getVehicleAngle();
```

Now create them in the .cc file:

```
double TraCICommandInterface::Vehicle::getVehicleSpeed()
{
    return traci->genericGetDouble(CMD_GET_VEHICLE_VARIABLE, nodeId,
       VAR_SPEED, RESPONSE_GET_VEHICLE_VARIABLE);
}

double TraCICommandInterface::Vehicle::getVehicleAngle()
{
    return traci->genericGetDouble(CMD_GET_VEHICLE_VARIABLE, nodeId,
       VAR_ANGLE, RESPONSE_GET_VEHICLE_VARIABLE);
}
```

These functions will get the vehicle speed and vehicle heading parameters through the TraCI interface.

To call them in the VeinsInetSampleApplication.cc use the following pointer:

```
double vehicleHeading = traciVehicle->getVehicleAngle(); // This will
    return the heading of the SUMO vehicle.
```

Any other extensions can be defined, just the SUMO Documentation has to be followed for variables (like VAR_ANGLE, it can be also right clicked and declaration can be opened where every declared variable can be found.)

## 8.4 Get SUMO Vehicles' Geo location

To get the Latitude and Longitude of a SUMO vehicle another function has to be created in VeinsInetSampleApplication. They were already defined in the header file (see 8.2). The function should look like this in the VeinsInetSampleApplication.cc file:

```
/*
 * This function gets the current position of the node from OMNeT++ in an
     inet::Coord type.
 * It transforms the inet::Coord type to the veins::Coord type.
 * Then it returns the Latitude and Longitude in an std::pair<double,double
     > type.
 * Longitude: pair.first
 * Latitude: pair.second
*/
std::pair<double,double> VeinsInetSampleApplication::getVehicleLatLong()
{
    // Get the vehicle coordinates from OMNeT++ stored in an INET Coord
    inet::Coord vehiclePosition = mobility->getCurrentPosition();
    // Veins Coord
    veins::Coord veinsCoord;
    // Veins Coord Longitude
    veinsCoord.x = vehiclePosition.x;
    // Veins Coord Latitude
    veinsCoord.y = vehiclePosition.y;
    std::pair<double, double> CoordPair = traci->getLonLat(veinsCoord);
    // Multiply values to get the correct form for CAM
    CoordPair.first *= 1000.0;
    CoordPair.second *= 1000.0;
    // Return the Coordinate Pair
    return CoordPair;
}
```

## 8.5 generationDeltaTime

For generationDeltaTime an epoch has to be created as the standard defined it. It is using an external header: Link

Create the following function in VeinsInetSampleApplication.cc with includes:

```
// Time function dependencies
#include <chrono>
#include "date.h"
#include <cstdint>

// Clock structure for generationDeltaTime
struct myclock
{
    using rep        = std::int32_t;
    using period     = std::milli;
    using duration   = std::chrono::duration<rep, period>;
    using time_point = std::chrono::time_point<myclock>;
    static constexpr bool is_steady = false;

    static time_point now() noexcept
    {
        using namespace std::chrono;
        using namespace date;
        return time_point {duration_cast<duration>(system_clock::now() -
            sys_days{jan/1/2004})};
    }
};

// Function to generate the generationDeltaTime for the CAM message
// DEFINE IT FIRST IN THE HEADER!
uint64_t  VeinsInetSampleApplication::getGenerationDeltaTime()
{
   auto tp = myclock::now();
   uint64_t generationDeltaTime;
   generationDeltaTime = tp.time_since_epoch().count();

   return generationDeltaTime;
}

// Usage by the standard:
uint64_t  generationDeltaTime = getGenerationDeltaTime() % 65536;
```

### 8.6 Adding pathPoints to pathHistory

The PathPoint can be created and added to the PathHistory field with the following example:

```
// PATH HISTORY
PathPoint* pathPoint = new PathPoint();
pathPoint->pathPosition.deltaLatitude = calculateDelta(Latitude, latBefore)
    ;
pathPoint->pathPosition.deltaLongitude = calculateDelta(Longitude,
    longBefore);
pathPoint->pathPosition.deltaAltitude = DeltaAltitude_unavailable;

// Add the pathPoint to the pathHistory
ASN_SEQUENCE_ADD(&cam->cam.camParameters.lowFrequencyContainer->choice.
    basicVehicleContainerLowFrequency.pathHistory, pathPoint);
```

## 9 Creating a custom RSU Application

To develop a custom RSU application, an OMNeT++ module has to be created first to simulate the RSU itself, which will run the application just like in the case of vehicle nodes. First, a new NED file has to be created in the custom project under *src/veins_inet*. Name the new NED file as RSUStation.ned and use the following source code for the module:

```
package tutorial.veins_inet;

import inet.networklayer.configurator.ipv4.HostAutoConfigurator;
import inet.node.inet.AdhocHost;

//
// Wireless-enabled Host
//
module RSUstation extends AdhocHost
{
    parameters:
        @display("p=150,150;");
        ipv4.configurator.networkConfiguratorModule = "";
    submodules:

}
```

With this module, the module of vehicles is recreated, so the RSU will be handled just like any vehicle node in the simulation. With the help of this, the same radio settings can be used both for vehicles and RSUs. Create the module of the application layer as RSUApplication.ned with the following source:

```
package tutorial.veins_inet;

import tutorial.veins_inet.VeinsInetApplicationBase;

simple RSUApplication extends VeinsInetApplicationBase
{
    parameters:
        @class(RSUApplication);

    gates:

}
}
```

To create the application itself, the RSUApplication.h must be created:

```
#pragma once

#include "veins_inet/veins_inet.h"
#include "veins_inet/VeinsInetApplicationBase.h"

class VEINS_INET_API RSUApplication : public veins::
   VeinsInetApplicationBase {

protected:

protected:
    virtual bool startApplication() override;
    virtual bool stopApplication() override;
    virtual void processPacket(std::shared_ptr<inet::Packet> pk) override;


public:
    RSUApplication();
    ~RSUApplication();
};
```

and the RSUApplication.cc:

```
#include "veins_inet/RSUApplication.h"
#include "inet/common/ModuleAccess.h"
#include "inet/common/packet/Packet.h"
#include "inet/common/TagBase_m.h"
#include "inet/common/TimeTag_m.h"
#include "inet/networklayer/common/L3AddressResolver.h"
#include "inet/networklayer/common/L3AddressTag_m.h"
#include "inet/transportlayer/contract/udp/UdpControlInfo_m.h"
#include <string.h>
#include <thread>
// Message
#include "veins_inet/VeinsInetSampleMessage_m.h"
// CAM Structure
#include "build/sources/CAM.h"
// Math
#include <math.h>
// Time function dependencies
#include <chrono>
#include "date.h"
#include <cstdint>
```

```

using namespace inet;


Define_Module(RSUApplication);

RSUApplication::RSUApplication()
{
}

bool RSUApplication::startApplication()
{
    EV_INFO << "RSU App Started!\n";
    // Insert your application code here, e.g. message generation and
       sending
    return true;
}

bool RSUApplication::stopApplication()
{
    return true;
}

RSUApplication::~RSUApplication()
{
}

// Packet processing
void RSUApplication::processPacket(std::shared_ptr<inet::Packet> pk)
{
    EV_INFO << "RSU received a message!\n";
    // Insert your application code here for message receive
    return;
}
```

To avoid crashes,VeinsInetApplicationBase.cc must be modified. Change the *handleStartOperation()* function according to the following:

```
void VeinsInetApplicationBase::handleStartOperation(LifecycleOperation*
    operation)
{
    mobility = veins::VeinsInetMobilityAccess().get(getParentModule());
    // Make sure to don't use TraCI id for RSU modules...
    EV_INFO << "MODULE NAME:" << getParentModule()->getName()<< "\n";
    if(std::strcmp( (getParentModule()->getName()), "node") == 0)
    {
        traci = mobility->getCommandInterface();
        traciVehicle = mobility->getVehicleCommandInterface();
    }

    L3AddressResolver().tryResolve("224.0.0.1", destAddress);
    ...
```

As the RSU is not present in the SUMO environment, it has to be excluded from getting a vehicle command interface, otherwise it would crash the simulation.

Now the RSU can be added to the Scenario.ned:

```
package tutorial.simulations.veins_inet;
```

```

import inet.physicallayer.common.packetlevel.NoiseSource;
import tutorial.veins_inet.VeinsInetManager;
import inet.physicallayer.ieee80211.packetlevel.
    Ieee80211DimensionalRadioMedium;
import tutorial.veins_inet.VeinsInetCar;
import tutorial.veins_inet.RSUstation;

network Scenario
{
    @display("bgb=1162,658;bgi=background/terrain,s");
    submodules:
        radioMedium: Ieee80211DimensionalRadioMedium;
        manager: VeinsInetManager;

        node[0]: VeinsInetCar {
            @display("p=531,287");
        }
        // Adding an RSU to the simulation
        rsu[1]: RSUstation {
            @display("p=150,150;i=device/antennatower");
        }
}
```

Set up the newly created RSU in omnetpp.ini:

```
# RSU Application Setup
*.rsu[*].mobility.typename = "StationaryMobility"
*.rsu[*].numApps = 1
*.rsu[*].app[0].typename = "tutorial.veins_inet.RSUApplication"
*.rsu[*].app[0].interface = "wlan0"
*.rsu[*].mobility.initFromDisplayString = false
// RSU Positioning
*.rsu[*].mobility.initialX = 150m
*.rsu[*].mobility.initialY = 150m

# RSU Radio settings
*.rsu[*].wlan[*].opMode = "p"
*.rsu[*].wlan[*].radio.bandName = "5.9 GHz"
*.rsu[*].wlan[*].radio.channelNumber = 3
*.rsu[*].wlan[*].radio.transmitter.power = 20mW
*.rsu[*].wlan[*].radio.typename = "Ieee80211DimensionalRadio"
*.rus[*].wlan[*].radio.transmitter.typename = "
    Ieee80211DimensionalTransmitter"
*.rsu[*].wlan[*].radio.receiver.typename = "Ieee80211DimensionalReceiver"
*.rsu[*].wlan[*].radio.bandwidth = 10 MHz
```

Now any number of RSUs can be added to the simulation, with any settings.

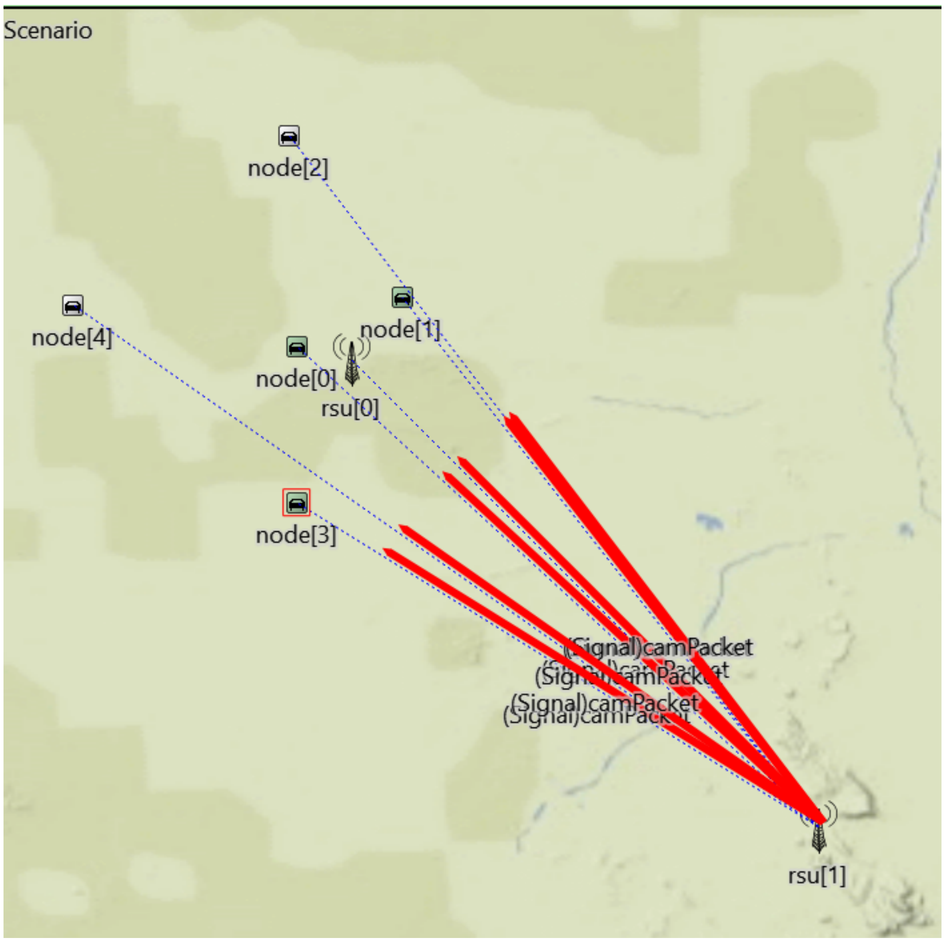
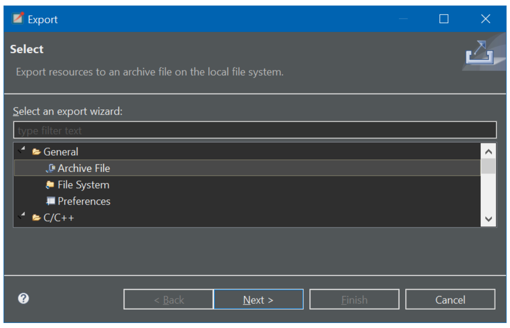


Figure 17: Multiple RSU simulation.

## 10 Moving a project

In case the project is needed on another computer, it can be easily moved. Just install the same OMNeT++ version. After that, the project has to be exported by using File -> Export and choosing Archive File under General. Settings can be seen in Fig. 18



Export

**Select**

Export resources to an archive file on the local file system.

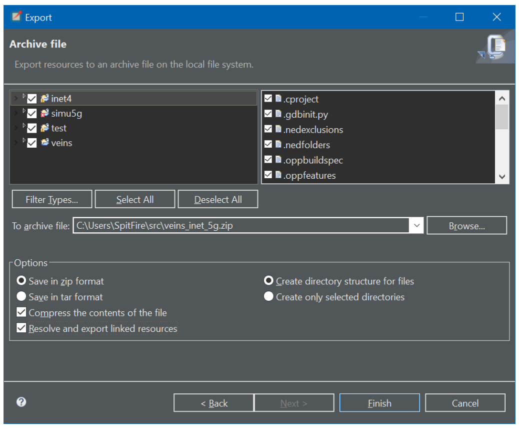
Select an export wizard:

type filter text

- General
  - Archive File
  - File System
  - Preferences
- C/C++

< Back | Next > | Finish | Cancel

(a) Export project as an archive file.



Export

**Archive file**

Export resources to an archive file on the local file system.

- [✓] inet4
- [✓] simu5g
- [✓] test
- [✓] veins

- [✓] .cproject
- [✓] .gdbinit.py
- [✓] .nedexclusions
- [✓] .nedfolders
- [✓] .oppbuildspec
- [✓] .oppfeatures

Filter Types... | Select All | Deselect All

To archive file: C:\Users\SpitFire\src\veins_inet_5g.zip | Browse...

Options

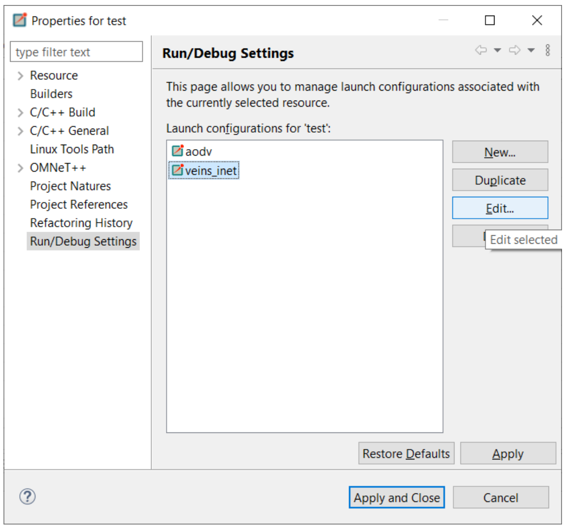
- [✓ Save in zip format]
- Save in tar format
- [✓] Compress the contents of the file
- [✓] Resolve and export linked resources
- [✓ Create directory structure for files]
- Create only selected directories

< Back | Next > | Finish | Cancel

(b) Export settings.

Figure 18: Exporting an existing project.

After importing the existing project back on the other computer, the project resources must be double checked. If an error (*Class "Veins::VeinsInetManager" not found – perhaps its code was not linked in, or the class wasn't registered with Register_Class(), or in the case of modules and channels, with Define_Module()/Define_Channel() – in module (omnetpp::cModule) Scenario (id=1), during network setup.)* arises while starting the simulation, probably run settings must be changed. For this, right click on the project folder and select Properties. Select Run/Debug settings. Select Edit and change the settings according to Fig. 10.



Properties for test

type filter text

- Resource
- Builders
- C/C++ Build
- C/C++ General
- Linux Tools Path
- OMNeT++
- Project Natures
- Project References
- Refactoring History
- Run/Debug Settings

**Run/Debug Settings**

This page allows you to manage launch configurations associated with the currently selected resource.

Launch configurations for 'test':

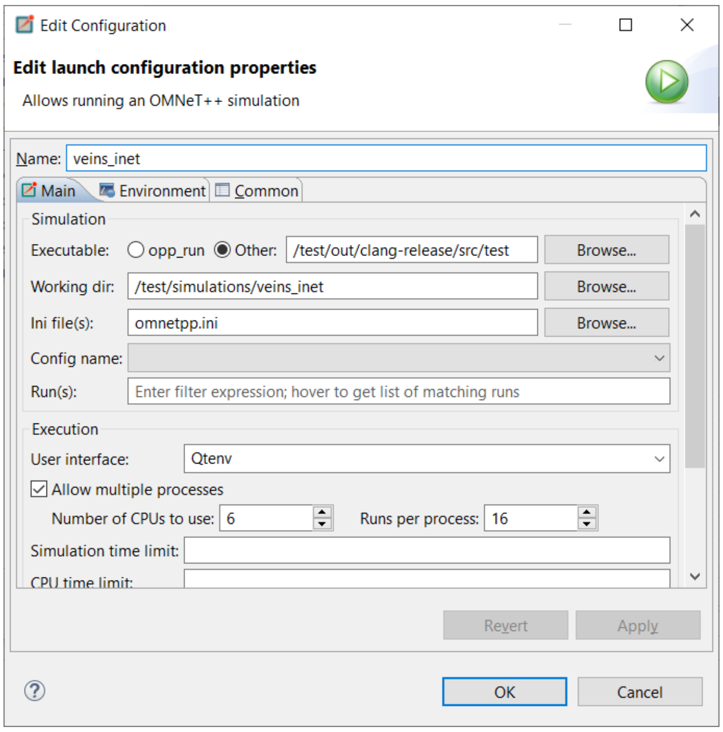
- aodv
- veins_inet

New... | Duplicate | Edit... | Edit selected

Restore Defaults | Apply

Apply and Close | Cancel

Figure 19: Run/Debug Settings.


Edit Configuration

**Edit launch configuration properties**

Allows running an OMNeT++ simulation

Name: veins_inet

Main | Environment | Common

Simulation

Executable: ( ) opp_run [✓ Other:] /test/out/clang-release/src/test

Working dir: /test/simulations/veins_inet

Ini file(s): omnetpp.ini

Config name:

Run(s): Enter filter expression; hover to get list of matching runs

Execution

User interface: Qtenv

☑ Allow multiple processes

Number of CPUs to use: 6 Runs per process: 16

Simulation time limit:

CPU time limit:

Figure 20: Veins_inet run settings.

Allowing multiple processes is not a must.

## 11 Obstacle shadowing

Creating a detailed simulation means, that the signal power loss caused by the objects in the environment, can not be neglected. For a Veins_INET project, the obstacle shadowing can be realized with the help of multiple modules. Be advised, that Veins 5.1 has a different visualizer and the obstacle shadowing example will not work with Veins 5.0. Nonetheless, it can be still realized, just have to use some other modules. The required modules, which has to be imported and used in the Scenario.ned file are the following:

- Ieee80211DimensionalRadioMedium
- PhysicalEnvironment
- PhysicalEnvironmentCanvasVisualizer

Veins 5.1 uses the IntegratedVisualizer, which is not compatible with Veins 5.0. It will stop the simulation with an error, telling that it can not visualize the radio medium for network nodes.

The Scenario.ned file should contain the following imports and modules:

```
// Required imports:
import inet.physicallayer.ieee80211.packetlevel.
    Ieee80211DimensionalRadioMedium;
import inet.visualizer.environment.PhysicalEnvironmentCanvasVisualizer;
import inet.environment.common.PhysicalEnvironment;

// Adding the modules
network Scenario
```

```
{
    @display("bgb=1466.245,943.0025;bgi=background/terrain,s");
    submodules:
        radioMedium: Ieee80211DimensionalRadioMedium {
            @display("p=1332.155,585.91504");

        physicalEnvironment: PhysicalEnvironment {
            @display("p=1332.155,690.85504");
        }
        visualizer: PhysicalEnvironmentCanvasVisualizer {
            @display("p=1332.155,480.975");

    // Other already used modules...

}
```

By adding these modules, the physical environment can be set up.

Settings can be defined in the simulation's omnetpp.ini file by adding the following lines:

```
#######################################################
# Obstacle parameters                                 #
#######################################################
*.physicalEnvironment.config = xmldoc("obstacles.xml")
*.radioMedium.physicalEnvironmentModule = "physicalEnvironment"
*.radioMedium.obstacleLoss.typename = "DielectricObstacleLoss"
*.radioMedium.obstacleLoss.enableDielectricLoss = true
*.radioMedium.obstacleLoss.enableReflectionLoss = true
*.visualizer.displayObjects = true
*.visualizer.physicalEnvironmentModule = "physicalEnvironment"
```

To add obstacles, a configuration file has to be created in the simulation folder. In this example, this is the obstacles.xml file. INET provides a detailed description of the physicalEnvironment module and its configuration here: `https://inet.omnetpp.org/docs/users-guide/ch-environment.html`

INET provides some default materials, which can be extended even in the configuration file or programatically in the *inet4/src/inet/environment/common/MaterialRegistry.cc* file. A material is defined by its dielectric properties:

- Resistivity
- Relative Permittivity
- Relative Permeability

By setting the radioMedium's obstacle loss module to use the DielectricObstacleLoss model, the signal's power loss will be calculated based on the material's properties and the propagation of the signal. This INET module also considers reflection losses. Instead of the DielectricObstacleLoss, the IdealObstacleLoss model can be also used, but it means a total signal loss, when the signal is shadowed by an obstacle.

Example configuration file:

```
<?xml version="1.0"?>
<environment>
    <object position="min 40 110 0" orientation="0 0 0" shape="cuboid 70 3
        20" material="concrete" fill-color="red" opacity="0.8" />
    <object position="min 130 110 0" orientation="0 0 0" shape="cuboid 70 3 20" material="concrete" fill-color="red" opacity="0.8" />

    <object position="min 40 140 0" orientation="0 0 0" shape="cuboid 70 3 20" material="concrete" fill-color="red" opacity="0.8" />
    <object position="min 130 140 0" orientation="0 0 0" shape="cuboid 70 3 20" material="concrete" fill-color="red" opacity="0.8" />

            <object position="min 110 40 0" orientation="0 0 0" shape="cuboid 3 70 20" material="concrete" fill-color="blue" opacity="0.8" />
    <object position="min 110 140 0" orientation="0 0 0" shape="cuboid 3 70 20" material="concrete" fill-color="blue" opacity="0.8" />

            <object position="min 130 40 0" orientation="0 0 0" shape="cuboid 3 70 20" material="concrete" fill-color="blue" opacity="0.8" />
    <object position="min 130 140 0" orientation="0 0 0" shape="cuboid 3 70 20" material="concrete" fill-color="blue" opacity="0.8" />
</environment>
```

This example provides the visualized obstacle shadowing depicted in Fig. 11

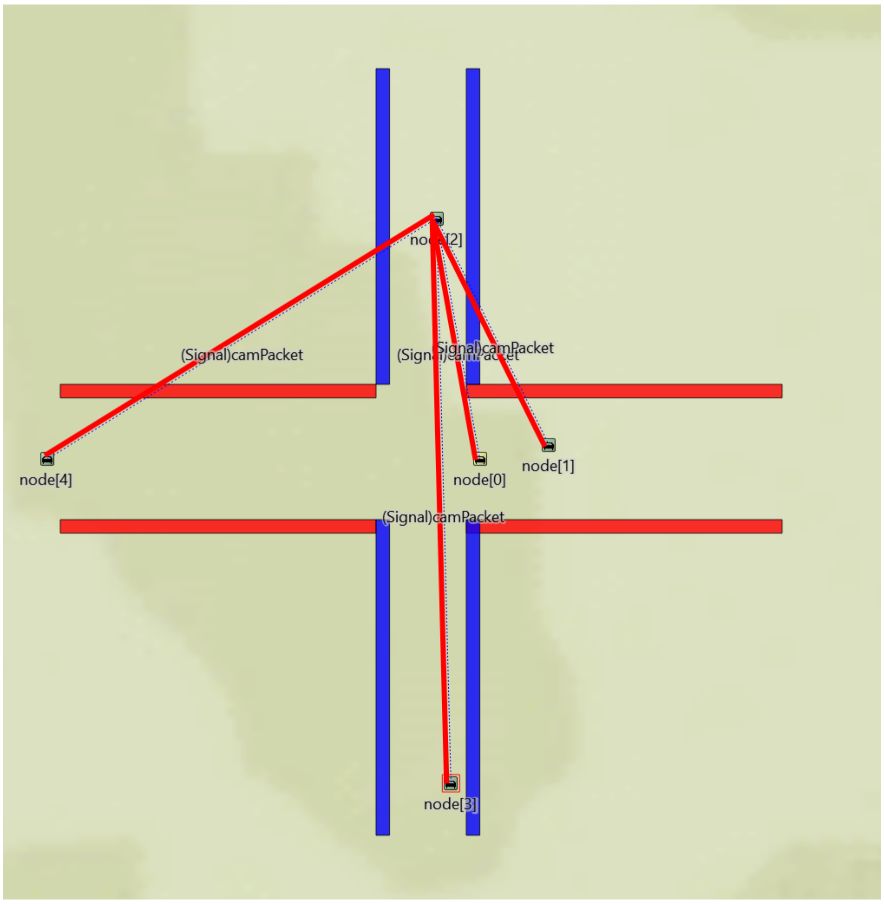


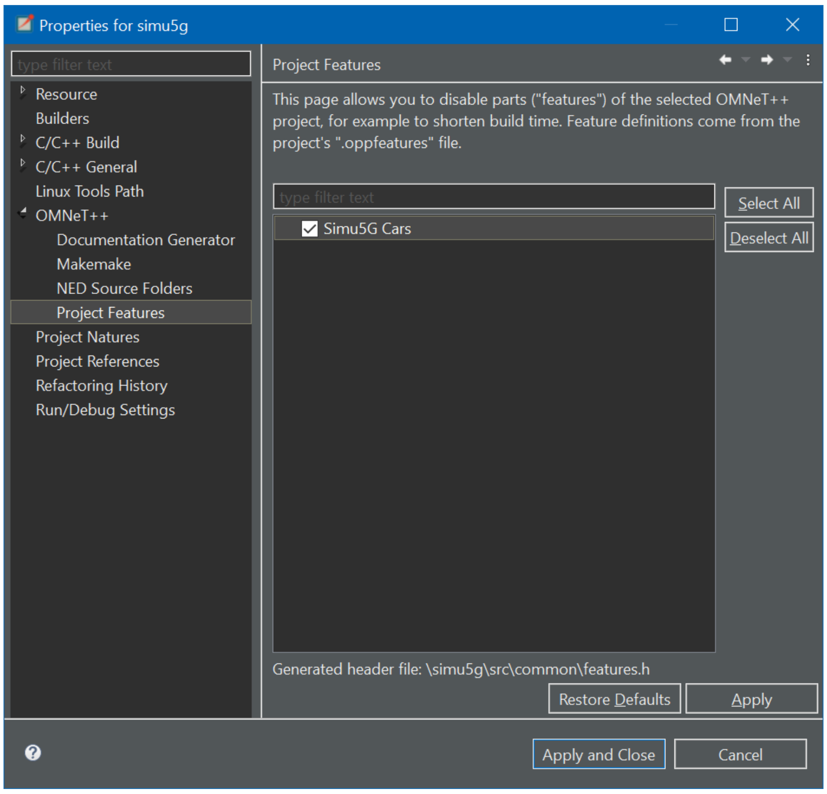
Figure 21: Veins_Obstacle shadowing.

# 12 Extending the system with Simu5G

## 12.1 Adding Simu5G to the custom project

To simulate 5G D2D (Device-to-Device) UE (User Equipment) communication through gNodeBs (5G wireless base stations) that transmit and receive communications between the user equipment and the mobile network, simply download it (`https://github.com/Unipisa/Simu5G/archive/refs/tags/v1.1.0.zip` v1.1.0) and import the folder in OMNeT++ as an existing project.

To use the 5G simulation, some modifications has to be done. First, the project has to be added as a resource to the custom project, by right clicking on the *tutorial* folder, selecting properties, and ticking simu5g under Project References. Simu5G references must be set also, by adding veins and INET as references of the simu5g project folder. The project has a Car feature, which also has to be enabled, by selecting the simu5g project folder in the project explorer by left clicking on it and then using the menu Project -> Project features. The Simu5G Cars feature can be enabled here.



This page allows you to disable parts ("features") of the selected OMNeT++ project, for example to shorten build time. Feature definitions come from the project's ".oppfeatures" file.

☑ Simu5G Cars

Generated header file: \simu5g\src\common\features.h

Figure 22: Veins_Enable cars feature.

In the custom makefile, the following lines has to be changed/added:

```
INCLUDE_PATH =  -I$(INET4_PROJ)/src -I. -I$(VEINS_PROJ)/src -I. -I$(
    SIMU5G_PROJ)/src -I.

LIBS = $(LDFLAG_LIBPATH)$(INET4_PROJ)/src $(LDFLAG_LIBPATH)$(VEINS_PROJ)/
    src C:/Users/src/omnetpp-5.6.2/tools/win64/libs/libasn1c.a -lINET$(D) -
    lveins$(D)  $(LDFLAG_LIBPATH)$(SIMU5G_PROJ)/src

SIMU5G_PROJ=../../simu5g
```

With these modifications, the custom project can use everything from the Simu5G framework.

Now the modifications for the Scenario.ned are the following (Note: In this case the project name was test instead of tutorial, see e.g. in imports.):

```
package test.simulations.veins_inet;

import inet.physicallayer.common.packetlevel.NoiseSource;
import test.veins_inet.VeinsInetManager;
import inet.physicallayer.ieee80211.packetlevel.
    Ieee80211DimensionalRadioMedium;
import test.veins_inet.VeinsInetCar;
import test.veins_inet.RSUstation;

// INET
import inet.networklayer.configurator.ipv4.Ipv4NetworkConfigurator;
import inet.networklayer.ipv4.RoutingTableRecorder;
import inet.node.inet.AdhocHost;
import inet.node.inet.Router;
import inet.node.inet.StandardHost;
import inet.node.ethernet.Eth10G;
import inet.networklayer.common.InterfaceTable;
import inet.networklayer.configurator.ipv4.Ipv4NodeConfigurator;

// Simu5G
import simu5g.world.radio.LteChannelControl;
import simu5g.common.carrierAggregation.CarrierAggregation;
import simu5g.nodes.Upf;
import simu5g.common.binder.Binder;
import simu5g.nodes.NR.gNodeB;
import simu5g.nodes.cars.Car;
import simu5g.nodes.PgwStandard;


network Scenario
{
    @display("bgb=2039.8379,1439.44;bgi=background/terrain,s");
    submodules:

        manager: VeinsInetManager {
            @display("p=1332.155,263.8075");
        }

        node[0]: VeinsInetCar {
            @display("p=531,287");
        }

        routingRecorder: RoutingTableRecorder {
            @display("p=1332.155,160.32501;is=s");
        }
        configurator: Ipv4NetworkConfigurator {
            @display("p=1332.155,371.6625");
            config = xmldoc("demo.xml");
        }

            //# LTE modules
        channelControl: LteChannelControl {
            @display("p=1329.5879,846.618;is=s");
        }
        binder: Binder {
            @display("p=1339.058,960.25793;is=s");
        }
        carrierAggregation: CarrierAggregation {
            @display("p=1331.4819,611.76196;is=s");
        }
        //          server: StandardHost {
        //              @display("p=990.56195,136.368;is=n;i=device/server");
        //          }
        //          router: Router {
        //              @display("p=797.37396,134.474;i=device/smallrouter");
        //          }
        //          upf: Upf {
        //              @display("p=787.904,443.19598;is=l");
        //          }
        //         pgw: PgwStandard {
        //              @display("p=645.85394,136.368;is=l");
        //          }
        gNodeB1: gNodeB {
            @display("p=681.83997,320.086;is=vl");
        }
        gNodeB2: gNodeB {
            @display("p=79.548,75.759995;is=vl");
        }

    connections allowunconnected:
//          server.pppg++ <--> Eth10G <--> router.pppg++;
//          router.pppg++ <--> Eth10G <--> pgw.filterGate;
//         pgw.pppg++ <--> Eth10G <--> gNodeB1.ppp;
//         pgw.pppg++ <--> Eth10G <--> gNodeB1.ppp;

        //# X2 connections
        gNodeB1.x2++ <--> Eth10G <--> gNodeB2.x2++;
}
```

The commented parts are optional. If they are used, the gNodeB stations are connected to a PGW/UPF (Packet data network gateway / User plane function).

The vehicle module has to be modified too, to behave like a 5G UE. For this, the VeinsInetCar.ned file should look like this:

```
package test.veins_inet;

import inet.networklayer.configurator.ipv4.HostAutoConfigurator;
import inet.node.inet.AdhocHost;

// SIMU5G
import inet.applications.contract.IApp;
import inet.mobility.contract.IMobility;
import inet.networklayer.common.InterfaceTable;
import inet.networklayer.contract.IRoutingTable;
import inet.networklayer.contract.INetworkLayer;
import inet.transportlayer.tcp.Tcp;
import inet.transportlayer.udp.Udp;
import inet.common.MessageDispatcher;
import simu5g.nodes.Ue;
import simu5g.nodes.NR.NRUe;



import org.car2x.veins.base.modules.*;

//
// Wireless-enabled Host AdhocHost if 802.11p and Ue if 5G
//
module VeinsInetCar extends Ue
{
    parameters:

        //# Mobility
       mobilityType = default("VeinsInetMobility");

        @display("i=device/cellphone;bgb=1283.625,951.3");
        ipv4.configurator.networkConfiguratorModule = ".ipv4.routingTable";
         //# Network Layer specs
        *.routingTableModule = default(absPath(".ipv4.routingTable"));
        ipv4.configurator.typename = "HostAutoConfigurator";
        ipv4.configurator.interfaces = "wlan";




    submodules:
            configurator: HostAutoConfigurator {
            @display("p=126.4,413.96");
        }
}
```

The last required modification is in the omnetpp.ini file. For this, simply add the following:

```
#######################################################
#                LTE specific parameters                        #
#######################################################
#
*.node[*].ipv4.configurator.typename="HostAutoConfigurator"

# LTE Radio Settings
*.channelControl.carrierFrequency = 5.925GHz
*.channelControl.propagationModel = "TwoRayGroundModel"

# Number of Resource Blocks
**.numBands = 25

# Transmission Power
**.ueTxPower = 26
**.eNodeBTxPower = 46


# Enable dynamic association of UEs (based on best SINR)
*.node[*].cellularNic.phy.dynamicCellAssociation = true
**.node[*].masterId = 1     # useless if dynamic association is disabled
**.node[*].macCellId = 1    # useless if dynamic association is disabled

# eNodeB configuration
**.gNodeB1.macCellId = 1
**.gNodeB1.macNodeId = 1
```

```
**.gNodeB2.macCellId = 2
**.gNodeB2.macNodeId = 2


# Enable handover
*.node[*].cellularNic.phy.enableHandover = true
*.gNodeB*.cellularNic.phy.enableHandover = true
*.gNodeB*.cellularNic.phy.handoverLatency = 50ms
*.gNodeB*.cellInfo.broadcastMessageInterval = 1s # eNB will sends broadcast
    triggers every second

# X2 and SCTP configuration
*.eNodeB*.numX2Apps = 1    # one x2App per peering eNodeB
*.eNodeB*.x2App[*].server.localPort = 5000 + ancestorIndex(1) # Server
   ports (x2App[0]=5000, x2App[1]=5001, ...)
*.eNodeB1.x2App[0].client.connectAddress = "eNodeB2%x2ppp0"
*.eNodeB2.x2App[0].client.connectAddress = "eNodeB1%x2ppp0"
**.sctp.nagleEnabled = false        # if true, transmission of small
   packets will be delayed on the X2
**.sctp.enableHeartbeats = false

# ---------------------------------------------- #
# Config "D2DMulticast"
#
# In this configuration, a transmitting car sends periodic alert messages
   to neighboring vehicles
#
[Config D2DMulticast]

### Enable D2D for the eNodeB and the UEs involved in direct communications
    ###
*.gNodeB*.nicType = "LteNicEnbD2D"
*.node[*].nicType = "LteNicUeD2D"
*.gNodeB*.cellularNic.LtePdcpRrcType = "LtePdcpRrcEnbD2D"
**.amcMode = "D2D"

# Channel model
*.*.cellularNic.LteChannelModelType = "LteRealisticChannelModel"

# Interfacetable display
*.gNodeB*.interfaceTable.displayAddresses = true
*.node[*].interfaceTable.displayAddresses = true
*.node[*].cellularNic.dualConnectivityEnabled = true
*.node[*].pcapRecorder[*].pcapFile = "frames.pcap"
### Select CQI for D2D transmissions ###
# One-to-Many communications work with fixed CQI values only.
# Set the parameter **.usePreconfiguredTxParams and select the desired CQI
   using the parameter **.d2dCqi
**.enableD2DCqiReporting = true
**.usePreconfiguredTxParams = true
**.d2dCqi = ${cqi=7}

### Traffic configuration: one-to-many traffic between UEs (car[0] --> car[
   x]) ###

# Transmitter
*.node[*].numApps = 2
*.node[*].app[0].typename = "AlertSender"
```

```
*.node[*].app[0].localPort = 3088+ancestorIndex(0)
*.node[*].app[0].startTime = uniform(0s,0.02s)
*.node[*].app[0].destAddress = "224.0.0.10"          # IP address of the
    multicast group
*.node[*].app[0].destPort = 1000

# Receivers (they must belong to the above multicast group)
*.node[*].app[1].typename = "AlertReceiver"
*.node[*].app[1].localPort = 1000

# enrolled multicast groups must be set in the HostAutoConfigurator (
    instead of demo.xml), seperated by a single space character
*.node[*].configurator.mcastGroups = "224.0.0.10"
```

With these settings, a D2D Multicasting communication is realized between vehicles by adding them to a multicast IP group. This configuration is using two applications. One is a message sender and one is a receiver. These applications can be programmed by the user in the simu5g project, under *src/apps/alert* by modifying *AlertSender.cc* and *AlertReceiver.cc*.

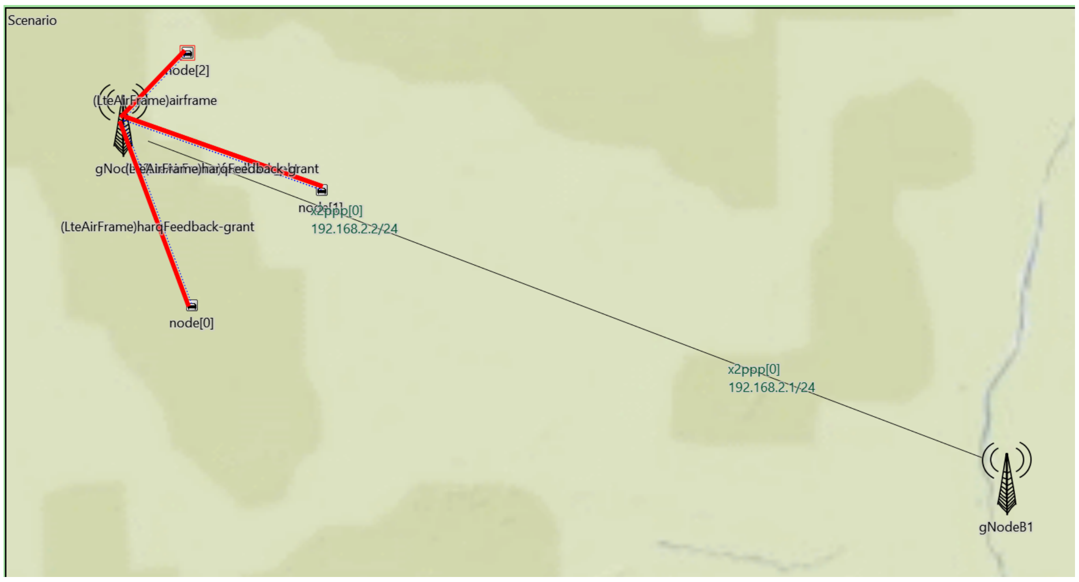


Figure 23: 5G Communication.

⚠ When building projects, simu5g will have an error (because it can not access veins), but this can be neglected.

## 12.2 Accessing TraCI commands with Simu5G

Simu5G works very easily with the custom made veins - inet project, however it does not provide an example of using the TraCI interface to interact with the SUMO simulation, which is an essential requirement if a control application has to be tested with the detailed communication simulation.

As it was already mentioned, the application layer is separated to a sender and a receiver application under *src/apps/alert*. By default, it could be just modified, but it could not access the TraCI interface. That is because it has no access to the veins_inet project as a reference. Unfortunately, adding it as a reference is not an option, as Omnet++ is not allowing any circular references between projects. This problem can be solved by creating two new Omnet++ modules in the custom veins_inet project. For this, copy the AlertSender and AlertReceiver source and header files to the custom project under *tutorial/src/veins_inet/*. Because this module is already existing in the Simu5G project, it has to be modified. Rename the copied files to e.g. *VeinsInetAlertSender.cc*, *VeinsInetAlertSender.h*, *VeinsInetAlertReceiver.cc* and *VeinsInetAlertReceiver.h*. In the same folder, the corresponding NED files must be created. These NED files are basically a copy of the original modules, but with a different path and name. The NEDs should look like this: (VeinsInetAlertSender.ned)

```
package tutorial.veins_inet;

import inet.applications.contract.IApp;

simple VeinsInetAlertSender like IApp
{
    parameters:
        int localPort = default(-1);
        int destPort = default(3000);
        string destAddress;
        int packetSize = default(10);
        volatile double period @unit("s") = default(0.02s);
        double startTime @unit("s") = default(0s);
        double stopTime @unit("s") = default(0s); // 0 means "never stops"
        string interfaceTableModule;
        string multicastInterface = default("wlan");  // if not empty, set
            the multicast output interface option on the socket (interface
            name expected)
        int tos = default(-1); // if not -1, set the Type Of Service (IPv4)
             / Traffic Class (IPv6) field of sent packets to this value

        @signal[alertSentMsg];
        @statistic[alertSentMsg](title="Alert messages sent"; unit="";
            source="alertSentMsg"; record=sum,vector);

        @display("i=block/source");
    gates:
        output socketOut;
        input socketIn;
}
```

(VeinsInetAlertReceiver.ned)

```
package tutorial.veins_inet;

import inet.applications.contract.IApp;

simple VeinsInetAlertReceiver like IApp
{
    parameters:
        int localPort = default(3000);
        string interfaceTableModule;
        string multicastInterface = default("wlan");  // if not empty, set
            the multicast output interface option on the socket (interface
            name expected)
        @signal[alertDelay];
        @statistic[alertDelay](title="Alert Message Delay"; unit="s";
            source="alertDelay"; record=mean,vector);
        @signal[alertRcvdMsg];
        @statistic[alertRcvdMsg](title="Alert Messages Received"; unit="s";
            source="alertRcvdMsg"; record=sum,vector);
        @display("i=block/source");
    gates:
        output socketOut;
        input socketIn;
}
```

⚠ It is important, to name the ned files the same as the source files were named!

Now, as a new module inside the custom project, it is possible to access the TraCI and mobility interfaces. For this, the *VeinsInetAlertSender and VeinsInetAlertReceiver* sources and headers have to be modified as the following.

VeinsInetAlertSender.cc:

```
#include <cmath>
#include <inet/common/ModuleAccess.h>
#include <inet/common/TimeTag_m.h>

#include "VeinsInetAlertSender.h"

#include "veins_inet\VeinsInetApplicationBase.h"
// CAM Structure
#include "build/sources/CAM.h"
// Math
#include <math.h>
// Time function dependencies
#include <chrono>
#include "date.h"
#include <cstdint>

#define round(x) floor((x) + 0.5)

Define_Module(VeinsInetAlertSender);
using namespace inet;
using namespace veins;

VeinsInetAlertSender::VeinsInetAlertSender()
{
    selfSender_ = nullptr;
    nextSno_ = 0;
}

VeinsInetAlertSender::~VeinsInetAlertSender()
{
    cancelAndDelete(selfSender_);
}

void VeinsInetAlertSender::initialize(int stage)
{
    mobility = veins::VeinsInetMobilityAccess().get(getParentModule()); //
        Access to the mobility module.
    traciVehicle = mobility->getVehicleCommandInterface(); // Access to the
        parent node's TraCI interface.
    EV << "AlertSender::initialize - stage " << stage << endl;

    cSimpleModule::initialize(stage);

    // avoid multiple initializations
    if (stage!=inet::INITSTAGE_APPLICATION_LAYER)
        return;

    selfSender_ = new cMessage("selfSender");

    size_ = par("packetSize");
    localPort_ = par("localPort");
    destPort_ = par("destPort");
    destAddress_ = inet::L3AddressResolver().resolve(par("destAddress").
        stringValue());

    socket.setOutputGate(gate("socketOut"));
    socket.bind(localPort_);

    int tos = par("tos");
    if (tos != -1)
        socket.setTos(tos);

    // for multicast support
    inet::IInterfaceTable *ift = inet::getModuleFromPar< inet::
        IInterfaceTable >(par("interfaceTableModule"), this);
    inet::MulticastGroupList mgl = ift->collectMulticastGroups();
    socket.joinLocalMulticastGroups(mgl);

    // if the multicastInterface parameter is not empty, set the interface
        explicitly
    const char *multicastInterface = par("multicastInterface");
    if (multicastInterface[0]) {
        InterfaceEntry *ie = ift->findInterfaceByName(multicastInterface);
        if (!ie)
            throw cRuntimeError("Wrong multicastInterface setting: no
                interface named \"%s\"", multicastInterface);
        socket.setMulticastOutputInterface(ie->getInterfaceId());
    }

    // -------------------- //

    alertSentMsg_ = registerSignal("alertSentMsg");

    EV << "AlertSender::initialize - binding to port: local:" << localPort_
        << " , dest:" << destPort_ << endl;

    // calculating traffic starting time
    simtime_t startTime = par("startTime");
    stopTime_ = par("stopTime");

    // TODO maybe un-necesessary
    // this conversion is made in order to obtain ms-aligned start time,
        even in case of random generated ones
    simtime_t offset = (round(SIMTIME_DBL(startTime)*1000)/1000)+simTime();

    scheduleAt(offset,selfSender_);
    EV << "\t starting traffic in " << offset << " seconds " << endl;
}

void VeinsInetAlertSender::handleMessage(cMessage *msg)
```

```
{
    if (msg->isSelfMessage())
    {
        if (!strcmp(msg->getName(), "selfSender"))
            sendAlertPacket();
        else
            throw cRuntimeError("AlertSender::handleMessage - Unrecognized
                self message");
    }
}

void VeinsInetAlertSender::sendAlertPacket()
{

    // The sender application comes here...
    alert->setSno(nextSno_);
    alert->setPayloadTimestamp(simTime());
    alert->setChunkLength(B(size_));
    alert->addTag<CreationTimeTag>()->setCreationTime(simTime());

    packet->insertAtBack(alert);

    EV << "AlertSender::sendAlertPacket - Sending message [" << nextSno_ <<
        "]\n";

    socket.sendTo(packet, destAddress_, destPort_);
    nextSno_++;

    emit(alertSentMsg_, (long)1);

    simtime_t d = simTime() + par("period");
    if (stopTime_ <= SIMTIME_ZERO || d < stopTime_) {
        scheduleAt(d, selfSender_);
    }
    else
        EV << "AlertSender::sendAlertPacket - Stop time reached, stopping
            transmissions" << endl;
}

void VeinsInetAlertSender::refreshDisplay() const
{
    char buf[80];
    sprintf(buf, "sent: %d pks", nextSno_);
    getDisplayString().setTagArg("t", 0, buf);
}
```

VeinsInetAlertSender.h:

```
#pragma once

#include <string.h>
#include <omnetpp.h>
#include <inet/transportlayer/contract/udp/UdpSocket.h>
#include <inet/networklayer/common/L3Address.h>
#include <inet/networklayer/common/L3AddressResolver.h>
#include "AlertPacket_m.h"

// Extension for CAM
```

```
#include "veins_inet/VeinsInetMobility.h"
#include "veins/modules/mobility/traci/TraCIScenarioManager.h"
#include "veins/modules/mobility/traci/TraCICommandInterface.h"
#include "build\sources\TimestampIts.h"

class VeinsInetAlertSender : public omnetpp::cSimpleModule
{
    inet::UdpSocket socket;

    //sender
    int nextSno_;
    int size_;

    omnetpp::simtime_t stopTime_;

    omnetpp::simsignal_t alertSentMsg_;
    // ----------------------------

    omnetpp::cMessage *selfSender_;

    int localPort_;
    int destPort_;
    inet::L3Address destAddress_;

    void sendAlertPacket();

  public:
    ~VeinsInetAlertSender();
       VeinsInetAlertSender();
       veins::TraCIScenarioManager* manager = veins::
           TraCIScenarioManagerAccess().get();
       veins::TraCICommandInterface* traci = manager->getCommandInterface()
           ;
       veins::VeinsInetMobility* mobility;
       veins::TraCICommandInterface::Vehicle* traciVehicle;

  protected:
    virtual int numInitStages() const override { return inet::
       NUM_INIT_STAGES; }
    void initialize(int stage) override;
    void handleMessage(omnetpp::cMessage *msg) override;

    // utility: show current statistics above the icon
    virtual void refreshDisplay() const override;
};
```

By including *CAM.h*, the CAM structure can be used just as like in the original 802.11p simulation. The same modifications has to be done in the *VeinsInetAlertReceiver* files (includes, define_module name and class name changes).

With the Simu5G application layer, message scheduling is different. Instead of using *sendPacket(std::move(packet));* function, provided by the *VeinsInetApplicationBase* and timed by a lambda function, it uses a *selfMessage* for message timing.